The Value Flow Graph: A Program Representation for Optimal Program Transformations

Bernhard Steffen * Jens Knoop † Oliver Rüthing†

Abstract

Data flow analysis algorithms for imperative programming languages can be split into two groups: first, into the *semantic* algorithms that determine *semantic equivalence* between terms, and second, into the *syntactic* algorithms that compute complex program properties based on syntactic term identity, which support powerful optimization techniques like for example *partial redundancy elimination*. *Value Flow Graphs* represent semantic equivalence of terms syntactically. This allows us to feed the knowledge of semantic equivalence into syntactic algorithms. The power of this technique, which leads to modularly extendable algorithms, is demonstrated by developing a two stage algorithm for the optimal placement of computations within a program wrt the *Herbrand interpretation*.

1 Introduction

There are two kinds of data flow analysis algorithms for imperative programming languages. First, the *semantic* algorithms that determine *semantic equivalence* between terms, e.g. the classical algorithm of Kildall [Ki1,Ki2]. Second, *syntactic* algorithms that compute complex program properties on the basis of syntactic term identity, which support powerful optimization techniques, e.g. Morel/Renvoise's algorithm for determining partial redundancies [MR]. *Value Flow Graphs* represent semantic equivalence syntactically. This allows us to feed the knowledge of semantic equivalence into syntactic algorithms. We will demonstrate the power of this technique by developing a two stage algorithm for the optimal placement of computations within a program wrt the *Herbrand interpretation*, which is structured as follows:

1. Construction of a Value Flow Graph for the Herbrand interpretation:
   (i) Determining term equivalence wrt the Herbrand interpretation (in short *Herbrand equivalence*) for every program point (Section 4.1).
   (ii) Computing a sufficiently large syntactic representation of the semantic equivalences for each program point (Section 4.2).
   (iii) Connecting the representations of equivalence classes of 1.(ii) according to the actual data flow. This results in a Value Flow Graph (Section 4.3).
2. Optimal placement of the computations:
   (i) Determining the computation points wrt the Value Flow Graph of step 1.(ii) by means of a Boolean equation system (Section 5.1).
   (ii) Placing the computations (Section 5.2).

*Department of Computer Science, University of Aarhus, DK-8000 Aarhus C — The work was done in part at the Laboratory for Foundations of Computer Science, University of Edinburgh

†Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, D-2300 Kiel 1 – The authors are supported by the Deutsche Forschungsgemeinschaft grant La 426/9-1

This is the only known algorithm of its kind that is (proved to be) optimal wrt the *Herbrand interpretation* for arbitrary control flow structures. It therefore generalizes and improves the known algorithms for common subexpression elimination, partial redundancy elimination and loop invariant code motion.

Rosen, Wegman and Zadeck developed an algorithm with a similar intent. However, they used a weaker representation for global semantic equivalence, the *static single assignment form* (SSA form), to represent global equivalence properties [RWZ]. Thus they could not apply the elegant and structurally independent technique of Morel and Renvoise [MR]. Rather they developed their own more complicated algorithm, which only works for particular program structures (reducible flow graphs). Moreover, their algorithm is only optimal for *acyclic* flow graphs (note, Herbrand equivalence is called *transparent equivalence* in [RWZ]).

It is worth mentioning that Steffen [St1,St2] and later Rosen, Wegman and Zadeck [RWZ] were the first who dealt with the *second order effects* of code motion. In our algorithm these effects are an automatic consequence of its optimality (see Corollary 5.7).

Practical experience with an implementation of our algorithm, which is implemented in a joint project with the NORSK DATA company, shows its practicality. In particular, all examples in this paper are computed by means of this implementation.

2 An Example

The following example illustrates the main features of our two stage algorithm. First, it works for arbitrary nondeterministic flow graphs (note that the loop construct of Figure 2.1 is not even reducible). Second, it considers semantic equivalence between terms.

The diagrams below represent the *nondeterministic branching structure* as arrows and *parallel assignments* as nodes:

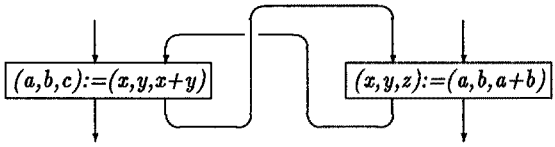

Figure 2.1

This program fragment has the following property:

while looping "$a+b$" and "$x+y$" evaluate to the same value

which suggests an optimization with the following result:

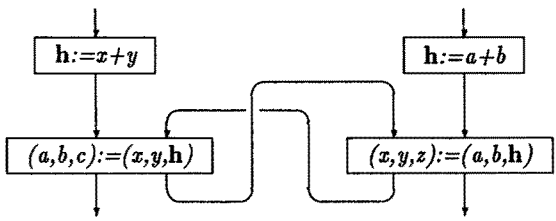

Figure 2.2

Already the basic variant (see 4.2) of our algorithm achieves this optimization. The following discussion demonstrates the effects of the five steps of our two stage algorithm.

The semantic analysis of step 1.(i) designates the flow graph with partitions characterizing all term equivalences wrt the Herbrand interpretation, i.e. all equivalences being valid independent of specific properties of the term operators (Figure 2.3). In particular, it detects the equivalence of "$a+b$" and "$x+y$" after the execution of either assignment.

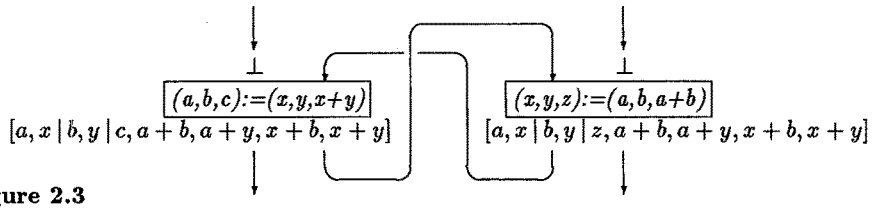

Figure 2.3

Afterwards, step 1.(ii) extends this designation for every program point to a syntactic representation of semantic equivalences which is large enough to perform our optimization:

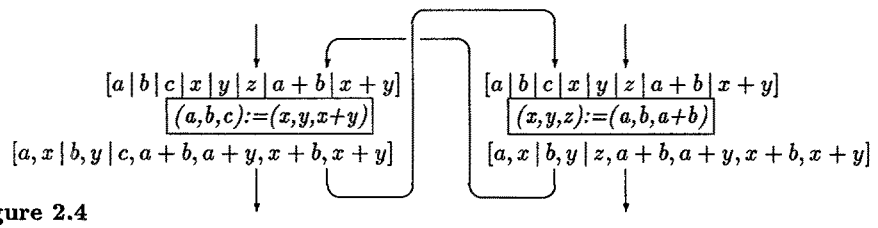

Figure 2.4

Subsequently, step 1.(iii) produces the corresponding Value Flow Graph, whose relevant part is shown in Figure 2.5:

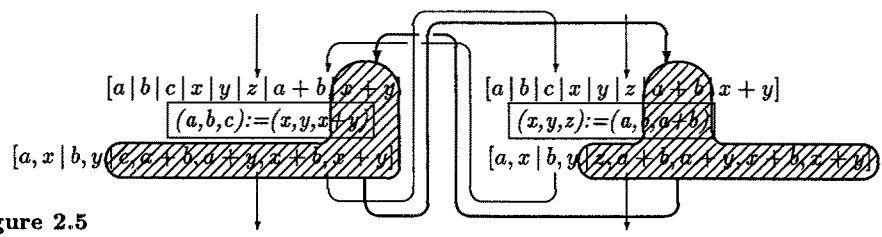

Figure 2.5

The placement procedure 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 they are displayed as members of the same equivalence class in the Value Flow Graph at this point. Applying a modification of Morel/Renvoise's algorithm (step 2.(i)) to the Value Flow Graph above yields:

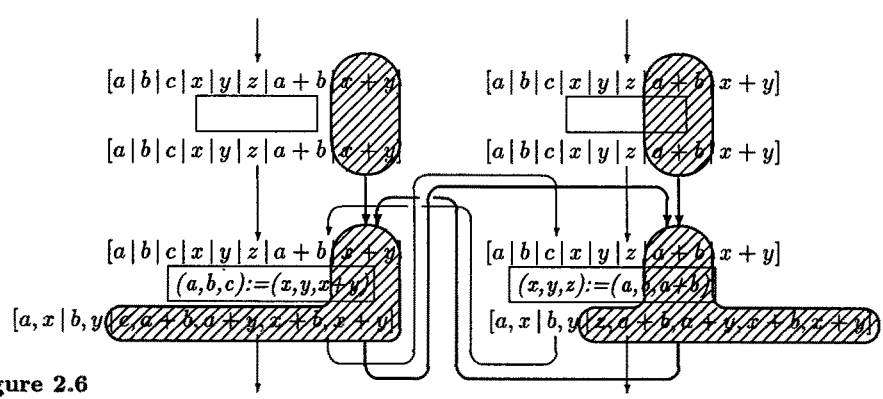

Figure 2.6

The inserted nodes are the optimal computation points (the insertion of synthetic nodes is common for code motion, see Section 5). – Now, application of step 2.(ii) results in:

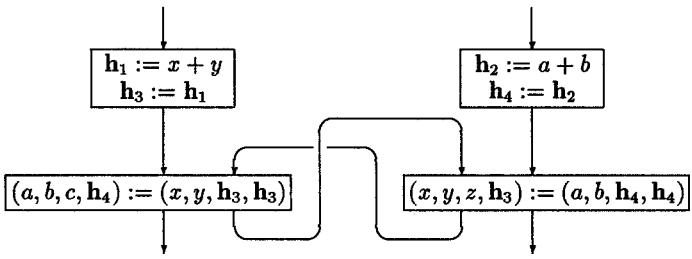

Figure 2.7

Subsequent *variable subsumption* [Ch,CACCHM] yields the desired result (Figure 2.2).

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 two-ary. However, an extension to operators of an arbitrary arity is straightforward. 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: $\forall \sigma \in \Sigma \; \forall t \in \mathbf{T}$.

$$\mathbf{H}(t)(\sigma) =_{df} \begin{cases} \sigma(v) & \text{if } t = v \in \mathbf{V} \\ \mathbf{H_0}(c) & \text{if } t = c \in \mathbf{C} \\ \mathbf{H_0}(op)(\mathbf{H}(t_1)(\sigma), \mathbf{H}(t_2)(\sigma)) & \text{if } t = op(t_1, t_2) \end{cases}$$

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. (These flow graphs are obtainable for example by the algorithm of [All]). 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 $i = j$, edges $(n, m) \in E$ the nondeterministic branching structure of G, and $\mathbf{s}$ and $\mathbf{e}$ denote 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 $\mathbf{e}$. The set of all such flow graphs is denoted by $\mathbf{FG}$.

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

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

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. Additionally, let $\mathcal{T}(n)$ denote the set of all terms which occur in the assignment represented by n.

A *finite path* of G is a sequence $(n_1,..,n_q)$ of nodes such that $(n_j,n_{j+1}) \in E$ for $j \in \{1,.,q-1\}$. $\mathbf{P}(n_1,n_q)$ denotes the set of all finite paths from n_1 to n_q 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 each path $p = (m \equiv n_1,.,n_q \equiv n) \in \mathbf{P}(m,n)$ we define $\Delta_p : \mathbf{T} \to \mathbf{T}$ by $\Delta_p =_{df} \delta_{n_q}$ if $q=1$ and $\Delta_{(n_1,.,n_{q-1})} \circ \delta_{n_q}$ otherwise, and $\Theta_p : \Sigma \to \Sigma$ by $\Theta_p =_{df} \theta_{n_1}$ if $q=1$ and $\Theta_{(n_2,.,n_q)} \circ \theta_{n_1}$ otherwise. The set of all *possible states* at a node $n \in N$ is given by

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

Now, we can define:

Definition 3.1 *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* $\forall \sigma \in \Sigma_n.\ \mathbf{H}(t_1)(\sigma) = \mathbf{H}(t_2)(\sigma)$.

4 Construction of a Value Flow Graph

The following subsections correspond to the three construction steps of a Value Flow Graph for a flow graph G, which we consider as to be given from now on.

4.1 Determining Local Semantic Equivalence

The semantic analysis determines all equivalences between program terms wrt the Herbrand interpretation (see 1.Optimality Theorem 4.6). These are expressed by means of structured partition DAGs (cp. [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 $\mathcal{P}_{fin} =_{df} \{T \mid T \subseteq (\mathbf{V} \cup \mathbf{C} \cup \mathbf{Op}) \wedge |T| \in \omega \setminus \{0\}\}$.

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

- *(N_D, E_D) is a directed acyclic multigraph 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. *$\forall \gamma \in N_D.\ |L_D(\gamma) \setminus \mathbf{V}| \leq 1$ 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 $r(\gamma)$.*
- *$\forall \gamma,\gamma' \in N_D.\ L_D(\gamma) \cap L_D(\gamma') \cap \mathbf{Op} \neq \emptyset \wedge l(\gamma) = l(\gamma') \wedge r(\gamma) = r(\gamma') \Rightarrow \gamma = \gamma'$.*

If N_D is finite, D is called a finite structured partition DAG. *The set of all structured partition DAGs and the set of all finite structured partition DAGs are denoted by $\mathcal{PD}$ and $\mathcal{PD}_{fin}$, respectively.*

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:

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

This can be illustrated as follows:

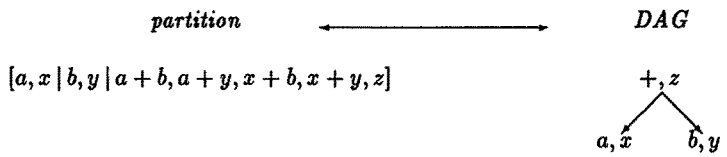

Figure 4.2

Viewing DAGs as equivalence relations makes $\mathcal{PD}$ a complete lattice, with inclusion defined set theoretically as usual. This guarantees existence and well definedness of $\mathcal{H}(P)$ in:

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

1. *$\mathcal{H}(D)$ is the smallest structured partition DAG with $D \subseteq \mathcal{H}(D)$ 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 with $t_1, t_2 \in \mathbf{T}_D(\gamma)$.*
3. *$t_1, t_2 \in \mathbf{T}$ are* semantically D-equivalent, *iff they are syntactically $\mathcal{H}(D)$-equivalent.*

We have (cf. [St2]):

Theorem 4.4 *Let $t_1, t_2 \in \mathbf{T}$, $n \in N$, and $pre[n] \in \mathcal{PD}_{fin}$ be the structured partition DAG of the entry information at node n computed by Algorithm A.1. Then t_1 and t_2 are Herbrand equivalent at node n iff they are semantically $pre[n]$-equivalent.*

Structured partition DAGs characterize the domain which is necessary to compute all term equivalences which do not depend on specific properties of the term operators. Moreover, they allow us to compute the effects of assignments essentially by updating the position of the left hand side variable:

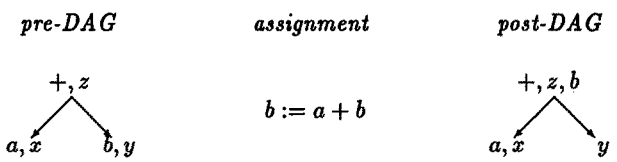

Figure 4.5

As a consequence of Theorem 4.4 we obtain:

Theorem 4.6 (1.Optimality Theorem)
Given an arbitrary flow graph, Algorithm A.1 terminates with a DAG-designation which exactly characterizes all equivalences of program terms wrt the Herbrand interpretation.

4.2 Computing the Syntactic Representation

In the last section we constructed finite structured partition DAGs that characterize Herbrand equivalence semantically (Definition 4.3(3)). However, the placement procedure (Section 5.1) considers the pre-DAGs and post-DAGs of a designation of a flow graph as purely syntactical objects, i.e. terms are considered equivalent iff they are syntactically equivalent (Definition 4.3(2)). Of course, it is not possible to finitely represent all Herbrand equivalences syntactically. However, it is possible to represent finite subsets that are sufficient for obtaining our optimality results (see Section 5.3). This is done by computing for every node n of G a finite set of terms T_{suf} that is

sufficient to represent all necessary equivalences at n syntactically, i.e. as the restriction of $\mathcal{H}(D)$ to T_{suf}.

Here, we sketch two strategies for the construction of such term closures. The first strategy associates every node n with the set of terms representing values that *must* be computed on every continuation of paths from $\mathbf{s}$ to n that end in $\mathbf{e}$, and the second strategy with the set of terms representing values that *may* be computed on a continuation of a path from $\mathbf{s}$ to n ending in $\mathbf{e}$. Both term closures are computed by backward analysis. The first strategy algorithm iteratively computes approximations of the closure for a node as the *meet* over the current approximations of the closures of its successors. The second strategy algorithm is essentially dual. However, it is necessary to constrain the iteration here because the straightforward dual algorithm would not terminate. These two strategies define the *basic* and *full* variant of our two stage algorithm.

There is another important variant of our algorithm, which we call RWZ-variant. It is based on a strategy for computing closures, which starts by invoking the first strategy algorithm. Subsequently, it applies this algorithm to all flow graphs that result from considering nodes as end nodes which possess at least one "brother".

4.3 The Value Flow Graph

A Value Flow Graph connects the term equivalence classes of a DAG designation according to the data flow. Essentially, its nodes are the equivalence classes and its edges representations of the data flow. For technical reasons we define the nodes of a Value Flow Graph as pairs of equivalence classes. However, identifying these pairs with their second component leads back to the original intuition, which will be referred to in the next section.

In the following let us assume that every node n of G is designated 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 $\dot{\bigcup}_{n \in N}(N_{\mathbf{pre}(n)} \times N_{\mathbf{post}(n)})$ by Γ and define a subset $\xleftarrow{\delta} \subseteq \Gamma$ (in the following $\gamma \xleftarrow{\delta} \gamma'$ stands for $(\gamma, \gamma') \in \xleftarrow{\delta}$) by:

$$\forall (\gamma, \gamma') \in \Gamma.\ \gamma \xleftarrow{\delta} \gamma' \iff_{df} \exists n \in N.\ \mathbf{T}_{\mathbf{pre}(n)}(\gamma) \supseteq \delta_n(\mathbf{T}_{\mathbf{post}(n)}(\gamma')).$$

Let now $\odot$ denote a new symbol, and $pred_G$ and $succ_G$ functions that map a node of G to its set of predecessors and successors, respectively. Then the technical definition of the Value Flow Graph for the DAG designation under consideration is as follows:

Definition 4.7 *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*

$$\nu = (\gamma_1, \gamma_2) \in VFN \iff_{df} \begin{cases} \gamma_1 \xleftarrow{\delta} \gamma_2 & \text{if } \gamma_1 \neq \odot \wedge \gamma_2 \neq \odot \\ \not\exists \gamma_3.\, \gamma_1 \xleftarrow{\delta} \gamma_3 & \text{if } \gamma_1 \neq \odot \wedge \gamma_2 = \odot \\ \not\exists \gamma_3.\, \gamma_3 \xleftarrow{\delta} \gamma_2 & \text{if } \gamma_1 = \odot \wedge \gamma_2 \neq \odot \end{cases}$$

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

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

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.

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 of a node n of G, such that γ_1 and γ_2 represent the same values, i.e. satisfy the inclusion $\mathbf{T_{pre(n)}}(\gamma_1) \supseteq \{t \mid \exists t' \in \mathbf{T_{post(n)}}(\gamma_2).\ t = \delta_n(t')\}$. Edges of the Value Flow Graph are pairs (ν, ν'), such that $\mathcal{N}(\nu)$ is a predecessor of $\mathcal{N}(\nu')$ and values are maintained along the connecting edge, i.e. $\mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu'))}(\nu'\downarrow_1) \subseteq \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu))}(\nu\downarrow_2)$. Finally, given a Value Flow Graph **VFG**, we define:

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

and

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

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

5 Optimal Placement of Computations

The placement procedure is optimal in its own right. It works for any Value Flow Graph, which need not be produced by the first stage algorithm or restricted to Herbrand equivalence.

Before going into details, let us mention a technicality, which is typical for code motion (cf. [RWZ]). Edges, leading from a node with more than one successor to a node with more than one predecessor, are split by insertion of a synthetic node. This is necessary in order to avoid "deadlock" during the code motion process, which may arise as illustrated in Figure 5.1(a). There the computation of "$a+b$" at node **3** is partially redundant wrt 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 node **2**, because this may introduce a new computation on a path which leaves node **2** on the right branch. On the other hand, it can safely be eliminated by moving the computation of "$a+b$" to the synthetic node **4** as it is displayed in Figure 5.1(b).

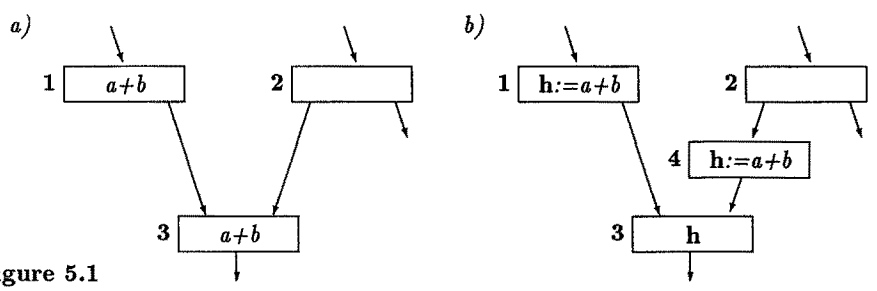

Figure 5.1

The following consideration assumes this simple transformation. In fact, the corresponding transformation of the Value Flow Graph is trivial as well, because all the inserted nodes represent skip-statements.

5.1 Determination of the Computation Points

The point of the placement procedure for computations is the solution of the following Boolean equation system (see Equation System 5.2), which we modified to work on Value Flow Graphs rather than flow graphs directly, in order to capture semantic equivalence. Following [MR], the names of the predicates are acronyms for the properties *"local anticipability"*, *"availability"* and *"placement possible"*:

Equation System 5.2 (Boolean Equation System)

- The Frame Conditions (Local Properties):

$$\mathbf{ANTLOC}(\nu) \Longleftrightarrow \nu\downarrow_1 \cap \mathcal{T}(\mathcal{N}(\nu)) \neq \emptyset$$

$$\mathbf{AVIN}(\nu) = \mathbf{PPIN}(\nu) = \mathit{false} \text{ if } \nu \in \mathit{VFN}_{\mathbf{S}}$$

$$\mathbf{PPOUT}(\nu) = \mathit{false} \text{ if } \nu \in \mathit{VFN}_{\mathbf{E}}$$

- The Fixed Point Equations (Global Properties):

$$\mathbf{AVIN}(\nu) \Longleftrightarrow \prod_{\nu' \in \mathit{pred}(\nu)} \mathbf{AVOUT}(\nu')$$

$$\mathbf{AVOUT}(\nu) \Longleftrightarrow \mathbf{AVIN}(\nu) \vee \mathbf{PPOUT}(\nu)$$

$$\mathbf{PPIN}(\nu) \Longleftrightarrow \mathbf{AVIN}(\nu) \wedge (\mathbf{ANTLOC}(\nu) \vee \mathbf{PPOUT}(\nu))$$

$$\mathbf{PPOUT}(\nu) \Longleftrightarrow \prod_{m \in \mathit{succ}(\mathcal{N}(\nu))} \sum_{\substack{\nu' \in \mathit{succ}(\nu) \\ \mathcal{N}(\nu') = m}} \mathbf{PPIN}(\nu')$$

Algorithm A.2 computes the greatest solution of this system, which determines the computation points by means of

$$\mathbf{INSERT}(\nu) =_{df} \mathbf{PPOUT}(\nu) \wedge \neg \mathbf{PPIN}(\nu)$$

5.2 Placing the Computations

The placement Algorithm A.3 proceeds in three steps:

1. It marks all nodes of the Value Flow Graph that occur on paths that lead from nodes satisfying **INSERT** to nodes satisfying **ANTLOC**.

2. It associates with every marked node of the Value Flow Graph 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. It initializes at every node of the Value Flow Graph satisfying **INSERT** its associated auxiliary variable by its initialization term. (Initialization terms of a node ν of the Value Flow Graph are minimal representatives of its corresponding equivalence class $\nu\downarrow_2$.)

   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.

Note, in order to eliminate all redundancies at once, the initializations of auxiliary variables are split into sequences of assignments that only have a single operator in their right hand side expression.

5.3 Optimality Results

An analysis of the Boolean Equation System 5.2 delivers not only the correctness of the derived program transformation, but also its optimality. Intuitively, a flow graph is defined to be optimal wrt a Value Flow Graph if it is "best" in the class of branching structure preserving flow graphs that are "safe" and "complete" for it. Here "best" means that it possesses a minimal number of computations on every path, and "safe" ("complete") that it computes on every path at most (at least) as many values. A formal definition of this notion of optimality is complicated, because all these properties need to be defined in terms of the Value Flow Graph. We will therefore only sketch the formal treatment. For this purpose we will assume (without loss of generality) that the synthetic nodes are already inserted, that linear sequences of nodes are abbreviated by a single node (basic block), and that the Value Flow Graph *covers* all computations of the underlying flow graph, i.e.:

$$\forall n \in N\ \forall t \in \mathcal{T}(n)\ \exists \nu \in VFN.\ \mathcal{N}(\nu) = n \wedge t \in \mathbf{T}_{\mathbf{pre}(n)}(\nu\downarrow_1)$$

Now, let **VFG** be a Value Flow Graph for a flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, and $G' = (N', E', \mathbf{s}', \mathbf{e}')$ a branching structure preserving flow graph for G, i.e. there exists a graph isomorphism Ψ from G' onto G with $\Psi(\mathbf{s}') = \mathbf{s}$ and $\Psi(\mathbf{e}') = \mathbf{e}$. Furthermore assume that $p \in \mathbf{P}(\mathbf{s}, \mathbf{e})$ and $p' = (n_1, .., n_q) \in \mathbf{P}(\mathbf{s}', \mathbf{e}')$ with $\Psi(p') = p$, and let $\mathbf{VFG}(p)$ denote the graph that results from unrolling **VFG** along the path p. Then $\mathbf{VFG}(p)$ is a collection of trees, which we will refer to as the **VFG**-values of p. This notion is motivated by the fact that $\mathbf{VFG}(p)$ defines an equivalence relation on (potential) term occurrences wrt p whose equivalence classes contain computations that evaluate to the same value during the execution of p, and that these classes are maximal such wrt **VFG**. Given a **VFG**-value C of p, $rg_p(C) =_{df} \{n_i \mid \exists \nu \in VFN_C.\ \mathcal{N}(\nu) = n_i\}$ denotes the *range* of C. A computation t' of p' at node n_i is Ψ-*covered* by a **VFG**-value C of p if $\Psi(n_i) \in rg_p(C)$ and if t' is covered by C at $\Psi(n_i)$, or if $\delta_{n_{i-1}}(t')$ is Ψ-covered by C at n_{i-1}. This complicated definition is necessary because a computation in p' need not match a term in **VFG**, for example because of additional (auxiliary) variables in G'. $C_{\mathbf{VFG}}(p)$ denotes the set of all **VFG**-values of p that cover at least one computation of p, and $C_{\mathbf{VFG}}(\Psi, p')$ the maximal set $\top$, or the smallest set of **VFG**-values of p which Ψ-cover all computations of p', if such a set exists.

After this preparation we are able to define the central notions of our optimality concept. G' is **VFG**-*safe* for G if $C_{\mathbf{VFG}}(\Psi, p') \subseteq C_{\mathbf{VFG}}(\Psi(p'))$, and it is **VFG**-*complete* for G if $C_{\mathbf{VFG}}(\Psi, p') \supseteq C_{\mathbf{VFG}}(\Psi(p'))$ for all $p' \in \mathbf{P}(\mathbf{s}', \mathbf{e}')$. Moreover, p' is *better* than p if it contains at most as many (non trivial) computations as p, and G' is *better* than G if p' is better than $\Psi(p')$ for all $p' \in \mathbf{P}(\mathbf{s}', \mathbf{e}')$. Finally, G is **VFG**-*optimal* if it is better than any branching structure preserving G' that is **VFG**-safe and **VFG**-complete for G.

Theorem 5.3 (2.Optimality Theorem)
Every flow graph transformed by the second stage of our algorithm is **VFG**-*optimal.*

Let us now consider the combined effect of the two stages of our algorithm. As mentioned already, it is not possible to finitely represent all Herbrand equivalences syntactically. However, using the 1.Optimality Theorem 4.6 we can show that there exists an infinite Value Flow Graph $\mathbf{VFG}_\infty$ that even represents all global Herbrand equivalences. This Value Flow Graph is the natural extension of a given Value Flow Graph where all partition DAGs D are replaced by $\mathcal{H}(D)$, see Definition 4.3.

Definition 5.4 *A* $\mathbf{VFG}_\infty$-*optimal program is called* Herbrand optimal.

We have:

Theorem 5.5 (Herbrand Optimality)
Every flow graph transformed by our two stage algorithm in the full variant is Herbrand optimal.

Herbrand optimal transformations may cause unboundedly many reinitializations of auxiliary variables in order to eliminate a single redundant computation. Thus, the costs of these reinitializations can easily exceed the costs of the eliminated computation. Motivated by this problem Rosen, Wegman and Zadeck introduced a notion of optimality, which is based on an additional technical constraint (see [RWZ] for details). Referring to this notion as RWZ-optimality we can prove:

Theorem 5.6 (*RWZ*-Optimality)
Every flow graph transformed by our two stage algorithm in the RWZ-variant is RWZ-optimal.

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

Finally, let $\mathtt{Trans} : \mathbf{FG} \to \mathbf{FG}$ be the operator specified by the full variant of our algorithm. Then we obtain by means of the 2.Optimality Theorem 5.3:

Corollary 5.7 $\mathtt{Trans}$ *is idempotent, i.e.* $\forall G \in \mathbf{FG}.\ \mathtt{Trans}(G) = \mathtt{Trans}(\mathtt{Trans}(G))$.

In particular, the full variant of our algorithm covers all second order effects (cf. [RWZ]).

6 Complexity

The second stage of our algorithm can be applied to arbitrary Value Flow Graphs, yielding optimal results relative to the equivalence information represented (see 2.Optimality Theorem 5.3). We therefore estimate the worst case time complexity, which as usual is based on the assumption of *constant branching* and *constant term depth*, independently for both stages. This requires the following three parameters: the number of nodes of a flow graph $\mathtt{n}$, the complexity of computing the meet of two equivalence informations $\mathtt{m}$, and the maximal number of Value Flow Graph nodes which are associated with a single node of the underlying flow graph, μ. This yields for the complexity of the five steps of our algorithm:

1. Construction of a Value Flow Graph for the Herbrand interpretation:

   (i) Determination of semantic equivalences: $O(\mathtt{n}^2 * \mathtt{m})$. Here "$\mathtt{n}^2$" reflects the maximal length of a descending chain of annotations of a flow graph. In fact, the number of analysis steps of Algorithm A.1 is linear in this chain length. This can be achieved by adding those nodes to a workset whose annotations have been changed. Then processing a worklist entry consists of updating the annotations of all its successors. This can be done in $O(\mathtt{m})$ because of our assumption of constant branching.

   To our knowledge, the exact nature of $\mathtt{m}$ is not studied in previous papers. This is probably due to the fact that, in practice, this effort hardly increases linearily in the size of the analysed program, and therefore is regarded as harmless. However, DAGs that arise during the analysis may represent sets of terms which increase exponentially in $\mathtt{n}$. Inspite of this fact, we conjecture that the compact representation of these sets by means of structured partition DAGs, together with the constraint that the DAGs arise during the analysis of a particular program, allows us to show that the number of nodes in such a DAG only increases quadratically in $\mathtt{n}$. This conjecture would suffice to prove an overall complexity of $O(\mathtt{n}^4)$ for the first step, because we know that the meet of two DAGs can be computed essentially linearily in the size of the resulting DAG.

(ii) Computation of the syntactic representation of the semantic equivalences: This complexity depends on the variant chosen. Whereas the basic variant and the RWZ-variant are both $O(n^3)$, the full variant seems to be exponential in n.

(iii) Construction of the Value Flow Graph: $O(n*\mu)$. This is based on two facts. First, if there exists an edge in the Value Flow Graph between two nodes ν_1 and ν_2 then the corresponding nodes $\mathcal{N}(\nu_1)$ and $\mathcal{N}(\nu_2)$ of the flow graph are connected as well. Thus every edge of the Value Flow Graph is associated with an edge of the original flow graph. Second, the effort to construct all edges of the Value Flow Graph that correspond to a single edge (n, m) in the original flow graph is linear in the number of Value Flow Graph nodes that annotate n, which can be estimated by μ.

2. Optimal placement of the computations:

(i) Determination of the computation points: $O(n*\mu)$. The argument needed here is based on that of the first step, however, we do not have constant branching, and the algorithm here is bidirectional. This leads to the product $n*\mu$ because all nodes of the Value Flow Graph can be updated once by executing only two elementary operations per edge of the Value Flow Graph, and the number of edges in a Value Flow Graph can be estimated by $O(n*\mu)$.

(ii) Placing the computations: $O(n*\mu)$. This is straightforward.

Let us finally give an estimation of the worst case time complexity of the practically motivated RWZ-variant. Here, $O(\mu)$ can be approximated by $O(n^2)$. In fact, exploiting the specific nature of the RWZ-closure already during the first step, we arrive at an algorithm with an overall complexity of $O(n^4)$. Assuming our conjecture, this result is also true for the RWZ-variant of our two stage algorithm presented above.

7 Conclusion

We have shown, how to combine semantic algorithms with syntactic ones, in order to obtain maximal optimization results. This technique, which is based on the introduction of Value Flow Graphs, has been illustrated by developing a two stage algorithm for the optimal placement of computations within a program.

In addition to their optimality, algorithms developed by means of this technique are easily to extend, because the separation of their semantic part from their (independently optimal) syntactic transformation part makes them modular. This modularity allows to independently enhance the semantic properties by modifying the first stage, and the transformation capacity by strenghtening the second stage. In our current implementation, the first stage is extended to deal with *constant propagation* and *constant folding* (see [SK1,SK2]). An extension of the second stage to *strength reduction* ([ACK,CK,JD1,JD2]) is under development.

Acknowledgements

The presentation in this paper profited from discussions with Torben Hagerup, Mark Jerrum, Barry Rosen and Ken Zadeck.

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, editors. "Program Flow Analysis: Theory and Applications", Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981

[Ch] G. J. Chaitin. *"Register Allocation and Spilling via Graph Coloring"*. IBM T. J. Watson Research Center, Computer Science Department, P.O. Box 218, Yorktown Height, N. Y. 10598, 1981

[CACCHM] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein. *"Register Allocation via Coloring"*. IBM T. J. Watson Research Center, Computer Science Department, P.O. Box 218, Yorktown Height, N. Y. 10598, 1980

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

[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

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

[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"*. TAPSOFT'87, 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, 1987

[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], LFCS Report Series, ECS-LFCS-89-79, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989

A Appendix: The Algorithms

Algorithm A.1 (The Semantic Analysis of Step 1.(i))

Input: *An arbitrary flow graph* $G=(N,E,\mathbf{s},\mathbf{e})$ *with unique start node* **s** *and unique stop node* **e**, *which are assumed to possess no predecessors and no successors, respectively.*

Output: *A designation of* G *with pre-DAGs (stored in pre) and post-DAGs (stored in post), characterizing valid and complete equivalence information at the entrance 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.* $[\![\ ]\!]$ *denotes the* local analysis component *and* $\sqcap$ *the* meet operation. *(*$[\![\ ]\!]$ *and* $\sqcap$ *operate on DAG-structures).* $pred(n)=_{df}\{m\,|\,(m,n)\in E\}$ *and* $succ(n)=_{df}\{m\,|\,(n,m)\in E\}$ *denote the set of all predecessors and successors of a node* n, *respectively. The variable workset controls the iterative process, and the auxiliary variable meet stores the result of the most recent meet operation.*

(Initialization of the designation arrays pre and post and the variable workset)
FOR *all nodes* $n \in N$ **DO**
  IF $n=\mathbf{s}$
    THEN $(pre[n],post[n]) := (\bot,[\![n]\!](\bot))$
    ELSE $(pre[n],post[n]) := (\top,\top)$ **FI**
OD;
$workset := \{\mathbf{s}\}$;

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
  LET $n \in workset$
    BEGIN
      $workset := workset\backslash\{n\}$;
      (Update the "environment" of node n)
      FOR *all nodes* $m \in succ(n)$ **DO**
        $meet := pre[m] \sqcap post[n]$;
        IF $pre[m] \sqsupset meet$
          THEN
            $pre[m] := meet$;
            $post[m] := [\![m]\!](pre[m])$;
            $workset := workset \cup \{m\}$
        FI
      OD
    END
OD.

Algorithm A.2 (Solution of the Boolean Equation System (Step 2(i)))

Input: *A Boolean equation system which is completely initialized wrt the local property ANTLOC.*

Output: *The greatest solution of the Boolean equation system.*

Remark: *With every node of the Value Flow Graph the predicates* **ANTLOC**, **AVIN**, **AVOUT**, **PPIN***and* **PPOUT** *are associated. However, only the later four are involved in the fixed point iteration, which therefore operates on the fourfold cartesian product of the complete semi-lattice* $\{false,true\}$ *with* $false \sqsubset true$. $pred(\nu)=_{df}\{\mu\,|\,(\mu,\nu)\in VFE\}$ *and* $succ(\nu)=_{df}\{\mu\,|\,(\nu,\mu)\in VFE\}$ *denote the set of all predecessors and successors of a node* ν, *respectively. The variable workset*

*controls the iterative process. For notational convenience we abbreviate (***AVIN**(ν), **AVOUT**(ν), **PPIN**(ν), **PPOUT**(ν)*) by* $\vec{p}_\nu$. *The auxiliary variable* $\vec{p}$ *stores the result of the most recent application of the local analysis component, and avin and ppout are further auxiliary variables.*

(Initialization)
FOR *all nodes* $\nu \in VFN$ **DO** $\vec{p}_\nu := (true, true, true, true)$ **OD**;
FOR *all nodes* $\nu \in VFN_{\mathbf{S}}$ **DO** $(\mathbf{AVIN}(\nu), \mathbf{PPIN}(\nu)) := (false, false)$ **OD**;
FOR *all nodes* $\nu \in VFN_{\mathbf{E}}$ **DO** $\mathbf{PPOUT}(\nu) := false$ **OD**;
$workset := VFN_{\mathbf{S}} \cup VFN_{\mathbf{E}}$;

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
  LET $\nu \in workset$
    BEGIN
      $workset := workset \setminus \{\nu\}$;
      (Update the "environment" of node ν *)*
      FOR *all nodes* $\mu \in pred(\nu) \cup succ(\nu)$ **DO**
        IF $\mu \in pred(\nu)$
          THEN
            $ppout := \mathbf{PPOUT}(\mu) \wedge \Sigma\{\mathbf{PPIN}(\lambda) \mid \lambda \in succ(\mu) \wedge \mathcal{N}(\lambda) = \mathcal{N}(\nu)\}$;
            $\vec{p} := (\mathbf{AVIN}(\mu), \mathbf{AVIN}(\mu) \vee ppout, \mathbf{AVIN}(\mu) \wedge (\mathbf{ANTLOC}(\mu) \vee ppout), ppout)$
          ELSE
            $avin := \mathbf{AVIN}(\mu) \wedge \mathbf{AVOUT}(\nu)$;
            $\vec{p} := (avin, avin \vee \mathbf{PPOUT}(\mu), avin \wedge (\mathbf{ANTLOC}(\mu) \vee \mathbf{PPOUT}(\mu)), \mathbf{PPOUT}(\mu))$

        FI;
        IF $\vec{p}_\mu \sqsupset \vec{p}$
          THEN
            $\vec{p}_\mu := \vec{p}$;
            $workset := workset \cup \{\mu\}$ **FI**
      OD
    END
OD.

Algorithm A.3 (The Optimizing Program Transformation (Step 2(ii)))

INPUT: *A flow graph* G *and an accociated Value Flow Graph* **VFG** *with attached predicate designation characterizing the greatest solution of the Boolean Equation System 5.2.*

OUTPUT: *The transformed flow graph* $G_{\mathbf{T}}$. *In* $G_{\mathbf{T}}$ *auxiliary variables are initialized at the optimal computation points by their minimal computation forms wrt to the equivalence information expressed by* **VFG**. *Original computations are replaced by auxiliary variables.*

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

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

The marking of a node $\nu \in VFN$ is indicated by the predicate $mark(\nu)$, the number of an associated auxiliary variable is denoted by $nr(\nu)$ and the variable count indicates the number of the last generated auxiliary variable.

(Phase 1: Marking of the definition-use chains in **VFG** *)*
FOR *all nodes* $\nu \in VFN$ **DO** $mark(\nu) := false$ **OD**;
$workset := \{ \nu \in VFN \mid \mathbf{ANTLOC}(\nu)\}$;
WHILE $workset \neq \emptyset$ **DO**
  LET $\nu \in workset$
    BEGIN
      $workset := workset \setminus \{\nu\}$;
      $mark(\nu) := true$;
      IF $\neg$**INSERT**(ν) **THEN** $workset := workset \cup \{\nu' \in pred(\nu) \mid \neg\ mark(\nu')\ \}$ **FI**
    END
OD;

(Phase 2: Allocating auxiliary variable numbers)
$count := 0$;
FOR *all nodes* $\nu \in VFN$ **DO** $nr(\nu) := 0$ **OD**;
$workset := \{\nu \in VFN \mid \mathbf{INSERT}(\nu)\}$;
WHILE $workset \neq \emptyset$ **DO**
  LET $\nu \in workset$
    BEGIN
      $workset := workset \setminus \{\nu\}$;
      $mark(\nu) := false$;
      IF **INSERT**$(\nu) \vee |pred(\nu)| \geq 2$
        THEN
          $count := count + 1$;
          $nr(\nu) := count$
        ELSE *(assume* $pred(\nu) = \{\nu'\}$ *)*
          $nr(\nu) := nr(\nu')$
      FI;
      $workset := workset \cup \{\nu' \in succ(\nu) \mid mark(\nu)\ \}$
    END
OD;

(Phase 3: Transformation of the flow graph)
(Initializing auxiliary variables at their computation points by their computation forms)
FOR *all nodes* $n \in N$ **DO**
  $workset := \{\nu \in VFN \mid \mathbf{INSERT}(\nu) \wedge \mathcal{N}(\nu) = n\ \}$;
  WHILE $workset \neq \emptyset$ **DO**
    LET $\nu \in \{\nu' \in workset \mid \forall \bar{\nu} \in workset\ .\ \bar{\nu}\downarrow_2 \notin succ^*(\nu'\downarrow_2)\}$
      BEGIN
        $workset := workset \setminus \{\nu\}$;
        IF $L_{\mathbf{post}(n)}(\nu\downarrow_2) \cap (\mathbf{V} \cup \mathbf{C}) \neq \emptyset$
          THEN
            LET $x \in L_{\mathbf{post}(n)}(\nu\downarrow_2) \cap (\mathbf{V} \cup \mathbf{C})$
              BEGIN
                Attach the assignment $\boxed{h_{nr(\nu)} := x}$ *at the end of node* n
              END

ELSE
LET $op \in L_{\mathbf{post}(n)}(\nu\downarrow_2)$,
$\nu' \in VFN.\ \nu'\downarrow_2 = l(\nu\downarrow_2)$,
$\bar{\nu} \in VFN.\ \bar{\nu}\downarrow_2 = r(\nu\downarrow_2)$
BEGIN
Attach the assignment $\boxed{h_{nr(\nu)} := h_{nr(\nu')}\ op\ h_{nr(\bar{\nu})}}$ *at the end of node* n
END
FI
END
OD
OD;

(Introducing spill code for the generated auxiliary variables)
FOR *all nodes* $\nu, \nu' \in \{\bar{\nu} \in VFN \mid nr(\bar{\nu}) \neq 0\}$ **DO**
IF $\nu \in pred(\nu') \wedge nr(\nu) \neq nr(\nu')$
THEN
Attach a component of spill code $\boxed{(.., h_{nr(\nu')}, ..) := (.., h_{nr(\nu)}, ..)}$ *at the end of node* $\mathcal{N}(\nu)$
FI
OD;

(Substituting original computations by a reference to their covering auxiliary variable)
FOR *all nodes* $n \in N$ **DO**
workset $:= \{\nu \in VFN \mid \mathbf{ANTLOC}(\nu) \wedge \mathcal{N}(\nu) = n\}$;
WHILE *workset* $\neq \emptyset$ **DO**
LET $\nu \in \{\nu' \in$ *workset* $\mid \forall \bar{\nu} \in$ *workset*. $\bar{\nu}\downarrow_1 \notin pred^*(\nu'\downarrow_1)\}$
BEGIN
workset $:=$ *workset* $\setminus \{\nu\}$;
Replace all original computations $t \in \mathcal{T}(n) \cap \mathbf{T}_{\mathbf{pre}(n)}(\nu\downarrow_1)$ *by* $h_{nr(\nu)}$
END
OD
OD.