91-14 A. Fasbender / W. Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes.

91-15 J. Börstler / A. Zündorf: Revisiting extensions to Modula-2 to improve reusability.

91-16 J. Börstler / T. Janning: Bridging the gap between Requirements Analysis and Design

91-17 A. Zündorf / A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems

91-18 M. Jarke / J. Mylopoulos / J. W. Schmidt / Y. Vassiliou: DAIDA: An Environment for Evolving Information Systems

91-19 M. Jeusfeld / M. Jarke: From Relational to Object-Oriented Integrity Simplification

91-20 G. Hogen / A. Kindler / R. Loogen: Automatic Parallelization of Lazy Functional Programs

91-21 O. Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint

91-22 H. Kuchen / F. Lücking / H. Stoltze: The Topology Description Language TDL

91-23 S. Graf / B. Steffen: Compositional Minimization of Finite State Systems

91-24 R. Cleaveland / J. Parrow / B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems

91-25 R. Mathar/J. Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks

91-26 M. Jeusfeld/ M. Staudt: Query Optimazation in Deduktive Object Bases

91-27 J. Knoop/B. Steffen: The Interprocedural Coincidence Theorem

91-28 J. Knoop/B. Steffen: Unifying Strength Reduction and Semantic Code Motion

91-29 R. Cleaveland/M. Dreim"uller/B. Steffen: Faster Model Checking in the Model Mu-Calculus

91-30 T. Margaria: First-Order Theories for the Verification of Complex FSMs

89-12 P. Martini: The DQDB Protocol - Is it Playing the Game?

89-13 M. Schümmer: CNC / DNC Communication with MAP

89-14 M. Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

89-15 M. Schümmer / Th. Welzel / P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

89-16 G. Vossen / K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

89-17 J. Derissen / P. Hruschka / M.v.d.Beeck / Th. Janning / M. Nagl: Integrating Structured Analysis and Information Modelling

89-18 A. Maaßen: Programming with Higher Order Functions

89-19 M. Rodriguez-Artalejo / H. Vogler: Note on A Narrowing Machine for Syntax Directed BABEL

89-20 H. Kuchen / R. Loogen / J. J. Moreno-Navarro / M. Rodriguez-Artalejo Graph-based Implementation of a Functional Logic Language

89-i4 O. Spaniol: Natürliche Intelligenz oder: Der Parallel-Igel

89-21 G. Vossen / S. Wang: Correct and Efficient Deadlock Detection and Resolution in Distributed Database Systems

89-22 G. Vossen / R. Schwarz: Dynamic Scheduling of Transactions in Design Databases based on Version Consistency

90-1 Jahresbericht 1989

90-2 V. Jansen / A. Potthoff / W. Thomas / U. Wermuth: A Short Guide to the AMORE System (computing Automata, MOnoids, and Regular Expressions)

90-3 J. Skurczynski: On Three Hierarchies of Weak SkS Formulas

90-4 R. Loogen: Stack-based Implementation of Narrowing

90-5 H. Kuchen / A. Wagener: Comparison of Dynamic Load Balancing Strategies

90-6 K. Jakobs / F. Reichert: Directory Services for Mobile Communication

90-7 K. Jakobs: What's Beyond the Interface-OSI Networks to Support CooperativeWork

90-8 K. Jakobs: Directory Names and Schema - An Evaluation

90-9 U. Quernheim / D. Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

90-10 U. Quernheim / St. Jacobs / D. Kreuer: Adaptive ARQ-Fehlersicherungsverfahren für die Point-to-Multipoint-Satellitenkommunikation

90-11 J.J.Moreno-Navarro / H. Kuchen / R. Loogen / M. Rodriguez-Artalejo: Lazy Nar-rowing in a Graph Machine

90-12 K. Jakobs / J. Kaltwasser / F. Reichert / O. Spaniol: Der Computer fährt mit

90-13 R. Mathar / A. Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

90-14 A. Maaßen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung

90-15 M. Nagl / A. Schürr: A Specification Environment for Graph Grammars

90-16 A. Schürr: PROGRESS: a VHL-Language Based on Graph Grammars

90-17 M. Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

90-18 E. Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

90-19 F. Strauß: Erklärungsfähigkeit eines ebenenorientierten Expertensystemmodells

90-20 Y. Ortega-Mallen / D. de Frutos-Escrig: A Complete Proof System for Timed Observations

90-21 M. Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

90-22 H. Faßbender / H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

90-23 Th. Janning / M. Lefering: A Transformation from Requirements Engineering into Design - The Method and the Tool

90-24 J. Noack: Natürlichsprachliche Interaktion mit Datenbanken

90-25 R. Mathar: Concurrence Probabilities for a Locally Slotted Packet Radio Network by Combinatorial Methods

90-26 F. Herrmann / Heiko Vogler: Primitive Recursive Programs for Defining Tree Functions

90-27 Z. Fülöp / F. Herrmann / S. Vagvölgyi / H. Vogler: Tree Transducers with External Functions

90-28 P. Bachmann / W. Oberschelp / B. Thalheim / G. Vossen: The Design of RAD: Towards an Interactive Toolbox for Database Design

91-1 Jahresbericht 1990

91-2 F. Beuting / F. Piron / K. Rausch: Ein neuronales Modell zur Feststellung von für die visuelle Aufmerksamkeit relevanten Figureneigenschaften

91-3 B. Steffen / A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

91-4 M. Portz: A new class of cryptosystems based on interconnection networks

91-5 H. Kuchen / G. Geiler: Distributed Applicative Arrays

91-6 L. Staiger: Kolmogorov Complexity and Hausdorff Dimension

91-7 L. Staiger: Syntactic congruences for ω-languages

91-8 K. Jakobs / K. D. Kleefeld: Using Public Communication Services for Archaeological Applications

91-9 E. Kuikka: A Proposal for a Syntax-Directed Text Processing System

91-10 K. Gladitz / H. Faßbender / H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming

91-11 R. Loogen / S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages

91-12 K. Indermark / M. Rodríguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming

91-13 R. Hager / W. Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline.

Aachener Informatik-Berichte

Please write to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, W-5100 Aachen

87-1 Jahresbericht 1986
87-2 D. de Frutos Escrig / K. Indermark: Equivalence relations of non-deterministic Ianov-schemes
87-3 M. Nagl: A Software Development Environment Based on Graph Technology
87-4 C. Lewerentz / M. Nagl / B. Westfechtel: On Integration Mechanisms within a Graph-based Software Development Environment
87-5 R. Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
87-6 W. Damm / G. Döhmen: Specifying Distributed Computer Architectures in AADL*
87-7 G. Engels / C. Lewerentz / W. Schäfer: Graph Grammar Engineering: A Software Specification Method
87-8 M. Nagl: Set Theoretic Approaches to Graph Grammars
87-9 C. Lewerentz / A. Schürr: Experiences with a Database System for Software Documents
87-10 H. Klaeren / K. Indermark: A New Implementation Technique for Recursive Function Definitions
87-11 R. Loogen: Design of a parallel programmable graph reduction machine with distributed memory
87-12 J. Börstler / U. Möncke / R. Wilhelm: Table Compression for Tree Automata
87-13 C. Lewerentz: Extended Programming in the Large in a Software Development Environment
87-14 G. Engels / Th. Janning / W. Schäfer: A Highly Integrated Tool Set for Program Development Support
87-15 R. Loogen / U. Goltz: A non-interleaving semantic model for nondeterministic concurrent processes
87-16 Th. Hafer: On the Boolean Closure of Büchi Tree Automaton Definable Sets of w-Trees.
87-17 U. Heuter: First-Order Properties of Trees, Star-Free Expressions and Aperiodicity
88-1 G. Esser / J. Rückert / F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
88-2 P. Martini / O. Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
88-3 Th. Welzel: Simulation of a Multiple Token Ring Backbone
88-4 P. Martini: Performance Comparison for HSLAN Media Access Protocols
88-5 P. Martini: Performance Analysis of Multiple Token Rings
88-6 A. Mann / J. Rückert / O. Spaniol: Datenfunknetze
88-7 A. Mann / J. Rückert: Packet Radio Networks for Data Exchange

88-8 A. Mann / J. Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
88-9 W. Kremer / F. Reichert / J. Rückert / A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
88-10 K. Jakobs: Towards User-Friendly Networking
88-11 K. Jakobs: The Directory - Evolution of a Standard
88-12 K. Jakobs: Directory Services in Distributed Systems - A Survey
88-13 M. Schümmer: RS-511, a Protocol for the Plant Floor
88-14 U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing
88-15 P. Martini / O. Spaniol / Th. Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation
88-16 Jahresbericht 1987
88-17 W. Thomas: Automata on Infinite Objects
88-18 M. Sonnenschein: On Petri Nets and Data Flow Graphs
88-19 H. Vogler: Functional Description of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers
88-20 Th. Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen
88-21 Th. Janning / C. Lewerentz: Integrated Project Team Management in a Software Development Environment
88-22 J. Engelfriet / H. Vogler: Modular Tree Transducers
88-23 W. Thomas: Automata and Quantifier Hierarchies
88-24 U. Heuter: Generalized Definite Tree Languages
89-1 Jahresbericht 1988
89-2 G. Esser / J. Rückert / F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
89-3 H. Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions
89-4 A. Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language
89-5 J. Börstler: Wiederverwendbarkeit und Softwareentwicklung - Probleme, Lösungsansätze und Bibliographie
89-6 K. Jakobs: OSI - An Appropriate Basis for Group Communication?
89-7 K. Jakobs: ISO's Directory Proposal Evolution, Current Status and Future Problems
89-8 B. Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo / Redo and Revision Control
89-9 P. Martini: High Speed Local Area Networks - A Tutorial
89-00 Jozef Filser: Bolidische Gedangen (Gabidel 5): "Ieber die Kombjuderkunzt"
89-10 P. Davids / Th. Welzel: Performance Analysis of DQDB Based on Simulation
89-11 M. Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science, Rolduc (NL) near Aachen (D), June 14-16, 1989.

```
                END
            ELSE
              IF SR-Insert(ν)
                THEN
                  LET M = { ν' | SR-Redundant(ν, ν') }
                    BEGIN
                      IF M ≠ ∅
                        THEN
                          LET ν' ∈ M
                            BEGIN
                              "Attach the assignment
                              [ h_nr(ν) := h_nr(ν') ]
                              at the entry of node n"
                            END
                        ELSE
                          LET ν1, ν2 ∈ VFN. Distr(ν, ν1, ν2)
                            BEGIN
                              "Attach the assignment
                              [ h_nr(ν) := h_nr(ν1) + h_nr(ν2) ]
                              at the entry of node n"
                            END
                    END
                ELSE
                  LET op ∈ L_post(n)(ν↓1),
                      ν' ∈ VFN. ν'↓1 = l(ν↓1),
                      ν̄ ∈ VFN. ν̄↓1 = r(ν↓1)
                    BEGIN
                      "Attach the assignment
                      [ h_nr(ν) := h_nr(ν') op h_nr(ν̄) ]
                      at the entry of node n"
                    END
              FI
          FI
        END
    OD
OD;

( Introducing spill code for the generated auxiliary variables )
FORALL (ν, ν') ∈ { (ν̄, ν̿) ∈ VFE | nr(ν̄) ≠ 0 ∧ nr(ν̿) ≠ 0 } DO
  IF nr(ν) ≠ nr(ν')
    THEN
      "Attach a component of spill code
      [ (.., h_nr(ν'), ..) := (.., h_nr(ν), ..) ]
      at the exit of node 𝒩(ν)"
  FI
```



```
OD;

( Substituting original computations by a reference to their covering auxiliary
variable )
FORALL n ∈ N DO
  ( Let n ≡ (x1, ., xr) := (t1, ., tr) )
  FORALL ν ∈ {ν̄ ∈ VFN | 𝒩(ν̄) = n ∧ T_pre(n)(ν̄↓1) ∩ {t1, ., tr} ≠ ∅ } DO
    Replace all original computations t ∈ {t1, ., tr} ∩ T_pre(n)(ν↓1) by h_nr(ν)
  OD
OD.
```

$avail[\mu] := \mathtt{Insert}(\mu)$;
$workset := workset \cup \{\mu\}$
FI
OD
END
OD.

Algorithm A.8 (The Optimizing Program Transformation (Step 2(ii)))

INPUT: *A flow graph G, an associated value flow graph VFG, and the predicates* `Antloc`, `Insert`, `SR-Insert`, `SR-Usable`, `SR-Redundant`, *and* `Distr`.

OUTPUT: *The transformed flow graph* $G_{\mathbf{T}}$. *In* $G_{\mathbf{T}}$ *abstract values (stored in auxiliary variables) are computed at their earliest down-safe computation points by their minimal computation forms with respect to the value flow graph under consideration. Original computations are replaced by auxiliary variables.*

REMARK: *The algorithm consists of three phases, namely*

- *Marking the definition-use chains in VFG.*
- *Allocating auxiliary variable numbers.*
- *Transforming the flow graph:*
  - *Initializing auxiliary variables at their computation points by their computation forms.*
  - *Introducing spill code for the generated auxiliary variables.*
  - *Substituting original computations by their covering auxiliary variable.*

The predicate $mark(\nu)$ *indicates, whether a node* $\nu \in VFN$ *is marked or not,* $nr(\nu)$ *denotes the index of the auxiliary variable associated with* $\nu \in VFN$, *and* $count$ *denotes the highest index of an auxiliary variable currently valid. Note that "attaching an assignment at the entry of a node" implies that the following correctness conditions are satisfied. First, an attached assignment follows all initializations of its right hand side auxiliary variables, and second, it precedes all uses of its left hand side auxiliary variable in the insertion node under consideration.*

(Phase 1: Marking the definition-use chains in VFG)
FORALL $\nu \in VFN$ **DO** $mark(\nu) := false$ **OD**;
$workset := \{\nu \in VFN \mid \mathtt{Antloc}(\nu) \vee \mathtt{SR\text{-}Usable}(\nu) \vee \exists \nu' \in VFN.\ \mathtt{SR\text{-}Redundant}(\nu', \nu)\}$;
WHILE $workset \neq \emptyset$ **DO**
LET $\nu \in workset$
BEGIN
$workset := workset \setminus \{\nu\}$;
$mark(\nu) := true$;
IF $\neg\mathtt{Insert}(\nu)$
THEN $workset := workset \cup \{\nu' \in pred_{VFG}(\nu) \mid \neg\ mark(\nu')\}$ **FI**
END
OD;

(Phase 2: Allocating auxiliary variable numbers)
$count := 0$;
FORALL $\nu \in VFN$ **DO** $nr(\nu) := 0$ **OD**;
$workset := \{\nu \in VFN \mid \mathtt{Insert}(\nu)\}$;
WHILE $workset \neq \emptyset$ **DO**
LET $\nu \in workset$
BEGIN
$workset := workset \setminus \{\nu\}$;
$mark(\nu) := false$;
IF $\mathtt{Insert}(\nu) \vee |pred_{VFG}(\nu)| \geq 2$
THEN
$count := count + 1$;
$nr(\nu) := count$
ELSE *(assume* $pred_{VFG}(\nu) = \{\nu'\}$ *)*
$nr(\nu) := nr(\nu')$
FI;
$workset := workset \cup \{\nu' \in succ_{VFG}(\nu) \mid mark(\nu)\}$
END
OD;

(Phase 3: Transforming the flow graph)
(Initializing auxiliary variables at their computation points by their computation forms)
FORALL $n \in N$ **DO**
$workset := \{\nu \in VFN \mid \mathtt{Insert}(\nu) \wedge \mathcal{N}(\nu) = n\}$;
WHILE $workset \neq \emptyset$ **DO**
LET $\nu \in \{\nu' \in workset \mid \forall \bar{\nu} \in workset.\ \bar{\nu}\downarrow_1 \notin succ^*(\nu'\downarrow_1)\}$
BEGIN
$workset := workset \setminus \{\nu\}$;
IF $L_{\mathbf{pre}(n)}(\nu\downarrow_1) \cap (\mathbf{V} \cup \mathbf{C}) \neq \emptyset$
THEN
LET $x \in L_{\mathbf{pre}(n)}(\nu\downarrow_1) \cap (\mathbf{V} \cup \mathbf{C})$
BEGIN
"Attach the assignment
$\boxed{h_{nr(\nu)} := x}$
at the entry of node n"

Remark: *The fixed point iteration operates on the complete semi-lattice {false,true} with false $\sqsubset$ true. The variable counted-or allows an efficient update of the predicate d-safe, which depends on a disjunction of properties. The variable workset controls the iterative process, and the auxiliary variable d-safe stores the result of the most recent application of the local analysis component.*

(Initialization)
FORALL $\nu \in VFN$ **DO**
  $d\text{-}safe[\nu] := \mathcal{N}(\nu) \neq \mathbf{e}$;
  FORALL $n \in succ_G(\mathcal{N}(\nu))$ **DO**
    $counted\text{-}or_\nu(n) := |\{\mu \mid \mu \in succ_{VFG}(\nu).\ \mathcal{N}(\mu) = n \ \wedge\ d\text{-}safe[\mu]\}|$
  OD
OD;
$workset := \{\nu \mid \mathcal{N}(\nu) = \mathbf{e}\}$

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
  LET $\nu \in workset$
    BEGIN
      $workset := workset \setminus \{\nu\}$;
      (Update the "environment" of node ν)
      FORALL $\mu \in pred_{VFG}(\nu)$ **DO**
        $counted\text{-}or_\mu(\mathcal{N}(\nu)) := counted\text{-}or_\mu(\mathcal{N}(\nu)) - 1$;
        $d\text{-}safe := d\text{-}safe[\mu] \ \wedge\ (\mathtt{Antloc}(\mu) \ \vee\ (counted\text{-}or_\mu(\mathcal{N}(\nu)) \geq 1))$;
        IF $d\text{-}safe[\mu] \sqsupset d\text{-}safe$
          THEN
            $d\text{-}safe[\mu] := d\text{-}safe$;
            $workset := workset \cup \{\mu\}$
        FI
      OD
    END
OD.

Algorithm A.6 (Earliestness (Step 2(i)))

Input: *The Boolean Equation System 5.6.*

Output: *The least solution* `Earliest` *of Equation System 5.6 (stored in* $earliest[\nu]$*).*

Remark: *The fixed point iteration operates on the complete semi-lattice {false,true} with false $\sqsubset$ true. The variable workset controls the iterative process, and the auxiliary variable earliest stores the result of the most recent application of the local analysis component.*

(Initialization)
FORALL $\nu \in VFN$ **DO** $earliest[\nu] := \nu \in VFN_s$ **OD**;

$workset := VFN_s$

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
  LET $\nu \in workset$
    BEGIN
      $workset := workset \setminus \{\nu\}$;
      (Update the "environment" of node ν)
      FORALL $\mu \in succ_{VFG}(\nu)$ **DO**
        $earliest := earliest[\mu] \ \vee\ \neg\mathtt{D\text{-}Safe}(\nu)$;
        IF $earliest[\mu] \sqsubset earliest$
          THEN
            $earliest[\mu] := earliest$;
            $workset := workset \cup \{\mu\}$
        FI
      OD
    END
OD.

Algorithm A.7 (Availability (Step 2(i)))

Input: *The Boolean Equation System 5.11.*

Output: *The greatest solution* `Avail` *of Equation System 5.11 (stored in* $avail[\nu]$*).*

Remark: *The fixed point iteration operates on the complete semi-lattice {false,true} with false $\sqsubset$ true. The variable workset controls the iterative process.*

(Initialization)
FORALL $\nu \in VFN$ **DO**
  IF $\nu \in VFN_s$ **THEN** $avail[\nu] := \mathtt{Insert}(\nu)$ **ELSE** $avail[\nu] := true$ **FI**
OD;
$workset := \{\nu \mid \neg avail[\nu]\}$

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
  LET $\nu \in workset$
    BEGIN
      $workset := workset \setminus \{\nu\}$;
      (Update the "environment" of node ν)
      FORALL $\mu \in succ_{VFG}(\nu)$ **DO**
        IF $avail[\mu] \sqsupset \mathtt{Insert}(\mu)$
          THEN

```
      roots(Q) := determine-roots(Q);
      initialize-hashtables(P, (x_1, ., x_r) := (t_1, ., t_r));
      FORALL γ ∈ roots(Q) DO construct(γ, P, Q, true, VFN) OD;
      VFN := VFN ∪ { (γ, ⊙) | γ ∈ nodes(P) }
   OD;

   (Construction of the edges of the value flow graph)
   VFE := ∅;
   FORALL (m, n) ∈ E DO
      (Let P = (N_P, E_P, L_P) =_df post(m) and Q = (N_Q, E_Q, L_Q) =_df pre(n) )
      initialize-mark(Q);
      roots(Q) := determine-roots(Q);
      initialize-hashtables(P, skip);
      FORALL γ ∈ roots(Q) DO construct(γ, P, Q, false, VFE) OD
   OD
END.

where

PROCEDURE initialize-mark (D ∈ PD_fin);
BEGIN
   FORALL γ ∈ N_D DO mark(γ) := false OD
END;

FUNCTION determine-roots (D ∈ PD_fin) : SET OF DAGnodes;
VAR roots: SET OF DAGnodes;
BEGIN
   roots := ∅;
   FORALL γ ∈ N_D DO
      IF is-root(γ) THEN roots := roots ∪ {γ} FI
   OD;
   determine-roots := roots
END;

PROCEDURE construct (γ ∈ N_Q; P, Q ∈ PD_fin; b : Boolean;
                     VAR VFNE: SET OF VFGnodes or SET OF
VFGedges);
VAR father: DAGnodes;
BEGIN
   IF ¬ mark(γ)
      THEN
         mark(γ) := true;
         IF is-leaf(γ)
            THEN
```



```
               LET z ∈ L_Q(γ)
                  BEGIN
                     γ' := get-vc-node(z);
                     IF γ' ∈ N_P
                        THEN
                           VFNE := VFNE ∪ {(γ', γ)};
                           ptr(γ) := ↑γ';
                           IF b THEN nodes(P) := nodes(P)\{γ'} FI
                        ELSE
                           ptr(γ) := nil;
                           IF b THEN VFNE := VFNE ∪ {(⊙, γ)} FI
                     FI
                  END
            ELSE
               construct(l(γ), P, Q, b, VFNE);
               construct(r(γ), P, Q, b, VFNE);
               IF ptr(l(γ)) ≠ nil ∧ ptr(r(γ)) ≠ nil
                  THEN
                     father := get-op-node(L_Q(γ)∩Op, ptr(l(γ))↑, ptr(r(γ))↑);
                     IF father ∈ N_P
                        THEN
                           VFNE := VFNE ∪ {(father, γ)};
                           ptr(γ) := ↑father;
                           IF b THEN nodes(P) := nodes(P)\{father} FI
                        ELSE
                           ptr(γ) := nil;
                           IF b THEN VFNE := VFNE ∪ {(⊙, γ)} FI
                     FI
                  ELSE
                     ptr(γ) := nil;
                     IF b THEN VFNE := VFNE ∪ {(⊙, γ)} FI
               FI
         FI
   FI
END;
```

A.2 The Second Stage Algorithms

Algorithm A.5 (Down-Safety (Step 2(i)))

Input: *The Boolean Equation System 5.3.*

Output: *The greatest solution* `D-Safe` *of Equation System 5.3 (stored in d-safe[ν]).*

Output: *An annotation of G with sets of terms $\mathbf{T}_{suf}(n)$, whose elements represent values that* must *be computed on every continuation of paths from* $\mathbf{s}$ *to* n *that end in* $\mathbf{e}$.
Remark: *Term annotations are denoted by* $\mathbf{TD}$. *The operators* $\overset{n}{\wedge} : \mathcal{P}(\mathbf{T}) \times \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T}), n \in N$, *are defined as follows:* $\overset{n}{\wedge}(T_1, T_2) =_{df} \{t \in T_1 \mid \exists t' \in T_2.\ t$ *and* t' *are post[n]-equivalent* $\}$. *(The operators* $\overset{n}{\wedge}$ *can naturally be generalized to other numbers of arguments). Finally, for every node* n *of* G, *let* $\mathcal{R}(n)$ *denote a representation system of terms of the equivalence classes of the post-DAG of* n.

```
BEGIN
   FORALL n ∈ N  DO T_suf(n) := Φ(T(n)) OD;
   closure(e, T_suf);
   FORALL n ∈ N  DO T_suf(n) := T_suf(n) ∪ R(n) OD
END.
```

where

```
PROCEDURE closure(n ∈ N, VAR T_suf ∈ TD);
VAR T_suf' ∈ TD;
BEGIN
   FORALL m ∈ N  DO
      IF  n = m
         THEN  T_suf'(m) := T_suf(m)
         ELSE  T_suf'(m) := T
      FI
   OD;
   workset := pred_G(n);
   WHILE workset ≠ ∅  DO
      LET m ∈ workset
         BEGIN
            workset := workset \{m};
            IF T_suf'(m) ⊃ T_suf(m) ∪ Φ( δ_m( ∧^m_{m' ∈ succ_G(m)} (T_suf'(m'))) )
               THEN
                  T_suf'(m) := T_suf(m) ∪ Φ( δ_m( ∧^m_{m' ∈ succ_G(m)} (T_suf'(m'))) );
                  workset := workset ∪ pred_G(m)
            FI
         END
   OD;
   T_suf := T_suf'
END;
```


Algorithm A.4 (Construction of the Value Flow Graph (Step 1(iii)))

Input: *A flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$, *whose nodes are annotated with structured partition DAGs as pre-DAGs and post-DAGs.*

Output: *The value flow graph according to the node annotation above.*

Remark: *The following auxiliary procedures and variables are used:*

- *initialize–hashtables: generates two hashtables for the structured partition DAG* P. *The first one allows to find every inner node of* P *in constant time, and the second one to find that node of* P, *which is labeled with a special constant or variable in constant time; the second parameter of this procedure controls whether the reference depends on a backward substitution (this is necessary for the construction of nodes).*
- *get–vc–node: given a variable or constant as argument, get–vc–node yields that node of* P *labeled by the argument if such a node exists; otherwise get–vc–node delivers a special node not in* N_P.
- *get–op–node: given two nodes of* N_P *and an operator symbol, get–op–node yields that node of* P, *which is labeled by the operator and whose sons are the two argument nodes if such a node exists; otherwise get–op–node delivers a special node not in* N_P.
- *is–leaf: given a node of a structured partition DAG, is–leaf determines whether this node is a leaf or not.*
- *is–root: given a node of a structured partition DAG, is–root determines whether this node is a root or not.*
- *mark*(γ), *ptr*(γ): *with every node* γ *of the structured partition DAG* Q *the auxiliary variables mark*(γ) *and ptr*(γ) *are associated. The first variable indicates whether the node* γ *has already been processed, the second variable contains a pointer (indicated by* $\uparrow$*) to that node of* P *which represents a superset of terms represented by* γ, *if such a node exists. Otherwise, ptr*(γ) *will be set to "nil".*
- *roots*(Q), *nodes*(P): *denote two auxiliary variables of the type set of DAGnodes.*

```
BEGIN
   (Construction of the nodes of the value flow graph)
   VFN := ∅;
   FORALL  n ∈ N  DO
      (Let P = (N_P, E_P, L_P) =_df pre(n) and Q = (N_Q, E_Q, L_Q) =_df post(n) and
      let n ≡ (x_1, ., x_r) := (t_1, ., t_r) )
      nodes(P) := N_P;
      initialize–mark(Q);
```

PROCEDURE *reconstruct* $(\gamma \in N_Q)$;
BEGIN
  FORALL $\gamma' \in pred(\gamma)$ **DO**
    $L_Q(\gamma') := L_Q(\gamma') \setminus \mathbf{Op}$;
    $E_Q := E_Q \setminus \{(\gamma', \gamma'') \mid \gamma'' \in succ(\gamma')\}$;
    IF $L_Q(\gamma') = \emptyset$ **THEN** *reconstruct*(γ') **FI**
  OD;
  $N_Q := N_Q \setminus \{\gamma\}$
END;

Algorithm A.2 (The "$\sqcap$"–Algorithm for Structured Partition DAGs)

Input: *Two finite structured partition DAGs* $P = (N_P, E_P, L_P)$ *and* $Q = (N_Q, E_Q, L_Q)$.
Output: *A finite structured partition DAG* $R = (N_R, E_R, L_R)$, *which represents the meet of* P *and* Q, *i.e.* $R = P \sqcap Q$. *Additionally,* R *satisfies:* $\mathbf{T}(R) = \mathbf{T}(P) \cap \mathbf{T}(Q)$.

Remark: *Nodes of* R *are denoted by pairs* (γ, γ'), *where* $\gamma \in N_P$ *and* $\gamma' \in N_Q$. *With every pair* $(\gamma, \gamma') \in N_P \times N_Q$ *a Boolean variable* $jstfd(\gamma, \gamma')$ *is associated. (Note that* $jstfd(\gamma, \gamma')$ *is initialized with a special value "undefined").*

BEGIN
  $N_R := \emptyset$;
  $E_R := \emptyset$;
  initialize–jstfd(P, Q);
  FORALL $(\gamma, \gamma') \in N_P \times N_Q$. $L_P(\gamma) \cap L_Q(\gamma') \neq \emptyset$ **DO**
    IF $L_P(\gamma) \cap L_Q(\gamma') \cap \mathbf{Op} \neq \emptyset$
      THEN
        IF *justified*$(l(\gamma), l(\gamma'))$ $\wedge$ *justified*$(r(\gamma), r(\gamma'))$
          THEN
            $N_R := N_R \cup \{(\gamma, \gamma')\}$;
            $L_R((\gamma, \gamma')) := L_P(\gamma) \cap L_Q(\gamma')$
          ELSE
            IF $(L_P(\gamma) \cap L_Q(\gamma')) \setminus \mathbf{Op} \neq \emptyset$
              THEN
                $N_R := N_R \cup \{(\gamma, \gamma')\}$;
                $L_R((\gamma, \gamma')) := (L_P(\gamma) \cap L_Q(\gamma')) \setminus \mathbf{Op}$
            FI
        FI
      ELSE
        $N_R := N_R \cup \{(\gamma, \gamma')\}$;
        $L_R((\gamma, \gamma')) := L_P(\gamma) \cap L_Q(\gamma')$
    FI
  OD;

  FORALL $(\gamma, \gamma') \in N_R$ **DO**
    IF $L_R((\gamma, \gamma')) \cap \mathbf{Op} \neq \emptyset$
      THEN $E_R := E_R \cup \{((\gamma, \gamma'), (l(\gamma), l(\gamma'))), ((\gamma, \gamma'), (r(\gamma), r(\gamma')))\}$
    FI
  OD
END.

where

PROCEDURE *initialize–jstfd* $(P, Q \in \mathcal{PD}_{fin})$;
BEGIN
  FORALL $(\gamma, \gamma') \in N_P \times N_Q$ **DO** *jstfd*(γ, γ') := *false* **OD**
END;

FUNCTION *justified* $(\gamma \in N_P,\ \gamma' \in N_Q)$: *Boolean*;
BEGIN
  IF *jstfd*(γ, γ') = *undefined*
    THEN *compute–justified*(γ, γ') **FI**;
  justified := *jstfd*(γ, γ')
END;

PROCEDURE *compute–justified* $(\gamma \in N_P,\ \gamma' \in N_Q)$;
BEGIN
  IF $L_P(\gamma) \cap L_Q(\gamma') = \emptyset$
    THEN
      jstfd(γ, γ') := *false*
    ELSE
      IF $L_P(\gamma) \cap L_Q(\gamma') \cap (\mathbf{V} \cup \mathbf{C}) \neq \emptyset$
        THEN *jstfd*(γ, γ') := *true*
        ELSE *jstfd*(γ, γ') := *justified*$(l(\gamma), l(\gamma'))$ $\wedge$ *justified*$(r(\gamma), r(\gamma'))$
      FI
  FI
END;

Algorithm A.3 (Computation of Relevant Terms (Step 1(ii)))

Input: *A flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$, *whose nodes are annotated with the pre-DAGs and post-DAGs computed by Algorithm 4.10.*

```
BEGIN
  Q := P;
  initialize-hashtables(Q);
  FORALL T ∈ {T_1, ..., T_r} DO
    initialize-mark(T);
    root(T) := determine-roots(T);
    extend-preDAG(root(T))
  OD;
  execute-assign
END.

where

PROCEDURE initialize-mark (D ∈ PD_fin);
BEGIN
  FORALL γ ∈ N_D DO mark(γ) := false OD
END;

FUNCTION determine-roots (D ∈ PD_fin) : SET OF DAGnodes;
VAR roots: SET OF DAGnodes;
BEGIN
  roots := ∅;
  FORALL γ ∈ N_D DO
    IF is-root(γ) THEN roots := roots ∪ {γ} FI
  OD;
  determine-roots := roots
END;

PROCEDURE extend-preDAG (γ ∈ N_T);
VAR left, right, father: DAGnodes;
BEGIN
  IF ¬ mark(γ)
    THEN
      mark(γ) := true;
      IF is-leaf(γ)
        THEN
          LET z ∈ L_T(γ)
            BEGIN
              IF get-vc-node(z) ∈ N_Q
                THEN
                  ptr(γ) := ↑get-vc-node(z)
                ELSE
                  N_Q := N_Q ∪̇ {γ'};
                  L_Q(γ') := L_T(γ);
                  ptr(γ) := ↑γ';
                  update-hashtables(γ')
              FI
            END
        ELSE
          extend-preDAG(l(γ));
          extend-preDAG(r(γ));
          left := ptr(l(γ))↑;
          right := ptr(r(γ))↑;
          father := get-op-node(L_T(γ)∩Op ,left,right);
          IF father ∈ N_Q
            THEN
              ptr(γ) := ↑father
            ELSE
              N_Q := N_Q ∪̇ {γ'};
              L_Q(γ') := L_T(γ);
              E_Q := E_Q ∪ { (γ',left), (γ',right) };
              ptr(γ) := ↑γ';
              update-hashtables(γ')
          FI
      FI
  FI
END;

PROCEDURE execute-assign;
VAR γ : DAGnodes;
    Γ : SET OF DAGnodes;
BEGIN
  Γ := ∅;
  FORALL x ∈ {x_1, ., x_r} DO
    γ := get-vc-node(x);
    IF γ ∈ N_Q
      THEN
        L_Q(γ) := L_Q(γ)\{x};
        Γ := Γ ∪ {γ}
    FI;
    (Let T be the right hand side term corresponding to x)
    L_Q(ptr(root(T))↑) := L_Q(ptr(root(T))↑) ∪ {x}
  OD;
  FORALL γ ∈ Γ DO
    IF L_Q(γ) = ∅ THEN reconstruct(γ) FI
  OD
END;
```

[KU] J. B. Kam and J. D. Ullman. *"Monotone Data Flow Analysis Frameworks"*. Acta Informatica 7, 309 - 317, 1977

[MR] E. Morel and C. Renvoise. *"Global Optimization by Suppression of Partial Redundancies"*. Communications of the ACM, 22(2):96-103, 1979

[Pa1] R. Paige. *"Formal Differentiation - A Program Synthesis Technique"*. UMI Research Press, 1981.

[Pa2] R. Paige. *"Transformational Programming – Applications to Algorithms and Systems"*. 10^{th} POPL, Austin, Texas, 73 - 87, 1983

[PK] R. Paige and S. Koenig. *"Finite Differencing of Computable Expressions"*. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, 402 - 454, 1982

[RWZ] B. K. Rosen, M. N. Wegman and F. K. Zadeck. *"Global Value Numbers and Redundant Computations"*. 15^{th} POPL, San Diego, California, 12 - 27, 1988

[St1] B. Steffen. *"Optimal Run Time Optimization. Proved by a New Look at Abstract Interpretations"*. 2^{nd} TAPSOFT, Pisa, Italy, LNCS 249, 52 - 68, 1987

[St2] B. Steffen. *"Abstrakte Interpretationen beim Optimieren von Programmlaufzeiten. Ein Optimalitätskonzept und seine Anwendung"*. PhD thesis, Christian-Albrechts-Universität Kiel, Germany, 1987

[St3] B. Steffen. *"Data Flow Analysis as Model Checking"*. In Proceedings TACS'91, LNCS 426, 346 - 364, Sendai, Japan, 1991.

[SK1] B. Steffen and J. Knoop. *"Finite Constants: Characterizations of a New Decidable Set of Constants"*. 14^{th} MFCS, Porąbka-Kozubnik, Poland, LNCS 379, 481 - 491, 1989

[SK2] B. Steffen and J. Knoop. *"Finite Constants: Characterizations of a New Decidable Set of Constants"*. Extended version of [SK1]. TCS 80, 303 - 318, 1991

[SKR1] B. Steffen, J. Knoop and O. Rüthing. *"The Value Flow Graph: A Program Representation for Optimal Program Transformations"*. 3^{rd} ESOP, Copenhagen, Denmark, LNCS 432, 389 - 405, 1990

[SKR2] B. Steffen, J. Knoop and O. Rüthing. *"Efficient Code Motion and an Adaption to Strength Reduction"*. 4^{th} TAPSOFT, Brighton, United Kingdom, LNCS 494, 394 - 415, 1991

A Appendix

A.1 The First Stage Algorithms

Algorithm A.1 (The Local Analysis Component "$[\![\ ]\!]$")

Input: *A node n of a flow graph G representing the parallel assignment $(x_1,.,x_r) := (t_1,.,t_r)$, where the right hand side terms $t_1,.,t_r$ are represented as structured partition DAGs $T_i = (N_{T_i}, E_{T_i}, L_{T_i})$, and a finite structured partition DAG $P = (N_P, E_P, L_P)$, which represents the valid term equivalences before the execution of node n.*

Output: *The finite structured partition DAG $Q = (N_Q, E_Q, L_Q)$ defined by $[\![n]\!](P)$.*

Remark: *The following auxiliary procedures and variables are used:*

- *initialize–hashtables: generates two hashtables for Q. The first one allows to find every inner node of Q in constant time, and the second one to find that node of Q which is labeled with a special constant or variable in constant time.*
- *get–vc–node: given a variable or constant as argument, get–vc–node yields that node of Q, which is labeled by the argument if such a node exists; otherwise get–vc–node delivers a special node not in N_Q.*
- *get–op–node: given two nodes of N_Q and an operator symbol, get–op–node yields that node of Q, which is labeled by the operator and whose sons are the two argument nodes if such a node exists; otherwise get–op–node delivers a special node not in N_Q.*
- *update–hashtables: having extended Q by a new node, update–hashtables updates the hashtables mentioned above.*
- *is–leaf: given a node of a structured partition DAG, is–leaf determines whether this node is a leaf or not.*
- *is–root: given a node of a structured partition DAG, is–root determines whether this node is a root or not.*
- *mark(γ), ptr(γ): with every node γ of a right hand side term the auxiliary variables mark(γ) and ptr(γ) are associated. The first variable indicates whether the node γ has already been processed, the second variable contains a pointer (indicated by $\uparrow$) to that node of Q labeled with the term which is represented by node γ.*
- *root(T), roots(Q): denote two auxiliary variables of the type DAGnode and set of DAGnodes, respectively (here we identify a set with one element with its element).*
- *pred(γ) $=_{df} \{\gamma' \mid (\gamma', \gamma) \in E_Q\}$ and succ(γ) $=_{df} \{\gamma' \mid (\gamma, \gamma') \in E_Q\}$ denote the set of all predecessors and successors of a node $\gamma \in N_Q$, respectively.*

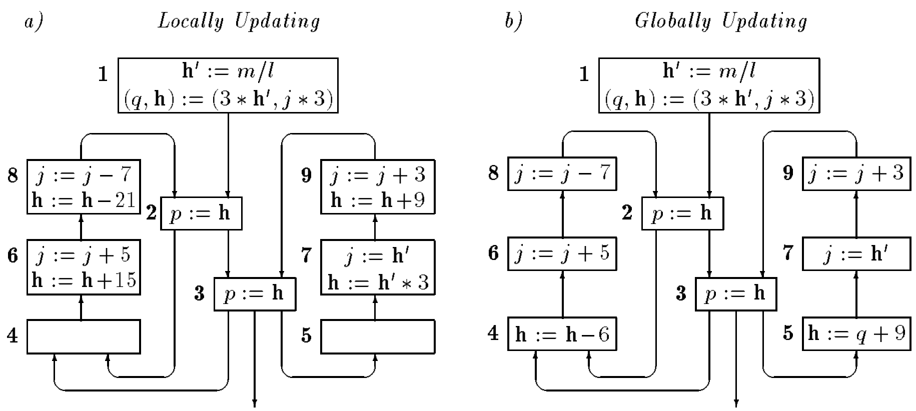

Figure 6.2

7 Conclusions

We have presented an algorithm that uniformly combines strength reduction and semantic code motion on the basis of optimally moving computations within programs with respect to the Herbrand equivalence. In addition to the algorithm of [SKR1], which generalizes and improves previous techniques for common subexpression elimination, partial redundancy elimination, and loop invariant code motion, the new algorithm also improves on all classical techniques for strength reduction in that it overcomes their restrictions concerning admissible program structures (previously detected loops) and admissible term structures (built of induction variables and region constants).

The development of this algorithm profited from the modular structure of the underlying code motion algorithm (cf. [SKR1]), which has been maintained in this extension. This supports further extensions. For example, it is straightforward to extend the algorithm's strength reduction capacity to further operators, e.g. to capture the reduction of exponentiations to multiplications[24]. Moreover, extensions to further optimization goals like ***constant propagation*** and ***constant folding*** ([SK1,SK2]) can be obtained by strengthening the analyis capacity of the first stage. We are also investigating an adaption of the "lazy" placing strategy realized by the syntactically based code motion and strength reduction algorithms of [KRS1] to the semantically based algorithm here in order to minimize the lifetimes of auxiliary variables.

However, without this assumption it would insert the assignment $\mathbf{h} := j * 3$ on the edge leaving node **8** instead of the insertions at node **6** and **8**. And in this case, there would be no positive effect on the run-time efficiency at all.

[24]More examples can be found in [ACK].

Acknowledgements

The presentation in this paper profited from discussions with Robert Paige.

References

[All] F. E. Allen. *"Control Flow Analysis"*. ACM Sigplan Notices, July 1970

[ACK] F. E. Allen, J. Cocke and K. Kennedy. *"Reduction of Operator Strength"*. In: St. S. Muchnick and N. D. Jones (Eds.). "Program Flow Analysis: Theory and Applications", Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1981

[Ch] G. J. Chaitin. *"Register Allocation and Spilling via Graph Coloring"*. SIGPLAN Notices, 17(6):98-105, 1982

[CACCHM] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein. *"Register Allocation via Coloring"*. Computer Languages Vol. 6, 47 - 57, 1981

[CK] J. Cocke and K. Kennedy. *"An Algorithm for Reduction of Operator Strength"*. Communications of the ACM, 20(11):850-856, 1977

[CP] J. Cai and R. Paige. *"Look Ma, No Hashing, And No Arrays Neither"*. 18^{th} POPL, Orlando, Florida, 1991

[FKU] A. Fong, J. B. Kam and J. D. Ullman. *"Application of Lattice Algebra to Loop Optimization"*. 2^{nd} POPL, Palo Alto, California, 1 - 9, 1975

[JD1] S. M. Joshi and D. M. Dhamdhere. *"A Composite Hoisting-Strength Reduction Transformation for Global Program Optimization – Part I"*. Internat. J. Computer Math. 11, 21 - 41, 1982

[JD2] S. M. Joshi and D. M. Dhamdhere. *"A Composite Hoisting-Strength Reduction Transformation for Global Program Optimization – Part II"*. Internat. J. Computer Math. 11, 111 - 126, 1982

[Ki1] G. A. Kildall. *"Global Expression Optimization during Compilation"*. Technical Report No. 72-06-02, University of Washington, Computer Science Group, Seattle, Washington, 1972

[Ki2] G. A. Kildall. *"A Unified Approach to Global Program Optimization"*. 1^{st} POPL, Boston, Massachusetts, 194 - 206, 1973

[KRS1] J. Knoop, O. Rüthing, and B. Steffen. *"Lazy Code Motion"*. Submitted to SIGPLAN'92

[KS] J. Knoop, and B. Steffen. *"Efficient and Optimal Interprocedural Bit-Vector Data Flow Analyses: A Uniform Interprocedural Framework"*. Submitted to ICCL'92

replace the initialization term of $\nu_1\!\downarrow_1$ by the auxiliary variable storing the value of ν_2 at the insertion point.

This procedure avoids the introduction of totally redundant computations caused by the strength reduction transformation, however, it does not avoid the introduction of partially redundant computations, a problem we are currently investigating.

Splitting of initializations: Initializing different auxiliary variables at the same insertion point with abstract values that are sharing common subvalues, may cause the introduction of local redundancies. This, however, can easily be avoided by splitting all initializations into sequences of assignments that only have a single operator in their right hand side expression.

Computations costs: As usual for code motion, our algorithm is devoted to the costs of computations. Costs of register loading and storing, however, are subsequently taken care of by variable subsumption and dead code elimination. An algorithm based on the graph coloring techniques of [Ch,CACCHM] is implemented for this purpose.

6 Related Work

Strength reduction was pioneered by Cocke and Kennedy [CK][20] and later on generalized and improved in particular by Allen, Cocke and Kennedy [ACK], and Joshi and Dhamdhere [JD1,JD2]. All these approaches, which characterize the state of the art, are:

- *Syntactic*: they optimize term by term, without exploiting semantic equivalences between syntactically different terms.
- *Locally updating*: they insert update assignments whenever an operand of a candidate expression for strength reduction is redefined, without investigating the global context for the necessity of this update. This may introduce terms, whose values are not computed in the original program. Thus, the resulting program transformation *cannot* be guaranteed to be *safe* or to *improve* run-time efficiency.
- *Structurally restricted*: [CK,ACK] work only for previously detected loops and terms built from induction variables and region constants, which excludes the optimization of more general program structures. In contrast, [JD1,JD2] work for arbitrary control flow structures and terms composed of variables and program constants. They pay for their ability to deal with general program structures by requiring an unnecessarily strong notion of admissible term structure.

In contrast, our algorithm is:

- *Semantic*: it exploits semantic equivalence between syntactically different terms.
- *Globally updating*: update assignments are only inserted, if they are required by the global context. This guarantees that the resulting program transformation is *safe* and that it *improves* the run-time efficiency of the original program.

[20] An efficient, hash-free solution to the strength reduction transformation of [CK] is presented in [CP].

- *General*: it works for arbitrary program structures without requiring additional constraints concerning admissible term structures[21].

To our knowledge, none of these points has been realized in a strength reduction algorithm before. In fact, also the (significantly different) approach of [Pa2,PK], the ***finite differencing***, fails these points. Its major achievement is the generalization of strength reduction to non-numerical applications, which we do not consider here.

Whereas the predicates "syntactic", "semantic", "structurally restricted" and "general" are self-explaining, "locally updating" and "globally updating" need some more explanation. We will therefore illustrate these two predicates by means of a simplified version of an example given in [JD1]:

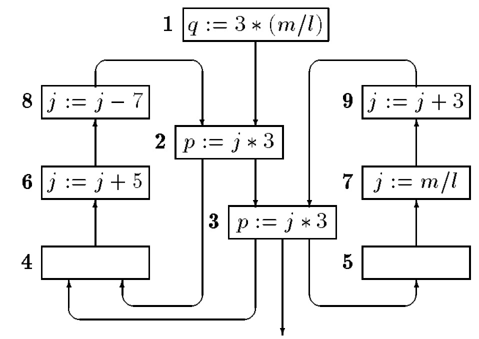

Figure 6.1

In the flow graph above the computation of $j{*}3$ in node **2** and **3** is a candidate expression for strength reduction. Local updating requires to insert for *every* redefinition of an operand of a candidate expression e a redefinition of the auxiliary variable $\mathbf{h}$. This guarantees $\mathbf{h}$ always to hold the value of e, even at places where it is not going to be used. Therefore, a locally updating transformation cannot arrive at a better result than the flow graph shown in Figure 6.2(a)[22], where a computation is introduced, whose value is not computed in the original program, namely the value of the computation of $\mathbf{h}+15$ at node **6**. Hence, the transformation is *not* safe. Moreover, it even impairs the run-time efficiency: on path $(\mathbf{3},\mathbf{5},\mathbf{7},\mathbf{9})$ one multiplication is saved, but a multiplication and an addition is inserted, and on path $(\mathbf{2},\mathbf{4},\mathbf{6},\mathbf{8})$ a multiplication is saved on the costs of two inserted additions, whose added computation costs may exceed those of the saved multiplication[23]. Our algorithm is unique in producing the flow graph of Figure 6.2(b).

[21] We do not even need region constants, because strength reduction is completely reduced to the availability of (abstract) values in our algorithm.

[22] This transformation is realized by the algorithms of [JD1,Pa1,Pa2,PK], whereas the algorithms of [ACK,CK,CP] leave the flow graph unchanged, because j is due to the assignment $j := m/l$ in node **7** not an induction variable (cf. [CK]).

[23] The algorithm of [JD2] deals with these problems using a machine dependent heuristics: assuming that the computation costs of two additions are less expensive than those of a multiplication, it would insert within the loop the assignments $\mathbf{h} := j * 3$ on the edge leaving node **9**, $\mathbf{h} := \mathbf{h} - 21$ at node **8**, and $\mathbf{h} := \mathbf{h} + 15$ at node **6**. This transformation would improve the "left" part of the loop construct.

Distributivity: The predicate `Distr` (*"Distributivity"*) establishes a relationship between candidates for strength reduction (given by terms of "$\nu_1\downarrow_1$" having "$*$" as top most operator) and abstract values (given by terms of "$\nu_2\downarrow_1$" and "$\nu_3\downarrow_1$"), whose sum is equivalent to the abstract value of the candidate:

$$\begin{array}{lcl} \mathtt{Distr}(\nu_1,\nu_2,\nu_3) & \Longleftrightarrow & \mathcal{N}(\nu_1)=\mathcal{N}(\nu_2)=\mathcal{N}(\nu_3)\ \wedge \\ & & \mathtt{Insert}(\nu_1)\wedge\mathtt{Avail}(\nu_2)\wedge\mathtt{Avail}(\nu_3)\ \wedge \\ & & L_{\mathbf{pre}(\mathcal{N}(\nu_1))}(\nu_1\downarrow_1)=\{*\}\ \wedge \\ & & \exists\, t_2\in\mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu_2))}(\nu_2\downarrow_1)\ \exists\, t_3\in\mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu_3))}(\nu_3\downarrow_1). \\ & & \quad (+,t_2,t_3)\in\Phi(\mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu_1))}(\nu_1\downarrow_1)) \end{array}$$

For notational convenience, we define the predicates `SR-Insert` and `SR-Usable` that are directly derived from `Distr`:

$$\mathtt{SR\text{-}Insert}(\nu)\quad\Longleftrightarrow\quad\exists\,\nu_1,\nu_2\in\mathit{VFN}.\ \mathtt{Distr}(\nu,\nu_1,\nu_2)$$

$$\mathtt{SR\text{-}Usable}(\nu)\quad\Longleftrightarrow\quad\exists\,\nu_1,\nu_2\in\mathit{VFN}.\ \mathtt{Distr}(\nu_1,\nu_2,\nu)$$

The intuitive meaning of the predicates `SR-Insert` (*"Strength Reduction Insertion"*) and `SR-Usable` (*"Usable for a Strength Reduction Insertion"*) is the following: whenever an abstract value ν satisfies the predicate `SR-Insert`, then there exist two further abstract values ν_1 and ν_2 satisfying the predicate `SR-Usable`, whose sum is equal to the abstract value of ν. This allows us to perform the strength reduction transformation, i.e. to choose a term for initialization, which has "$+$" as top most operator and initialization terms of $\nu_1\downarrow_1$ and $\nu_2\downarrow_1$ as operands, instead of the "standard" minimal representative of $\nu\downarrow_1$, which has "$*$" as top most operator. Note that due to the availability of ν_1 and ν_2 this choice is provably safe and it improves the run-time efficiency. This is a consequence of the fact that our transformation guarantees that abstract values satisfying the availability predicate, and being required by a strength reduction insertion, are hold in auxiliary variables at the insertion points (cf. Algorithm A.8). Obviously, there is no better choice of initialization terms:

Proposition 5.14 (Computation Forms)
The computation forms (initialization terms) are optimal with respect to the convertibility relation and the local equivalence information expressed by the value flow graph under consideration.

Every flow graph that is transformed by our two stage algorithm has essentially the same computations points as the flow graph that results from the algorithm of [SKR1] applied to the same value flow graph. Slight differences, which do not affect the efficiency of the resulting program, are only due to the fact that the algorithm of [SKR1] inserts computations at node exits rather than at node entries as it is done here. Initialization terms, however, differ significantly, because of the greater flexibility in the choice of the initialization terms here.

5.2 Placing the Computations

The placement Algorithm A.8 is a straightforward adaption of the placement algorithm of [SKR1]. It proceeds in three steps:

1. *Marking the definition-use chains in the value flow graph*:
   All nodes of the value flow graph that occur on paths leading from nodes satisfying `Insert` to nodes satisfying `Antloc` or `SR-Usable` are marked.

2. *Allocating auxiliary variables*:
   Every marked node of the value flow graph is associated with an auxiliary variable. This is a new auxiliary variable, if the marked node either satisfies `Insert` or has more than one predecessor in the value flow graph. Otherwise the auxiliary variable of its unique predecessor in the value flow graph is taken.

3. *Transforming the flow graph*:
   Auxiliary variables associated with a node ν of the value flow graph satisfying `Insert` are initialized by their initialization terms (cf. Section 5.1.2).

   If two marked nodes, which are associated with different auxiliary variables, say $\mathbf{h}_k$ and $\mathbf{h}_l$, are connected by an edge in the value flow graph, a trivial assignment $\mathbf{h}_l := \mathbf{h}_k$ is added at the end of the first node.

   Finally, original computations of the flow graph are replaced by the corresponding auxiliary variables.

5.3 Pragmatics

In this section we discuss some pragmatic aspects of the strength reduction transformation introduced so far.

Second order effects: As mentioned already the algorithm here has a greater flexibility in the choice of the initialization terms than the algorithm of [SKR1]. This is because of its potential of replacing multiplications by additions, which may lead to second order effects (cf. [RWZ]): replacing multiplications by summations according to the distributive law may introduce (partial) redundancies in the program. This is a consequence of the fact that the specific properties of "$+$" and "$*$" are only considered by the second stage of the algorithm, but not during the semantic analysis of the first stage. Introducing the predicate `SR-Redundant` (*"Redundant by the Strength Reduction Transformation"*) is sufficient to avoid the introduction of totally redundant computations (see Algorithm A.8 for details):

$$\mathtt{SR\text{-}Redundant}(\nu_1,\nu_2)\ \Longleftrightarrow\ \exists\,\nu_3,\nu_4\in\mathit{VFN}.\ \nu_3\downarrow_1=l(\nu_2\downarrow_1)\ \wedge\ \nu_4\downarrow_1=r(\nu_2\downarrow_1)\ \wedge$$
$$\mathtt{Distr}(\nu_1,\nu_3,\nu_4)\ \wedge\ \mathtt{Avail}(\nu_2)\ \wedge\ L_{\mathbf{pre}(\mathcal{N}(\nu_2))}(\nu_2\downarrow_1)\supseteq\{+\}$$

Intuitively, a pair of value flow graph nodes (ν_1,ν_2) satisfies the predicate `SR-Redundant`, if ν_1 is a candidate for strength reduction, whose abstract value has an additive representation by a term of $\nu_2\downarrow_1$ that is available at the insertion point. Therefore, keeping the value of this representation alive until the initialization point of ν_1, allows us to

Let now $M =_{df} \{ \nu_{r_{p_\nu}} \mid p_\nu \in P_\nu \}$, and $M' =_{df} \{ \nu_i \mid p_\nu \in P_\nu$ and $i \in \{1,..,r_{p_\nu} - 1\} \}$. Then investigating Equation System 5.3 and (**), we immediately obtain for all nodes μ of M: $\neg$`D-Safe`(μ). This, however, yields together with (**) and the consistency condition of Equation System 5.3 also: $\neg$`D-Safe`(μ) for all $\mu \in M'$, and therefore a contradiction, because of $\nu \in M \cup M'$. □

Earliestness: Earliestness of a placement is defined as follows:

Definition 5.5 (Earliestness)
The placement of an abstract value $\nu \in VFN$ is earliest *at the entry of node $\mathcal{N}(\nu)$, if there is a path $p = (n_1, .., n_q) \in \mathbf{P}[\mathbf{s}, \mathcal{N}(\nu)]$ and a path $p_\nu = (\nu_1, .., \nu_r) \in \mathbf{P}_{VFG}[\mu, \nu]$ corresponding to a postfix of p (i.e. $\mathcal{N}(\nu_i) = n_{i+(q-r)}$, $i \in \{1, .., r\}$), such that $\mu \in VFN_{\mathbf{s}}$, and for all $i \in \{1, .., r\}$ the abstract value ν_i is not down-safe at $\mathcal{N}(\nu_i)$.*

The set of earliest computation points for an abstract value is given by the least solution of Equation System 5.6, which specifies a forward analysis of VFG. As Equation System 5.3, it is a straightforward adaption of the corresponding equation systems of [KRS1], which are tailored for flow graphs, to value flow graphs here in order to capture semantic equivalence of terms.

Equation System 5.6 (Earliestness)

$$\mathbf{EARLIEST}(\nu) = \begin{cases} true & \text{if } \nu \in VFN_{\mathbf{s}} \\ \sum_{\nu' \in pred_{VFG}(\nu)} \neg\texttt{D-Safe}(\nu') \wedge \mathbf{EARLIEST}(\nu') & \text{otherwise} \end{cases}$$

Denoting the least solution of Equation System 5.6 by `Earliest`, we have:

Lemma 5.7 (Earliestness)
The placement of an abstract value $\nu \in VFN$ is earliest at the entry of node $\mathcal{N}(\nu)$ if and only if `Earliest`(ν) *holds.*

The proof of Lemma 5.7 is similar to the one of Lemma 5.4 and is therefore omitted. The computation points are now determined by

$$\texttt{Insert}(\nu) =_{df} \texttt{D-Safe}(\nu) \wedge \texttt{Earliest}(\nu)$$

Applying Lemma 5.4 and Lemma 5.7, we obtain the following theorem which guarantees that the program resulting from these computation points cannot be improved any further by means of semantics preserving code motion with respect to the value flow graph under consideration (cf. [SKR1]).

Theorem 5.8 (Computation Points)
The computation points satisfying `Insert`(ν) *are the earliest down-safe computation points for the abstract value ν.*

The proof of the following lemma is similar to that of Lemma 4.11.

Lemma 5.9 *Algorithm A.5 and Algorithm A.6 terminate with:*

$$\forall \nu \in VFN.\ \texttt{D-Safe}(\nu) = d\text{-}safe[\nu] \wedge \texttt{Earliest}(\nu) = earliest[\nu]$$

5.1.2 Choosing Computation Forms

In this step we determine for every value flow graph node ν satisfying the predicate `Insert` an initialization term (computation form), i.e. a term with "minimal" executions costs that represents the abstract value of the equivalence class $\nu\!\downarrow_1$. In the case of the Herbrand interpretation an initialization term is just a minimal representative of $\nu\!\downarrow_1$ (cf. [SKR1]). However, in order to capture the effects of strength reduction a more careful choice is necessary.

As mentioned already, the essence of classical strength reduction is the combination of ***availability*** and ***distributivity***. This is reflected by introducing the two predicates `Avail` and `Distr` that are concerned with the availability of abstract values and the distributivity of terms, respectively.

Availability: Intuitively, an abstract value ν is available at the entry of node $\mathcal{N}(\nu)$, if it is computed on every path from the start node of the program reaching node $\mathcal{N}(\nu)$. Usually, availability is determined with respect to the original computations of a program. Here, however, it is computed with respect to the computation points determined in the previous step (cf. Section 5.1.1). Thus, it means availability ***after*** the code motion transformation here, which, in terms of the original computations of the program, can be considered as ***potential availability***.

Definition 5.10 (Availability)
An abstract value $\nu \in VFN$ is available *at the entry of node $\mathcal{N}(\nu)$, if every path $p = (n_1, .., n_q) \in \mathbf{P}[\mathbf{s}, \mathcal{N}(\nu)]$ has a postfix with a corresponding path $p_\nu = (\nu_1, .., \nu_r) \in \mathbf{P}_{VFG}[\mu, \nu]$ (i.e. $\mathcal{N}(\nu_i) = n_{i+(q-r)}$, $i \in \{1, .., r\}$), such that* `Insert`(μ) *holds.*

The availability predicate `Avail` is defined as the greatest solution of Equation System 5.11, whose computation requires a forward analysis of VFG.

Equation System 5.11 (Availability)

$$\mathbf{AVAIL}(\nu) = \begin{cases} \texttt{Insert}(\nu) & \text{if } \nu \in VFN_{\mathbf{s}} \\ \texttt{Insert}(\nu) \vee \prod_{\nu' \in pred_{VFG}(\nu)} \mathbf{AVAIL}(\nu') & \text{otherwise} \end{cases}$$

We have:

Lemma 5.12 (Availability)
An abstract value ν is available at the entry of node $\mathcal{N}(\nu)$ if and only if `Avail`(ν) *holds.*

Whereas Lemma 5.12 can be proved in the same way as Lemma 5.4, the proof of the following lemma is similar to that of Lemma 4.11.

Lemma 5.13 *Algorithm A.7 terminates with:* $\forall \nu \in VFN.$ `Avail`$(\nu) = avail[\nu]$

5.1 Determination of Computation Points and Forms

The determination of computation points and forms is split into two steps. First, determining the computation points with respect to the equivalence information expressed by the value flow graph under consideration, and second, choosing the computation forms for the computation points computed in the first step. The first step is essentially the same as in [SKR1]. Only the bi-directional character of the equation system there is decomposed here into a backward and a forward analysis along the lines of [KRS1,KS,St3]. This decomposition does not affect the result of the analysis at all. The second step, however, has to be modified significantly here in comparison to [SKR1], in order to capture strength reduction. In fact, this step has been trivial in [SKR1], where computation forms are simply minimal representatives of the Herbrand equivalence classes associated with the computation points. However, in the context of strength reduction this choice is much more elaborate, because semantic equations need to be exploited in order to take care of replacing "expensive" by "cheap" operations (cf. Proposition 5.14).

5.1.1 Determining Computation Points

The point of this step is to determine for every abstract value ν of the value flow graph under consideration its *earliest down-safe* computation points. Intuitively, *down-safe* means that ν is used on every terminating path starting at the considered point. This is a necessary requirement for computation points in order to guarantee that the program semantics is preserved. *Earliest* means that a placement at an "earlier" position would not be down-safe, and therefore would change the program semantics by introducing the computation of a new abstract value on a path. Earliestness guarantees optimality of the placement.

Down-Safety: For every abstract value ν we define a local predicate `Antloc`[18] indicating whether ν is used in $\mathcal{N}(\nu)$ or not.

$$\texttt{Antloc}(\nu) \iff \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu))}(\nu\!\downarrow_1) \cap \mathcal{T}(\,\mathcal{N}(\nu)\,) \neq \emptyset$$

As mentioned already, down-safety of an abstract value at a node n means that it is used on every terminating program path starting at n.

Definition 5.2 (Down-Safety)
The placement of an abstract value $\nu \in VFN$[19] is down-safe *at the entry of node $\mathcal{N}(\nu)$, if every path $p=(n_1,..,n_q) \in \mathbf{P}[\mathcal{N}(\nu),\mathbf{e}]$ has a prefix with a corresponding path $p'=(\nu_1,..,\nu_r) \in \mathbf{P}_{VFG}[\nu,\mu]$ (i.e. $\mathcal{N}(\nu_i)=n_i,\ i \in \{1,..,r\}$), such that `Antloc`$(\mu)$ holds.*

The set of down-safe computation points for the abstract value ν is given by the greatest solution of Equation System 5.3, which specifies a backward analysis of VFG.

[18] The name of this predicate stands for *local anticipability* (cf. [MR]).
[19] Placement here means to insert a computation of a term t represented by $\nu\!\downarrow_1$ at the entry of $\mathcal{N}(\nu)$.

Equation System 5.3 (Down-Safety)

$$\mathbf{D\text{-}SAFE}(\nu) = \begin{cases} \mathit{false} & \text{if } \mathcal{N}(\nu)=\mathbf{e} \\ \texttt{Antloc}(\nu) \ \vee \prod\limits_{m \in succ_G(\mathcal{N}(\nu))} \ \sum\limits_{\substack{\nu' \in succ_{VFG}(\nu) \\ \mathcal{N}(\nu')=m}} \mathbf{D\text{-}SAFE}(\nu') & \text{otherwise} \end{cases}$$

Denoting the greatest solution of Equation System 5.3 by `D-Safe`, we have:

Lemma 5.4 (Down-Safety)
The placement of an abstract value ν is down-safe at the entry of node $\mathcal{N}(\nu)$ if and only if `D-Safe`(ν) holds.

Proof. We first prove the implication from the left to the right. Thus, let ν be an abstract value, which can down-safely be placed at the entry of $\mathcal{N}(\nu)$. Then, according to Definition 5.2, every path $p=(n_1,...,n_q) \in \mathbf{P}[\mathcal{N}(\nu),\mathbf{e}]$ possess a, w.l.o.g. shortest, prefix with a corresponding path $p_\nu=(\nu_1,...,\nu_{r_p}) \in \mathbf{P}_{VFG}[\nu,\mu]$ satisfying `Antloc`(μ) and $\mathcal{N}(\nu_i)=n_i$ for $i \in \{1,..,r_p\}$. Based on this definition of r_p with respect to a path $p \in \mathbf{P}[\mathcal{N}(\nu),\mathbf{e}]$, we define the set of value flow graph nodes $M =_{df} M_A \cup M_T \subseteq VFN$, where

$$M_A =_{df} \{\, \nu_{r_p} \mid p \in \mathbf{P}[\mathcal{N}(\nu),\mathbf{e}]\,\}$$

and

$$M_T =_{df} \{\, \nu_i \mid p \in \mathbf{P}[\mathcal{N}(\nu),\mathbf{e}] \text{ and } i \in \{1,..,r_p-1\}\,\}$$

Obviously, we have:

1. $\nu \in M$
2. $\forall\, \mu \in M_T.\ succ_{VFG}(\mu) \subseteq M_T \cup M_A$
3. $\forall\, \mu \in M_A.$ `Antloc`(μ)

`D-Safe` is defined as the greatest solution of Equation System 5.3. Thus, according to (1), it is sufficient to check that $\mathbf{D\text{-}SAFE}(\mu)=\mathit{true},\ \mu \in M$, is consistent with this equation system in order to verify the first implication. For nodes $\mu \in M_A$, this consistency is an immediate consequence of (3), and for nodes $\mu \in M_T$, it easily follows from (2) and the consistency of $\mathbf{D\text{-}SAFE}(\mu)=\mathit{true}$ for the nodes of M_A.

For the proof of the remaining implication, let $\nu \in VFN$ satisfy the predicate `D-Safe`(ν), and assume that there is a path $p=(n_1,..,n_q) \in \mathbf{P}[n,\mathbf{e}]$ violating the lemma. Moreover, let P_ν be the set of maximal corresponding paths to prefices of p. Then every path $p_\nu=(\nu_1,..,\nu_{r_{p_\nu}}) \in P_\nu$ satisfies $\mathcal{N}(\nu_i)=n_i,\ i \in \{1,..,r_{p_\nu}\}$, and $\mathcal{N}(\nu_{r_{p_\nu}})=\mathbf{e}$ or $n_{r_{p_\nu}+1} \notin \mathcal{N}(succ_{VFG}(\nu_{r_{p_\nu}}))$, and therefore due to the choice of p and P_ν:

(∗∗) $\forall\, p_\nu \in P_\nu\ \forall\, i \in \{1,..,r_{p_\nu}\}.\ \neg$`Antloc`$(\nu_i)$

Thus, nodes ν of the value flow graph are pairs (γ_1, γ_2), where γ_1 is a node of the pre-DAG and γ_2 a node of the post-DAG for node n of G, such that γ_1 and γ_2 represent the same values, i.e. satisfy the inclusion $\mathbf{T}_{\mathbf{pre}(n)}(\gamma_1) \supseteq \{t \mid \exists t' \in \mathbf{T}_{\mathbf{post}(n)}(\gamma_2)\ t = \delta_n(t')\}$. In fact, we have the following lemma, which can easily be proved by means of the Substitution Lemma 3.1 and the Optimality Theorem 4.9:

Lemma 4.14 (*VFN*-**Lemma**)

$$\forall \nu \in VFN\ \forall p \in \mathbf{P}[\mathbf{s}, \mathcal{N}(\nu))\ \forall t_1 \in \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu))}(\nu\!\downarrow_1)\ \forall t_2 \in \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu))}(\nu\!\downarrow_2).$$

$$\mathbf{H}(t_1)(\Theta_p(\sigma_0)) = \mathbf{H}(t_2)(\Theta_{p;(n)}(\sigma_0))$$

This lemma motivates our decision to refer to value flow graph nodes as ***abstract values***, because on every path reaching node $\mathcal{N}(\nu)$ all terms represented by the first and second projection of ν have the same value with respect to the Herbrand interpretation.

Edges of the value flow graph are pairs (ν, ν'), such that $\mathcal{N}(\nu')$ is a successor of $\mathcal{N}(\nu)$ and values are maintained along the connecting edge $(\mathcal{N}(\nu), \mathcal{N}(\nu'))$, i.e. $\mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu))}(\nu\!\downarrow_2) \supseteq \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu'))}(\nu'\!\downarrow_1)$. Here we have:

Lemma 4.15 (*VFE*-**Lemma**)

$$\forall (\nu, \nu') \in VFE\ \forall p \in \mathbf{P}[\mathbf{s}, \mathcal{N}(\nu)]\ \forall t_2 \in \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu))}(\nu\!\downarrow_2)\ \forall t_1 \in \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu'))}(\nu'\!\downarrow_1).$$

$$\mathbf{H}(t_2)(\Theta_p(\sigma_0)) = \mathbf{H}(t_1)(\Theta_p(\sigma_0))$$

Thus, nodes of the value flow graph model the value flow over a single assignment statement, and edges the value flow along the (local) branching structure of G. See Figure 2.4 for illustration[15].

As an inductive extension of the *VFE*-Lemma 4.15 we obtain the *VFG*-Lemma 4.16, which expresses the central property of a value flow graph, namely that it models the value flow over the global branching structure of G. In order to define this property precisely, let $\mathbf{P}_{VFG}[\nu, \nu']$ and $\mathbf{P}_{VFG}[\nu, \nu')$ denote the set of all finite paths from ν to ν', and from ν to a predecessor of ν' in *VFG*, respectively. Moreover, let $p_{VFG} = (\nu_1, ..., \nu_q) \in \mathbf{P}_{VFG}[\nu, \nu']$, and $p_G = (n_1, ..., n_r) \in \mathbf{P}[n, n']$. Then the paths p_{VFG} and p_G are said to ***correspond*** to each other if and only if $q = r$ and $\mathcal{N}(\nu_i) = n_i,\ i \in \{1, .., q\}$[16]. Denoting the corresponding path of $p_{VFG} \in \mathbf{P}_{VFG}[\nu, \nu']$ in $\mathbf{P}[\mathcal{N}(\nu), \mathcal{N}(\nu')]$ by $\mathcal{C}[p_{VFG}]$, and its unique prefix in $\mathbf{P}[\mathcal{N}(\nu), \mathcal{N}(\nu'))$ by $\mathcal{C}[p_{VFG})$, we obtain:

Lemma 4.16 (*VFG*-**Lemma**)

$$\forall p_{VFG} \in \mathbf{P}_{VFG}[\nu_1, \nu_2]\ \forall p' \in \mathbf{P}[\mathbf{s}, \mathcal{N}(\nu_1))\ \forall t_1 \in \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu_1))}(\nu_1\!\downarrow_1)\ \forall t_2 \in \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu_2))}(\nu_2\!\downarrow_1).$$

$$\mathbf{H}(t_1)(\Theta_{p'}(\sigma_0)) = \mathbf{H}(t_2)(\Theta_{p';\mathcal{C}[p_{VFG})}(\sigma_0))$$

[15] A more detailed explanation can be found in the first part of [SKR2].

[16] Note that every path in $\mathbf{P}_{VFG}$ has a corresponding path in G, whereas the converse is in general false.

Intuitively this yields that whenever a path of the value flow graph p_{VFG} corresponds to a part of a program execution (path in the flow graph), then the abstract values of p_{VFG} that belong to the different steps of the program execution represent the same value.

We conclude this section by introducing two abbreviations:

$$VFN_{\mathbf{s}} =_{df} \{\, \nu \mid \mathcal{N}(\, pred_{VFG}(\nu)\,) \neq pred_G(\, \mathcal{N}(\nu)\,)\ \vee\ \mathcal{N}(\nu) = \mathbf{s}\,\}$$

and

$$VFN_{\mathbf{e}} =_{df} \{\, \nu \mid \mathcal{N}(\, succ_{VFG}(\nu)\,) \neq succ_G(\, \mathcal{N}(\nu)\,)\ \vee\ \mathcal{N}(\nu) = \mathbf{e}\,\}$$

Here, $pred_{VFG}$ and $succ_{VFG}$ denote functions that map a node of *VFG* to its set of predecessors and successors, respectively.

5 Placement of Computations

Before going into details, let us mention a technicality, which is typical for code motion (cf. [KRS1,KS,RWZ,SKR1,St3]). Edges, leading from a node with more than one successor to a node with more than one predecessor are ***critical***, since they may "block" the code motion process:

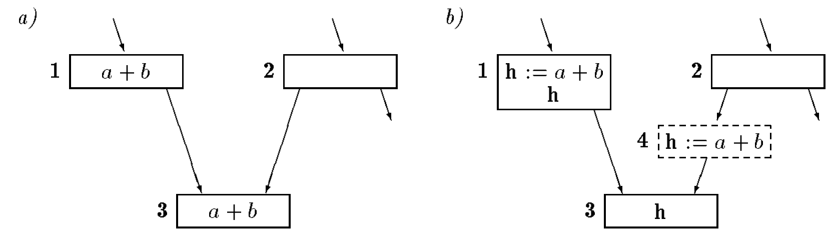

Figure 5.1

In Figure 5.1(a) the computation of "$a+b$" at node **3** is partially redundant with respect to the computation of "$a+b$" at node **1**. However, this partial redundancy cannot safely be eliminated by moving the computation of "$a+b$" to its preceding nodes, because this may introduce a new computation on a path leaving node **2** on the right branch. On the other hand, it can safely be eliminated after inserting a synthetic node **4** in the critical edge $(\mathbf{2}, \mathbf{3})$, as it is illustrated in Figure 5.1(b).

In order to exclude any blocking, we will therefore assume that every original edge leading to a node with more than one predecessor has been split by inserting a synthetic node. This simple transformation certainly implies that all critical edges are eliminated. Moreover, it simplifies the subsequent analysis, since it allows us to meet our optimality result by inserting all computations uniformly at the entry of nodes[17].

[17] Splitting critical edges would require a placement procedure which is capable of placing computations both at node entries and exits. On the other hand, if all edges are split leaving a node with more than one successor, the same optimality result can be obtained by placing all computations uniformly at the exit of nodes as it was done in [SKR1]. Thus, after a simple syntactic transformation all placing strategies have the same transformational power; and here we follow the placing strategy of [KRS1].

with

$$\begin{array}{rcl}(pre^k[n'], post^k[n']) & = & (pre^{k-1}[n'] \sqcap post^{k-1}[m], [\![n]\!](pre^{k-1}[n'] \sqcap post^{k-1}[m])) \\ & \sqsubset & (pre^{k-1}[n'], post^{k-1}[n'])\end{array}$$

are added to the workset.

Collecting the results of the invariants (2) and (3), and the fact that the workset is empty after the (guaranteed) termination, we obtain that $pre[n]$ contains a solution of Equation System 4.7 after the termination of Algorithm 4.10. Finally, invariant (1) yields the required maximality of this solution, which completes the proof of the lemma. □

Theorem 4.9 and Lemma 4.11 yield the central result of this section:

Theorem 4.12 (Optimality Theorem)
Algorithm 4.10 terminates with an annotation of the flow graph with finite structured partition DAGs, which semantically characterizes all equivalences of program terms with respect to the Herbrand interpretation.

4.2 Computation of Relevant Terms

The placement process of our algorithm (Section 5.1) considers the pre-DAGs and post-DAGs of a flow graph annotation as purely syntactical objects, i.e. terms are considered equivalent iff they are syntactically equivalent (Definition 4.3(2)). Thus, we need to extend the flow graph annotation constructed in Section 4.1, which characterizes Herbrand equivalence semantically (Definition 4.3(3)), to a sufficiently large syntactic representation. As in [SKR1], this is achieved by computing for every node n of G a finite set of *relevant* terms $T_{suf}(n)$ that contains a representative of all equivalence classes that are necessary at node n. However, in order to capture (classical) strength reduction, we additionally need to exploit algebraic laws. Remember, classical strength reduction essentially replaces computations of the form $u*(v+w)$ by $(u*v)+(u*w)$. This is safe and profitable, whenever the values of $u*v$ and $u*w$ are available. Therefore, we consider a term $(u*v)+(u*w)$ and its subterms as relevant here, whenever the term $u*(v+w)$ is relevant in the sense of [SKR1]. Moreover, commutativity and associativity are necessary in order to evaluate subterms with constant operands, whose values can be computed already at compile time. This requires to enlarge the number of relevant expressions accordingly.

Technically, this is realized by enhancing the strategies of [SKR1] for computing relevant terms by means of the closure operator $\Phi : \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$, which is defined by:

$$\forall\, T \subseteq \mathbf{T} : \Phi(T) =_{df} \{\, t' \mid \exists\, t \in T.\ t \equiv_c t' \,\}$$

where $\equiv_c \subseteq \mathbf{T} \times \mathbf{T}$ denotes a convertibility relation between terms: $t_1 \equiv_c t_2$ if and only if t_1 and t_2 can be deduced from each other by means of the commutative, associative and distributive law for "$+$" and "$*$", together with the evaluation of subterms with constant operands.

In this paper, we consider the basic strategy of [SKR1] for computing relevant terms, which determines for every program point the set of all terms whose value *must* be computed on every continuation of a program execution passing this point. Enhancing this strategy by means of the closure operator Φ is already sufficient to uniformly capture the known strength reduction algorithms[14]. The complete closure algorithm is given in the Appendix (see Algorithm A.3).

4.3 The Value Flow Graph

The value flow graph represents global equivalence information explicitly. Essentially, its nodes represent term equivalence classes and its edges the data flow. For technical reasons, the nodes of a value flow graph are defined as pairs of equivalence classes. However, identifying these pairs with their first components leads back to the original intuition.

In the following let us assume that every node n of G is annotated by a pre-DAG $\mathbf{pre}(n)$ and a post-DAG $\mathbf{post}(n)$ according to the results of Section 4.2. For the sake of readability we abbreviate $\bigcup_{n \in N} (N_{\mathbf{pre}(n)} \times N_{\mathbf{post}(n)})$ by Γ, and denote the flow graph node corresponding to a pair $(\gamma, \gamma') \in \Gamma$ by $\mathcal{N}(\gamma, \gamma')$. This allows us to define the backward substitution relation $\stackrel{\delta}{\longleftarrow} \subseteq \Gamma$ by:

$$\forall\, (\gamma, \gamma') \in \Gamma.\ \gamma \stackrel{\delta}{\longleftarrow} \gamma' \Longleftrightarrow_{df} \quad \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\gamma,\gamma'))}(\gamma) \supseteq \delta_{\mathcal{N}(\gamma,\gamma')}(\mathbf{T}_{\mathbf{post}(\mathcal{N}(\gamma,\gamma'))}(\gamma'))$$

where $(\gamma, \gamma') \in \stackrel{\delta}{\longleftarrow}$ is abbreviated by $\gamma \stackrel{\delta}{\longleftarrow} \gamma'$. Intuitively, $\gamma \stackrel{\delta}{\longleftarrow} \gamma'$ means that the set of terms represented by γ', backward substituted with respect to the assignment of node $\mathcal{N}(\gamma, \gamma')$, is included in the set of terms represented by γ.

Let now $\odot$ denote a new symbol. Then the formal definition of the value flow graph for the DAG annotation under consideration is as follows (cf. [St1,SKR1]):

Definition 4.13 (Value Flow Graph)
A value flow graph VFG is a pair (VFN, VFE) consisting of

- *a set of nodes $VFN \subseteq \dot{\bigcup}_{n \in N} ((N_{\mathbf{pre}(n)} \cup \{\odot\}) \times (N_{\mathbf{post}(n)} \cup \{\odot\}))$, where*

$$\nu = (\gamma_1, \gamma_2) \in VFN \Longleftrightarrow_{df} \begin{cases} \gamma_1 \stackrel{\delta}{\longleftarrow} \gamma_2 & \text{if } \gamma_1 \neq \odot \wedge \gamma_2 \neq \odot \\ \nexists\, \gamma_3 . \gamma_1 \stackrel{\delta}{\longleftarrow} \gamma_3 & \text{if } \gamma_1 \neq \odot \wedge \gamma_2 = \odot \\ \nexists\, \gamma_3 . \gamma_3 \stackrel{\delta}{\longleftarrow} \gamma_2 & \text{if } \gamma_1 = \odot \wedge \gamma_2 \neq \odot \end{cases}$$

- *a set of edges $VFE \subseteq VFN \times VFN$, where*

$$(\nu, \nu') \in VFE \Longleftrightarrow_{df} \begin{cases} \nu\!\downarrow_2 \neq \odot \wedge \nu'\!\downarrow_1 \neq \odot \wedge \\ \mathcal{N}(\nu') \in succ_G(\mathcal{N}(\nu)) \wedge \\ \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu))}(\nu\!\downarrow_2) \supseteq \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu'))}(\nu'\!\downarrow_1) \end{cases}$$

where "$\downarrow_1$" and "$\downarrow_2$" denote the projection of a node ν to its first and second component, respectively, and $\mathcal{N}(\nu)$ the node of the flow graph that is related to ν.

[14] Of course, the same is true for the other, more complex strategies.

```
FORALL m ∈ N DO
  IF m = s
    THEN (pre[m], post[m]) := (⊥, ⟦ m ⟧(⊥))
    ELSE (pre[m], post[m]) := (⊤, ⊤)
  FI
OD;
workset := {s};

( Iterative fixed point computation )
WHILE workset ≠ ∅ DO
  LET m ∈ workset
    BEGIN
      workset := workset\{m};
      ( Update the "environment" of node m )
      FORALL n ∈ succ_G(m) DO
        meet := pre[n] ⊓ post[m];
        IF pre[n] ⊐ meet
          THEN
            pre[n] := meet;
            post[n] := ⟦ n ⟧(pre[n]);
            workset := workset ∪ {n}
        FI
      OD
    END
OD.
```

In order to simplify the formulation of the central properties of Algorithm 4.10, we abbreviate the values of *workset*, $pre[n]$ and $post[n]$ after the k-th execution of the while-loop by $workset^k$, $pre^k[n]$ and $post^k[n]$, respectively. We have:

Lemma 4.11 *Algorithm 4.10 terminates with:* $\forall\, n \in N.\ \mathbf{pre}_{\perp}(n) = pre[n]$.

Proof. The termination of Algorithm 4.10 is an immediate consequence of Lemma 4.6 and the fact that $\mathcal{PD}$ satisfies the descending chain condition.

For the proof of the remaining part of the lemma, let *fix-pre* and *fix-post* denote an arbitrary solution of Equation System 4.7. This implies in particular:

$$(*) \quad (\textit{fix-pre}[\mathbf{s}], \textit{fix-post}[\mathbf{s}]) = (\perp, [\![\, n \,]\!](\perp))$$

The central step in proving Lemma 4.11 is to check the following three invariants of Algorithm 4.10.

1. $\forall\, k \in \omega\ \forall\, n \in N.\ (\textit{fix-pre}[n], \textit{fix-post}[n]) \sqsubseteq (pre^k[n], post^k[n])$
2. $\forall\, k \in \omega\ \forall\, n \in N.\ pre^k[n] \sqcap \bigsqcap\{\, post^k[m] \mid m \in pred_G(n) \cap workset^k \,\} =$
$$\begin{cases} \perp & \text{if } n = \mathbf{s} \\ \bigsqcap\{\, post^k[m] \mid m \in pred_G(n) \,\} & \text{otherwise} \end{cases}$$
3. $\forall\, k \in \omega\ \forall\, n \in N.\ post^k[n] = [\![\, n \,]\!](pre^k[n])$

These three invariants are proved simultaneously by induction on the number k of iterations of the while-loop. For $k = 0$, the investigation of the initialization part of Algorithm 4.10 delivers

$$\begin{array}{lll} (a) & pre^0[n] = \begin{cases} \perp & \text{if } n = \mathbf{s} \\ \top & \text{otherwise} \end{cases} \\ (b) & post^0[n] = \begin{cases} [\![\, n \,]\!](\perp) & \text{if } n = \mathbf{s} \\ \top & \text{otherwise} \end{cases} \\ (c) & workset^0 = \{\, \mathbf{s} \,\} \end{array}$$

Thus, (1) is an immediate consequence of (a) (b), and $(*)$. Moreover, (2) holds trivially for all nodes $n \notin succ_G(\mathbf{s})$, and for $n \in succ_G(\mathbf{s})$ it is a consequence of the following sequence of equalities[12]:

$$\begin{array}{lcl} & & pre^0[n] \sqcap \bigsqcap\{\, post^0[m] \mid m \in pred_G(n) \cap workset^0 \,\} \\ (\,(a), (c), \text{ and } n \neq \mathbf{s}\,) & = & \top \sqcap post^0[\mathbf{s}] \\ (\,(b)\,) & = & \bigsqcap\{\, post^0[m] \mid m \in pred_G(n) \backslash \{\mathbf{s}\} \,\} \sqcap post^0[\mathbf{s}] \\ & = & \bigsqcap\{\, post^0[m] \mid m \in pred_G(n) \,\} \end{array}$$

Finally, (3) follows from (b) and the $\top$-assumption[13].

For the induction step, let $k>0$ and m be the element currently chosen from the workset. Obviously, we have

$$(**) \quad pre^k[n] = \begin{cases} pre^{k-1}[n] \sqcap post^{k-1}[m] & \text{if } n \in succ_G(m) \ \wedge \\ & \quad pre^{k-1}[n] \sqsupset pre^{k-1}[n] \sqcap post^{k-1}[m] \\ pre^{k-1}[n] & \text{otherwise} \end{cases}$$

$$(***) \quad post^k[n] = \begin{cases} [\![\, n \,]\!](pre^k[n]) & \text{if } n \in succ_G(m) \ \wedge \\ & \quad pre^{k-1}[n] \sqsupset pre^{k-1}[n] \sqcap post^{k-1}[m] \\ post^{k-1}[n] & \text{otherwise} \end{cases}$$

Together with Lemma 4.6 and the induction hypothesis, this implies the induction step for the invariants (1) and (3).

For the induction step of (2), we can assume: $n \in succ_G(m) \cup succ_G(succ_G(m))$, because otherwise (2) follows trivially by induction. In the case of $n \in succ_G(m)$, the induction step is a consequence of the induction hypothesis, $(**)$ on $(***)$.

Thus, we are left with the case $n \in succ_G(succ_G(m)) \backslash succ_G(m)$. If no predecessor of n is added to the workset during the k^{th} iteration of the while-loop, i.e. if all nodes $n' \in pred_G(n)$ satisfy

$$(pre^k[n'], post^k[n']) \quad = \quad (pre^{k-1}[n'], post^{k-1}[n'])$$

the induction step holds trivially by induction. Otherwise, the induction step follows from the induction hypothesis, $(**)$, $(***)$, and the fact that all nodes $n' \in pred_G(n)$

[12] Note that $n \in succ_G(\mathbf{s})$ implies $n \neq \mathbf{s}$.
[13] See Algorithm 4.10 (**INPUT**).

Structured partition DAGs are tailored for computing Herbrand equivalences. In fact, an annotation of the flow graph with pre and post DAGs that characterize all Herbrand equivalences at a given program point semantically results from a fixed point globalization of the following local abstract semantics (cf. Optimality Theorem 4.12):

Definition 4.4 (Local Abstract Semantics)
The polymorphic functional

$$[\![\ ]\!]: \left.\begin{matrix}\mathcal{P}(\mathbf{T})\\ N\end{matrix}\right\} \rightarrow (\mathcal{PD} \rightarrow \mathcal{PD})$$

specifies a local abstract semantics that is defined by

- $\forall\, T \in \mathcal{P}(\mathbf{T})\ \forall\, D \in \mathcal{PD}.\ [\![\, T \,]\!](D) =_{df} \bigsqcap \{\, D' \in \mathcal{PD} \mid D \sqsubseteq D' \ \wedge\ \mathbf{T}(D') \supseteq \mathbf{T}(D) \cup T \,\}$
- $\forall\, n \in N\ \forall\, D \in \mathcal{PD}.\ [\![\, n \,]\!](D) =_{df} \{\, (t, t') \mid (\delta_n(t), \delta_n(t')) \in [\![\, \mathcal{T}(n) \,]\!](D) \,\}$

Intuitively, $[\![\, T \,]\!](D)$ extends the carrier set of D onto T, and $[\![\, n \,]\!](D)$ computes the effect of the assignment represented by n after extending the carrier set of D to the set of terms occurring in n. This is illustrated in the example of Figure 4.5. In particular, this example shows that structured partition DAGs allow an efficient realization of $[\![\ ]\!]$. In fact, the effects of assignments can be computed essentially by updating the position of the left hand side variables:

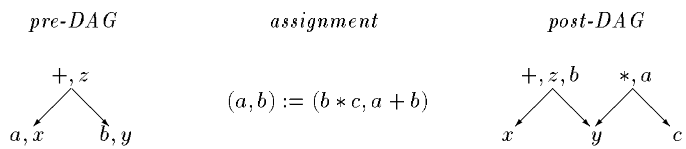

Figure 4.5

The following lemma is easy to prove:

Lemma 4.6 (Distributivity)
The semantic functions $[\![\, T \,]\!]$, $T \subseteq \mathbf{T}$, *and* $[\![\, n \,]\!]$, $n \in N$, *are distributive*[9].

We are now going to present the fixed point globalization of $[\![\ ]\!]$ for G. Consider Equation System 4.7:

Equation System 4.7

$$\mathbf{pre}(n) \;=\; \begin{cases} \bot & \text{if } n = \mathbf{s} \\ \bigsqcap \{\, \mathbf{post}(m) \mid m \in pred_G(n) \,\} & \text{otherwise} \end{cases}$$

$$\mathbf{post}(n) \;=\; [\![\, n \,]\!](\mathbf{pre}(n))$$

[9]A function $f \ : \ C \rightarrow D$ for some complete semi-lattices C and D is called *distributive* iff $\forall C' \subseteq C.\ f(\bigsqcap C') = \bigsqcap \{ f(c) \mid c \in C' \}$.

Let $\mathbf{pre}_\bot$ denote the greatest solution of Equation System 4.7 with respect to the node entries[10], and $[\![\, p \,]\!]$, $p = (n_1, .., n_q) \in \mathbf{P}[m, n]$, the straightforward extension of $[\![\ ]\!]$ to finite paths, i.e.:

$$[\![\, p \,]\!] =_{df} \begin{cases} id_{\mathcal{PD}} & \text{if } p \equiv \varepsilon \\ [\![\, (n_2, .., n_q) \,]\!] \circ [\![\, n_1 \,]\!] & \text{otherwise} \end{cases}$$

where $id_{\mathcal{PD}}$ denotes the identity on $\mathcal{PD}$. Then the application of Lemma 4.6 and the Coincidence Theorem of [Ki2,KU] directly delivers[11]:

Lemma 4.8 $\forall\, n \in N.\ \mathbf{pre}_\bot(n) = \bigsqcap \{\, [\![\, p \,]\!](\bot) \mid p \in \mathbf{P}[\mathbf{s}, n) \,\}$

Lemma 4.8 allows us to prove Theorem 4.9 by straightforward induction.

Theorem 4.9 $\forall\, t, t' \in \mathbf{T}\ \forall\, n \in N.$ *t and t' are Herbrand-equivalent at n iff t and t' are semantically* $\mathbf{pre}_\bot(n)$*-equivalent.*

The practical impact of Theorem 4.9 is consequence of $\mathcal{PD}$ satisfying the descending chain condition, which yields the computability of the Herbrand equivalence (cf. Section 4.1.1).

4.1.1 The Algorithm

In this section we present an algorithm, which computes an annotation of G with structured partition DAGs that semantically characterizes all Herbrand equivalences.

Algorithm 4.10 (The Semantic Analysis of Step 1(i))

Input: *A flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$ *with unique start node* $\mathbf{s}$ *and unique stop node* $\mathbf{e}$, *which are assumed to possess no predecessors and no successors, respectively. Additionally, the local abstract semantics* $[\![\, n \,]\!]$, $n \in N$, *according to Definition 4.4, where all semantic functions* $[\![\, n \,]\!]$ *are assumed to map* $\top$ *to* $\top$ *($\top$-assumption).*

Output: *An annotation of G with pre-DAGs (stored in pre) and post-DAGs (stored in post) of* $\mathcal{PD}_{fin}$ *characterizing valid and complete equivalence information at the entry and at the exit of every node* $n \in N$, *respectively.*

Remark: $\bot$ *denotes the "empty" data flow information and* $\top$ *its complement, the "universal" data flow information, which is assumed to "contain" every data flow information. The local analysis component* $[\![\ ]\!]$ *(see Algorithm A.1) and the* meet *operation* $\sqcap$ *(see Algorithm A.2) are assumed to work on structured partition DAGs. The variable workset controls the iterative process, and the auxiliary variable meet stores the result of the most recent meet operation.*

(Initialization of the annotation arrays pre and post and the variable workset)

[10]Note that $\mathbf{pre}_\bot$ is the *maximal fixed point* solution in the sense of [Ki2,KU].
[11]Note that $\bigsqcap \{\, [\![\, p \,]\!](\bot) \mid p \in \mathbf{P}[\mathbf{s}, n) \,\}$ is the *meet over all paths* solution in the sense of [Ki2,KU].

which completes the proof of the Substitution Lemma 3.1. □

A *finite path* of G is a sequence $(n_1,..,n_q)$ of nodes such that $(n_j,n_{j+1}) \in E$ for $j \in \{1,..,q-1\}$. $\mathbf{P}[m,n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}[m,n)$ the set of all finite paths from m to a predecessor of n. Additionally, ε denotes the unique path of length 0, and ";" the concatenation of two paths. Now the backward substitution functions $\delta_n : \mathbf{T} \to \mathbf{T}$ and the state transformations $\theta_n : \Sigma \to \Sigma$ can be extended to cover finite paths as well. For every path $p = (m \equiv n_1, \ldots, n_q \equiv n) \in \mathbf{P}[m,n]$, we define $\Delta_p : \mathbf{T} \to \mathbf{T}$ by

$$\Delta_p =_{df} \begin{cases} id_{\mathbf{T}} & \text{if } p \equiv \varepsilon \\ \Delta_{(n_1,..,n_{q-1})} \circ \delta_{n_q} & \text{otherwise} \end{cases}$$

and $\Theta_p : \Sigma \to \Sigma$ by

$$\Theta_p =_{df} \begin{cases} id_{\Sigma} & \text{if } p \equiv \varepsilon \\ \Theta_{(n_2,..,n_q)} \circ \theta_{n_1} & \text{otherwise} \end{cases}$$

where $id_{\mathbf{T}}$ and id_{Σ} denote the identity on $\mathbf{T}$ and Σ, respectively.

The set of all *possible states* at a node $n \in N$, Σ_n, is given by

$$\Sigma_n =_{df} \{ \sigma \in \Sigma \mid \exists p \in \mathbf{P}[\mathbf{s}, n) : \Theta_p(\sigma_0) = \sigma \}$$

Now we can define:

Definition 3.2 (Herbrand Equivalence)
Let $t_1, t_2 \in \mathbf{T}$ and $n \in N$. Then t_1 and t_2 are Herbrand equivalent *at node n iff*

$$\forall \sigma \in \Sigma_n.\ \mathbf{H}(t_1)(\sigma) = \mathbf{H}(t_2)(\sigma)$$

The development in the remainder of this paper is made with respect to a fixed flow graph G, which we consider as to be given from now on.

4 Construction of a Value Flow Graph

4.1 Local Semantic Equivalence

The semantic analysis of step 1(i) coincides with that of [SKR1]. Therefore, we only give a brief outline of the central notions and results that are relevant here. Equivalences are expressed by means of structured partition DAGs (cf. [FKU]), which are directed, acyclic multigraphs, whose nodes are labeled with at most one operator or constant and a set of variables. Given a structured partition DAG, two terms are equivalent iff they are represented by the same node of the DAG. – To define the notion of a structured partition DAG precisely, let $\mathcal{P}_{fin} =_{df} \{ T \mid T \subseteq (\mathbf{V} \cup \mathbf{C} \cup \mathbf{Op}) \wedge |T| \in \omega \backslash \{0\} \}$.

Definition 4.1 (Structured Partition DAG)
A structured partition DAG *is a triple $D = (N_D, E_D, L_D)$, where*

- *(N_D, E_D) is a directed acyclic graph with node set N_D and edge set $E_D \subseteq N_D \times N_D$*
- *$L_D : N_D \to \mathcal{P}_{fin}$ is a labelling function, which satisfies*
  1. *$\forall \gamma \in N_D.\ |L_D(\gamma) \backslash \mathbf{V}| \leq 1$ and*
  2. *$\forall \gamma, \gamma' \in N_D.\ \gamma \neq \gamma' \Rightarrow L_D(\gamma) \cap L_D(\gamma') \subseteq \mathbf{Op}$*
- *Leaves of D are the nodes $\gamma \in N_D$ with $L_D(\gamma) \cap \mathbf{Op} = \emptyset$*
- *An inner node γ of D possesses exactly two successors, which we denote by $l(\gamma)$ and $r(\gamma)$*
- *$\forall \gamma, \gamma' \in N_D.\ L_D(\gamma) \cap L_D(\gamma') \cap \mathbf{Op} \neq \emptyset \ \wedge \ l(\gamma) = l(\gamma') \ \wedge \ r(\gamma) = r(\gamma') \Rightarrow \gamma = \gamma'$*

Additionally, D is called a finite structured partition DAG, *if N_D is finite. The set of all structured partition DAGs (finite structured partition DAGs) is denoted by $\mathcal{PD}$ ($\mathcal{PD}_{fin}$).*

A node $\gamma \in N_D$ of a structured partition DAG is meant to represent an equivalence class of program terms:

$$\mathbf{T}_D(\gamma) = ((\mathbf{V} \cup \mathbf{C}) \cap L_D(\gamma)) \cup \{(op, t, t') \mid op \in (\mathbf{Op} \cap L_D(\gamma)) \wedge (t, t') \in \mathbf{T}_D(l(\gamma)) \times \mathbf{T}_D(r(\gamma))\}$$

Thus a full DAG represents a partition (or equivalence relation) on its *carrier set* $\mathbf{T}(D)$:

$$\mathbf{T}(D) =_{df} \bigcup \{\mathbf{T}_D(\gamma) \mid \gamma \in N_D\} \subseteq \mathbf{T}$$

This can be illustrated as follows:

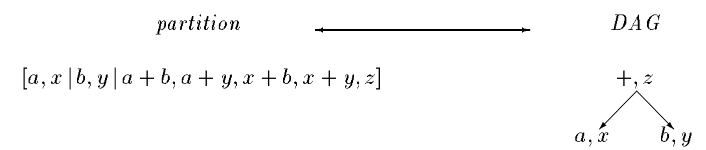

Figure 4.2

Viewing DAGs as equivalence relations makes $\mathcal{PD}$ a complete lattice, with bottom element $\perp = \emptyset$, top element $\top = \{\mathbf{T}\}$, and inclusion defined set theoretically as usual[8]. This guarantees the existence and well definedness of $\mathcal{H}(D)$ in:

Definition 4.3 *Let $D \in \mathcal{PD}$. Then*

1. *$\mathcal{H}(D)$ is the smallest structured partition DAG satisfying that $D \subseteq \mathcal{H}(D)$ and $\mathbf{T}(\mathcal{H}(D)) = \mathbf{T}$.*
2. *$t_1, t_2 \in \mathbf{T}$ are* syntactically D-equivalent, *iff D possesses a node γ satisfying that $t_1, t_2 \in \mathbf{T}_D(\gamma)$.*
3. *$t_1, t_2 \in \mathbf{T}$ are* semantically D-equivalent, *iff they are syntactically $\mathcal{H}(D)$-equivalent.*

[8] The equivalence relation $\top = \{\mathbf{T}\}$ has no corresponding structured partition DAG. It is rather an "artificial" greatest element, which is necessary here for completion.

In contrast to [SKR1], where the algorithm for determining the computation points is bi-directionally specified in the style of Morel and Renvoise [MR], it is decomposed here into a sequence of a backward and a forward analysis as in [KRS1,KS,St3] (cf. Section 5.1.1). These uni-directional analyses determine the predicates `D-Safe` and `Earliest`, respectively. Intuitively, the combined analysis yields the ***earliest down-safe*** computation points of a value, i.e. the "earliest" points satisfying that an inserted value is used on every terminating program path starting at an insertion point.

In addition to the corresponding step of [SKR1], the determination of computation forms must also exploit the distributive law and explicite knowledge about the availability of values in order to capture strength reduction. This is achieved by computing the predicates `Distr` and `Avail` (cf. Section 5.1.2), which highlights the essence of classical strength reduction: ***distributivity*** and ***availability***.

After this preparation, the placement procedure of step 2(ii) results in the following flow graph (cf. Section 5.2)[6]:

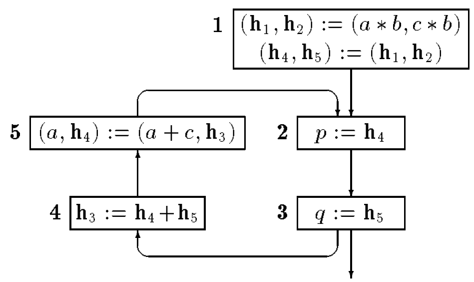

Figure 2.5

Subsequent ***variable subsumption*** [Ch,CACCHM] yields the desired result (Figure 2.1(b)).

3 Preliminaries

We consider terms $t \in \mathbf{T}$, which are inductively built from variables $v \in \mathbf{V}$, constants $c \in \mathbf{C}$ and operators $op \in \mathbf{Op}$. To keep our notation simple, we assume that all operators are binary. The ***semantics*** of terms of $\mathbf{T}$ is induced by the *Herbrand interpretation* $\mathbf{H} = (\mathbf{D}, \mathbf{H}_0)$, where $\mathbf{D} =_{df} \mathbf{T}$ denotes the non empty data domain and $\mathbf{H}_0$ the function, which maps every constant $c \in \mathbf{C}$ to the datum $\mathbf{H}_0(c) = c \in \mathbf{D}$ and every operator $op \in \mathbf{Op}$ to the total function $\mathbf{H}_0(op) : \mathbf{D} \times \mathbf{D} \to \mathbf{D}$, which is defined by $\mathbf{H}_0(op)(t_1, t_2) =_{df} (op, t_1, t_2)$ for all $t_1, t_2 \in \mathbf{D}$. $\Sigma = \{ \sigma \mid \sigma : \mathbf{V} \to \mathbf{D} \}$ denotes the set of all ***Herbrand states*** and σ_0 the distinct ***start state***, which is the identity on $\mathbf{V}$ (this choice of σ_0 reflects the fact that we do not assume anything about the context of the program being optimized). The ***semantics*** of terms $t \in \mathbf{T}$ is given by the *Herbrand semantics* $\mathbf{H} : \mathbf{T} \to (\Sigma \to \mathbf{D})$, which is inductively defined by:

[6] Note, also for the computation "$a + c$" at node **5** an auxiliary variable will be initialized at node **4**, a fact which we neglect here in order to keep the example simple.

$$\forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma.\ \ \mathbf{H}(t)(\sigma) =_{df} \begin{cases} \sigma(v) & \text{if } t = v \in \mathbf{V} \\ \mathbf{H}_0(c) & \text{if } t = c \in \mathbf{C} \\ \mathbf{H}_0(op)(\mathbf{H}(t_1)(\sigma), \mathbf{H}(t_2)(\sigma)) & \text{if } t = (op, t_1, t_2) \end{cases}$$

As usual, we represent imperative programs as ***directed flow graphs*** $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N and edge set E[7]. Nodes $n \in N$ represent ***parallel assignments*** of the form $(x_1, ., x_r) := (t_1, ., t_r)$, where $r \geq 0$ and $x_i = x_j$ implies $i = j$, and edges $(n, m) \in E$ the nondeterministic branching structure of G. $pred_G(n) =_{df} \{ m \mid (m, n) \in E \}$ denotes the set of all predecessors and $succ_G(n) =_{df} \{ m \mid (n, m) \in E \}$ the set of all successors of a node n, and $\mathbf{s}$ and $\mathbf{e}$ the unique ***start node*** and ***end node*** of G, which are assumed to possess no predecessors and successors, respectively. Furthermore, we assume that $\mathbf{s}$ and $\mathbf{e}$ represent the empty statement "skip", and that every node $n \in N$ lies on a path from $\mathbf{s}$ to $\mathbf{e}$. Finally, $\mathcal{T}(n)$ denotes the set of all terms occurring in the assignment represented by n, and $\mathcal{P}(\mathbf{T}) =_{df} \{ T \mid T \subseteq \mathbf{T} \}$, i.e. $\mathcal{P}$ denotes the power set operator.

For every node $n \equiv (x_1, ., x_r) := (t_1, ., t_r)$ of a flow graph G we define two functions

$\delta_n : \mathbf{T} \to \mathbf{T}$ by $\forall\, t \in \mathbf{T}.\ \delta_n(t) =_{df} t[t_1, ., t_r / x_1, ., x_r]$,

where $t[t_1, ., t_r / x_1, ., x_r]$ stands for the simultaneous replacement of all occurrences of x_i by t_i in t, $i \in \{1, ., r\}$, and $\theta_n : \Sigma \to \Sigma$, defined by:

$$\forall\, \sigma \in \Sigma\ \forall\, y \in \mathbf{V}.\ \ \theta_n(\sigma)(y) =_{df} \begin{cases} \mathbf{H}(t_i)(\sigma) & \text{if } y = x_i,\ i \in \{1, ., r\} \\ \sigma(y) & \text{otherwise} \end{cases}$$

δ_n realizes the backward substitution, and θ_n the state transformation caused by the assignment of node n. The Substitution Lemma 3.1 establishes a well-known relationship between δ_n and θ_n, which is central:

Lemma 3.1 (Substitution Lemma)

$$\forall\, n \in N\ \forall\, \sigma \in \Sigma\ \forall\, t \in \mathbf{T}.\ \mathbf{H}(\delta_n(t))(\sigma) = \mathbf{H}(t)(\theta(\sigma))$$

Proof. Let $n \equiv (x_1, ., x_r) := (t_1, ., t_r) \in N$, $\sigma \in \Sigma$, and $t \in \mathbf{T}$. The lemma is trivial for t being a constant or a variable $v \notin \{x_1, ., x_r\}$, and the case $t \equiv x_i \in \{x_1, ., x_r\}$ is dealt with by means of:

$$\mathbf{H}(\delta_n(t))(\sigma) = \mathbf{H}(\delta_n(x_i))(\sigma) = \mathbf{H}(t_i)(\sigma) = \mathbf{H}(x_i)(\theta_n(\sigma)) = \mathbf{H}(t)(\theta_n(\sigma))$$

This yields the base for the following induction on the structure of the term t.

For the induction step, we can assume that t is of the form $t \equiv (op, t_1, t_2)$. Applying the induction hypothesis $\mathbf{H}(\delta_n(t_i))(\sigma) = \mathbf{H}(t_i)(\theta(\sigma_n))$ for both subterms t_1 and t_2, we obtain:

$$\begin{array}{rcl} \mathbf{H}(\delta_n(t))(\sigma) & = & \mathbf{H}(\delta_n(\,(op, t_1, t_2)\,)(\sigma) \\ & = & \mathbf{H}_0(op)(\,\mathbf{H}(\delta_n(t_1))(\sigma), \mathbf{H}(\delta_n(t_2))(\sigma)\,) \\ \text{(IH)} & = & \mathbf{H}_0(op)(\,\mathbf{H}(t_1)(\theta_n(\sigma)), \mathbf{H}(t_2)(\theta_n(\sigma))\,) \\ & = & \mathbf{H}(t)(\theta_n(\sigma)) \end{array}$$

[7] These flow graphs are obtainable for example by the algorithm of [All].

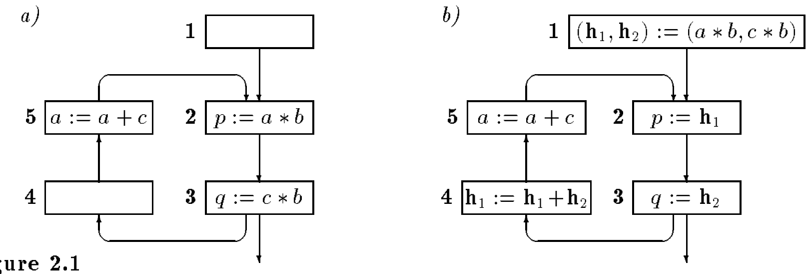

Figure 2.1

The semantic analysis of step 1(i) annotates the flow graph with partitions[4] that characterize all equivalences between terms with respect to the Herbrand interpretation, i.e. all equivalences that are valid independently of specific properties of the term operators (Figure 2.2). In particular, this analysis detects the equivalence of p and $a * b$, and of q and $c * b$ after the execution of node **3** (cf. Section 4.1).

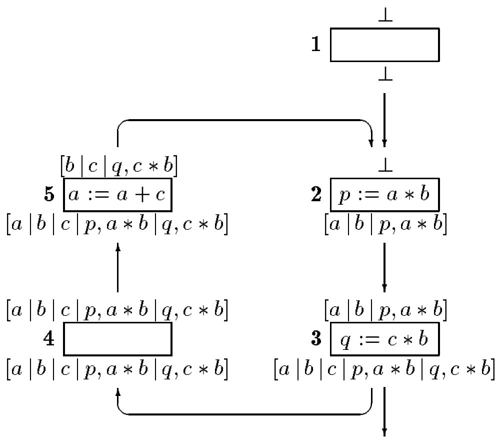

Figure 2.2

Afterwards, step 1(ii) computes for every program point a set of "relevant" terms, which contains a representation system of those equivalence classes that express all necessary equivalences syntactically (cf. Section 4.2), and extends the node annotation computed in step 1(i) accordingly. This (straightforward) extension is necessary, because the placement process of the second stage only refers to term equivalences that are explicit in the value flow graph under consideration, i.e. two terms are equivalent at a program point if and only if they are commonly represented by a node of the value flow graph at this point. In addition to the corresponding step of the algorithm of [SKR1], strength reduction requires to consider terms as relevant that arise from an application of arithmetic laws: the essence of classical strength reduction is to exploit the ***distributive*** law for sums and products, i.e. $(u + v) * w = u * w + v * w$. Therefore,

[4]Partitions are represented by means of structured partition DAGs (see Section 4.1).

whenever a term of the form $(u + v) * w$ is relevant, the terms $u * w$ and $v * w$ are also relevant[5]. Figure 2.3 shows the result of this extension.

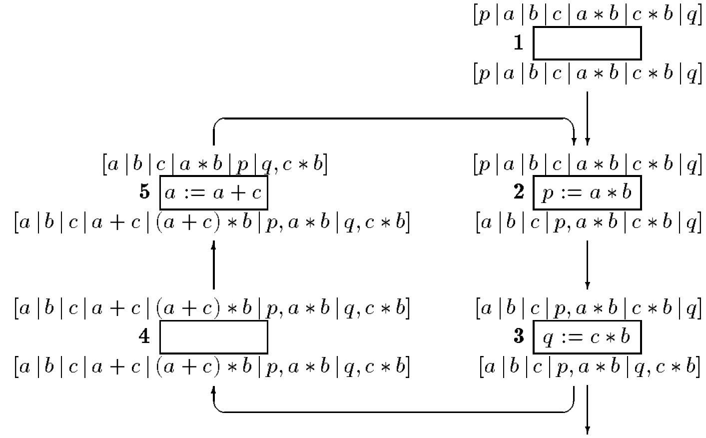

Figure 2.3

Step 1(iii) produces the corresponding value flow graph (cf. Section 4.3), whose relevant part is displayed in Figure 2.4:

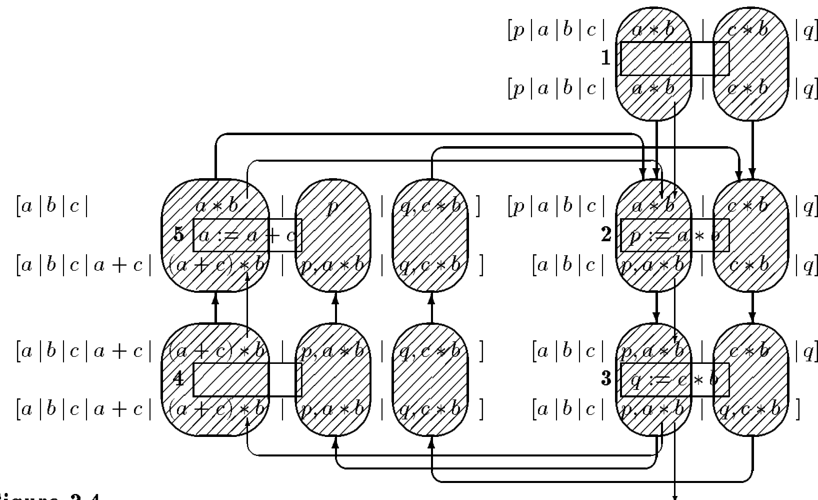

Figure 2.4

Applying a modification of the placement procedures of [KRS1,KS,St3] to the value flow graph above yields the required computation points and computation forms (step 2(i), Section 5.1).

[5]In the example, $(a + c) * b$ makes $a * b$ and $c * b$ relevant. In the special situation here these terms arise already after step 1(i).

1 Motivation

Strength reduction is a powerful technique for the optimization of loops, which improves run-time efficiency by reducing "expensive" operations, e.g. "$*$", to less expensive ones, e.g. "$+$". Its essence can be sketched as follows:

Let $x * y$ be a multiplication occurring in a loop L. Then try to eliminate all calculations of $x * y$ in L by performing the following three steps:

- Initialize a new auxiliary variable $\mathbf{h}$ with $x * y$ before entering L.
- Insert assignments of the form $\mathbf{h} := d \pm e$ in L that update $\mathbf{h}$ according to the redefinitions of x and y.
- Replace all occurrences of $x * y$ in L by $\mathbf{h}$.

Note, if no updating assignments are inserted, this three step procedure performs ***loop invariant code motion***. In fact, a clean realization of it should transform the flow graph of Figure 1.1(a) into the one displayed in Figure 1.1(b)[3]:

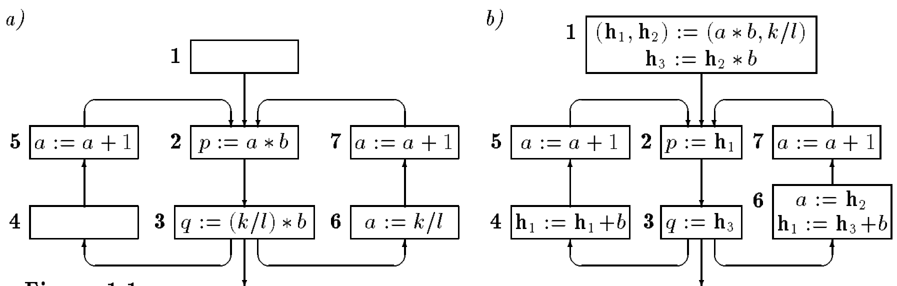

Figure 1.1

In this paper we present such a clean realization. It evolves as a uniform extension of the two stage algorithm of [SKR1], which optimally moves computations within programs with respect to Herbrand equivalence (cf. Section 3). In fact, this extension does not affect the structure of the underlying algorithm at all. It only requires two conceptual changes in the steps 1(ii) and 2(i), and a straightforward modification of step 2(ii):

1. Construction of a value flow graph:
   - (i) Determining for every program point a characterization of all term equivalences with respect to the Herbrand interpretation (Section 4.1).
   - (ii) Computing for every program point a finite set of "relevant" terms that allows us to syntactically represent enough term equivalences in order to perform our optimization (Section 4.2).
   - (iii) Connecting the representations of equivalence classes obtained in step 1(ii) according to the actual data flow in order to arrive at a value flow graph (Section 4.3).
2. Placement of the computations:
   - (i) Determining the computation points and computation forms with respect to the value flow graph obtained in step 1(iii) by a sequence of a backward and two forward analyses (Section 5.1).
   - (ii) Placing the computations (Section 5.2).

[3]However, to the best of our knowledge, all the published algorithms for strength reduction would fail this test.

This algorithm uniformly combines strength reduction and semantic code motion, where the underlying code motion algorithm optimally moves computations within programs with respect to Herbrand equivalence. The point of this approach is that it reduces strength reduction completely to the ***availability*** of values at the computation points. This allows us to overcome all restrictions concerning admissible program structures (e.g. previously detected loops) and admissible term structures (e.g. terms built of induction variables and region constants) that are required by previous strength reduction techniques (cf. Section 6). Moreover, it is the key for proving that program transformations obtained by our algorithm are guaranteed to be ***safe*** and to ***improve*** run-time efficiency. Both properties can be violated by previous techniques (cf. Section 6). The power of our algorithm that generalizes and improves the classical algorithms for strength reduction, common subexpression elimination, partial redundancy elimination, and loop invariant code motion is illustrated in the example of Figure 1.1(a), where it is unique in performing the optimization displayed in Figure 1.1(b).

Structure of the Paper

After illustrating our algorithm by means of a simple example in Section 2, we state the preliminary definitions in Section 3. Subsequently, we follow the structure of our algorithm in Section 4 and 5 and their subsections, discuss the related work concerning strength reduction in Section 6, and state our conclusions in Section 7. The Appendix contains the detailed algorithms.

2 A Simple Example

In this section we discuss the effects of the five steps of our two stage algorithm by means of the example of Figure 2.1(a), which will be transformed into the flow graph displayed in Figure 2.1(b). In the diagrams below arrows represent the ***nondeterministic branching structure*** and nodes ***(parallel) assignments***:

Unifying Strength Reduction and Semantic Code Motion[1]

Jens Knoop[2]
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preußerstr. 1 - 9
D–2300 Kiel 1

Bernhard Steffen
Lehrstuhl für Informatik II
Rheinisch-Westfälische Technische Hochschule Aachen
Ahornstr. 55
D–5100 Aachen

Aachener Informatikberichte
Nr. 91 – 28

Abstract

We present an algorithm that unifies strength reduction and semantic code motion. This algorithm evolves as a modular extension of the semantic code motion algorithm of [SKR1], which optimally moves computations within programs with respect to Herbrand equivalence. Our algorithm generalizes and improves all classical techniques for strength reduction in that it overcomes their structural restrictions concerning admissible program structures (e.g. previously determined loops) and admissible term structures (e.g. terms built of induction variables and region constants). Additionally, the program transformation obtained by our algorithm is guaranteed to be safe and to improve run-time efficiency. Previous techniques fail both of these properties.

Keywords: (loop invariant) code motion; common subexpression elimination; computation points and forms; data flow analysis; flow graphs (acyclic, irreducibel, nondeterministic); Herbrand interpretation and equivalence; imperative programming languages; partial redundancy elimination; program optimization; strength reduction; structured partition DAG; second order effects; value flow graph.

[1]This paper is an extended and revised version of the second part of [SKR2].
[2]The author is supported by the DFG grant La 426/9–2.

Contents