UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Code Motion and Code Placement: Just Synonyms?

Jens Knoop[1]
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau
knoop@fmi.uni-passau.de

Oliver Rüthing
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preußerstraße 1–9
D–24105 Kiel
or@informatik.uni-kiel.de

Bernhard Steffen
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau
steffen@fmi.uni-passau.de

MIP-9716
Juli 1997

[1]The work of the author was partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

Abstract

We prove that there is no difference between *code motion* (*CM*) and *code placement* (*CP*) in the traditional *syntactic* setting, however, a dramatic difference in the *semantic* setting. We demonstrate this by re-investigating *semantic* CM under the perspective of the recent development of *syntactic* CM. Besides clarifying and highlightening the analogies and essential differences between the syntactic and the semantic approach, this leads as a side-effect to a drastical reduction of the conceptual complexity of the value-flow based procedure for semantic CM of [28], as the original bidirectional analysis is decomposed into purely unidirectional components. On the theoretical side, this establishes a natural semantical understanding in terms of the Herbrand interpretation (transparent equivalence), and thus eases the proof of correctness; moreover, it shows the frontier of semantic CM, and gives reason for the lack of algorithms going beyond. On the practical side, it simplifies the implementation and increases the efficiency, which, like for its syntactic counterpart, can be the catalyst for its migration from academia into industrial practice.

Keywords

Program optimization, data-flow analysis, code motion, code placement, partial redundancy elimination, transparent equivalence, Herbrand interpretation.

Contents

1 Motivation

Code motion (*CM*) is a classical optimization technique for eliminating *partial redundancies* (*PRE*).[2] Living in an ideal world a PRE-algorithm would yield the program of Figure 1(b) when applied to the program of Figure 1(a). A truly optimal result; even free of any redundancies!

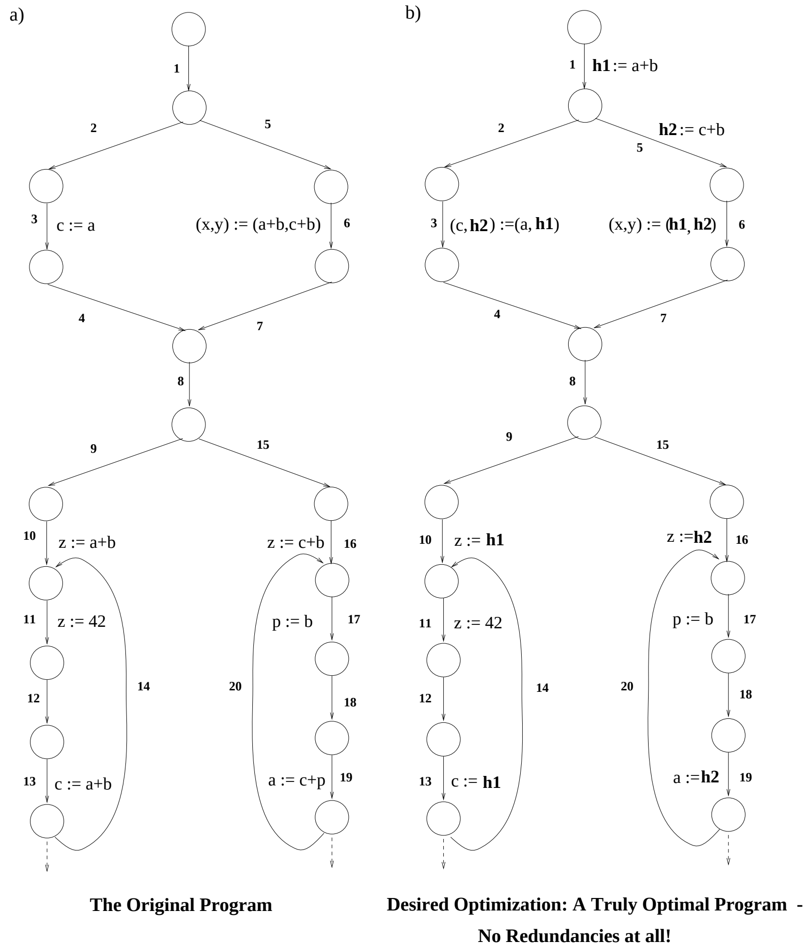

Figure 1: Living in an Ideal World: The Effect of Partial Redundancy Elimination

Unfortunately, the world is not that ideal. Up to now, there is no algorithm achieving the result of Figure 1(b). In reality, PRE is characterized by two different approaches for CM: a *syntactic* and a *semantic* one. Figure 2 illustrates their effects on the example of Figure 1(a). The point of *syntactic* CM is to treat all term patterns independently and to regard each assignment as destructive to any term pattern containing the left-hand side variable. In the example of Figure 1(a) it succeeds in eliminating the redundancy of $a + b$ in the left loop, but fails on the redundancy of $c + p$ in the right loop, which, because of the assignment to p, is not

[2]CM and PRE are often identified. To be precise, however, CM is a specific technique for PRE. As we are going to show here, identifying them is inadequate in general, and thus, we are precise on this distinction.

redundant in the "syntactic" sense inside the loop. In contrast, *semantic* CM fully models the effect of assignments, usually by means of a kind of symbolic execution (value numbering) or by backward substitution: by exploiting the equality of p and b after the assignment $p := b$, it succeeds in eliminating the redundant computation of $c + p$ inside the right loop as well. However, neither syntactic nor semantic CM succeeds in eliminating the partial redundancies at the edges **9** and **15**. This article is concerned with answering why: we will prove that redundancies like in this example are out of the scope of any "motion"-based PRE-technique. Eliminating them requires to switch from motion-based to "placement"-based techniques. This observation together with the highlightening of the analogies and differences between syntactic and semantic CM and CP as illustrated in Figures 2 (a) and (b), and Figure 1(b), respectively, are original contributions of this article.

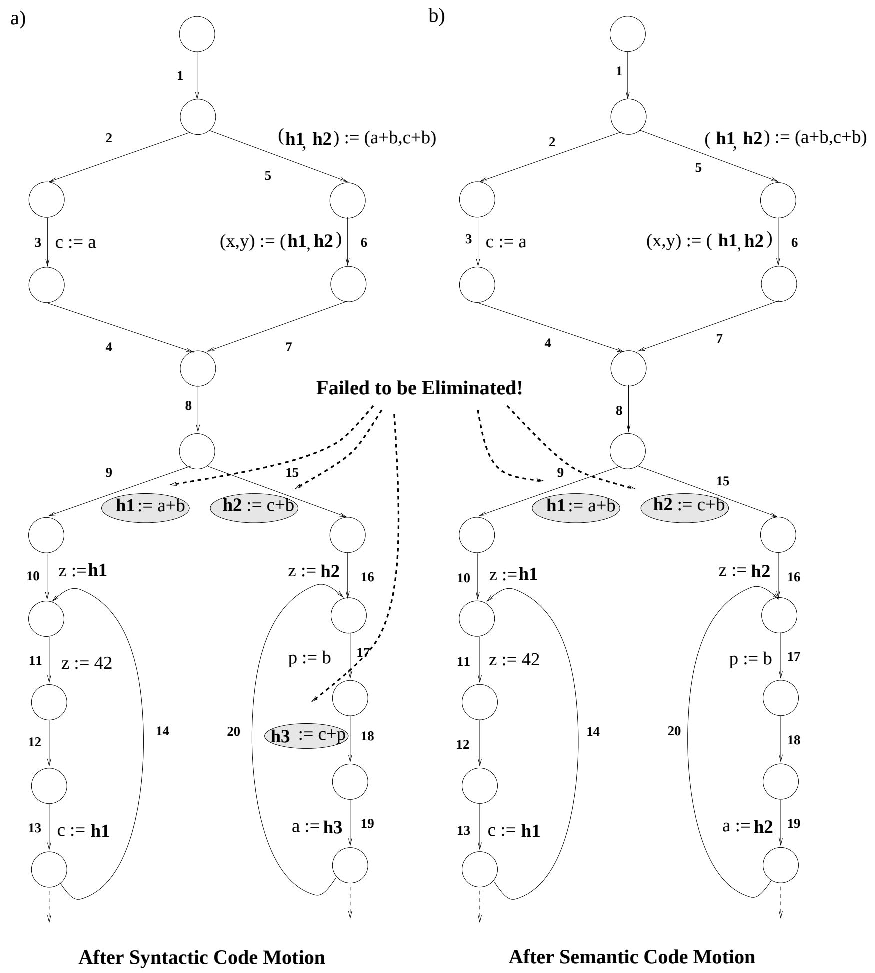

Figure 2: Back to Reality: Syntactic Code Motion vs. Semantic Code Motion

History and Current Situation: *Syntactic* CM (cf. [2, 6, 7, 8, 9, 10, 11, 14, 15, 18, 19, 21, 25]) is well-understood and integrated in many commercial compilers.[3]

[3]E.g., based on [18, 19] in the Sun SPARCompiler language systems (SPARCompiler is a registered trademark of SPARC International, Inc., and is licensed exclusively to Sun Microsystems,

In contrast, the much more powerful and aggressive *semantic* CM (see Figure 3, which illustrates its power on an irreducible program fragment)[4] has currently a very limited practical impact.

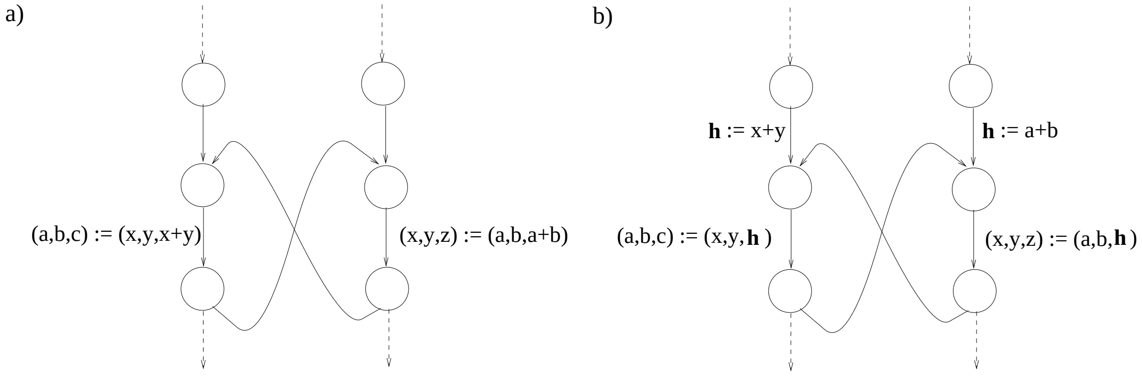

Figure 3: Illustrating the Power of Semantic Code Motion

In fact, only the "local" *value numbering* for basic blocks [4] is widely used. Globalizations of this technique can be classified into two categories: limited globalizations, where code can only be moved to dominators [3, 22, 23], and aggressive globalizations, where code can be moved more liberally [24, 28, 29]. The limited approaches are quite efficient, however, at the price of losing significant parts of the optimization power: they even fail in eliminating some of the redundancies covered by syntactic methods. In contrast, the aggressive approaches are rather complex, both conceptually and computationally, and are therefore considered impractical. This judgement is supported by the state-of-the-art here, which is still based on bidirectional analyses and heuristics making the proposed algorithms almost incomprehensible.

In this article we re-investigate (aggressive) semantic CM under the perspective of the very successful recent development of syntactic CM. This investigation highlights the conceptual analogies and differences between the syntactic and the semantic approach. In particular, it allows us to show that:

- the decomposition technique into unidirectional analyses developed in [18, 19] can be transferred to the semantic setting. Besides establishing a natural connection between the Herbrand interpretation (transparent equivalence) and the algorithm, which eases the proof of its correctness,[5] this decomposition leads to a more efficient and easier implementation. In fact, due to this simplification, we are optimistic that semantic CM, which is unique in capturing optimizations like the one sketched in Figure 3, will find its way into industrial practice.

Inc.).

[4]Note that this optimization, of which a detailed account was given in [26], is far beyond the scope of *syntactic* CM.

[5]Previously (cf. [28, 29]), this connection, which is essential for the conceptual understanding, had to be established in a very complicated indirect fashion.

- there is a significant difference between *motion* and *placement* techniques (see Figures 1 and 2), which only shows up in the semantic setting. The point of this example is that the computations of $a + b$ and $c + b$ cannot safely be "moved" to their computation points in Figure 1(b), but they can safely be "placed" there (see Figure 4 for an illustration of the essentials of this example).

The major contributions of this article are thus as follows. On the *conceptual* side: (1) Uncovering that CM and CP are no synomyms in the semantic setting (but in the syntactic one), (2) showing the frontier of semantic CM, and (3) giving theoretical and practical reasons for the lack of algorithms going beyond! On the *technical* side, though almost as a side-effect, equilibrially, presenting a new algorithm for computationally optimal semantic CM, which is conceptually and technically much simpler as its precursor of [28].

Whereas the difference between motion and placement techniques will primarily be discussed on a conceptual level, the other points will be treated in detail. In particular, it will be shown that the straightforward transfer of the syntactic technique to the semantic setting has *second-order effects*, which, however, can be avoided completely by means of a hierarchical organization of the computation.

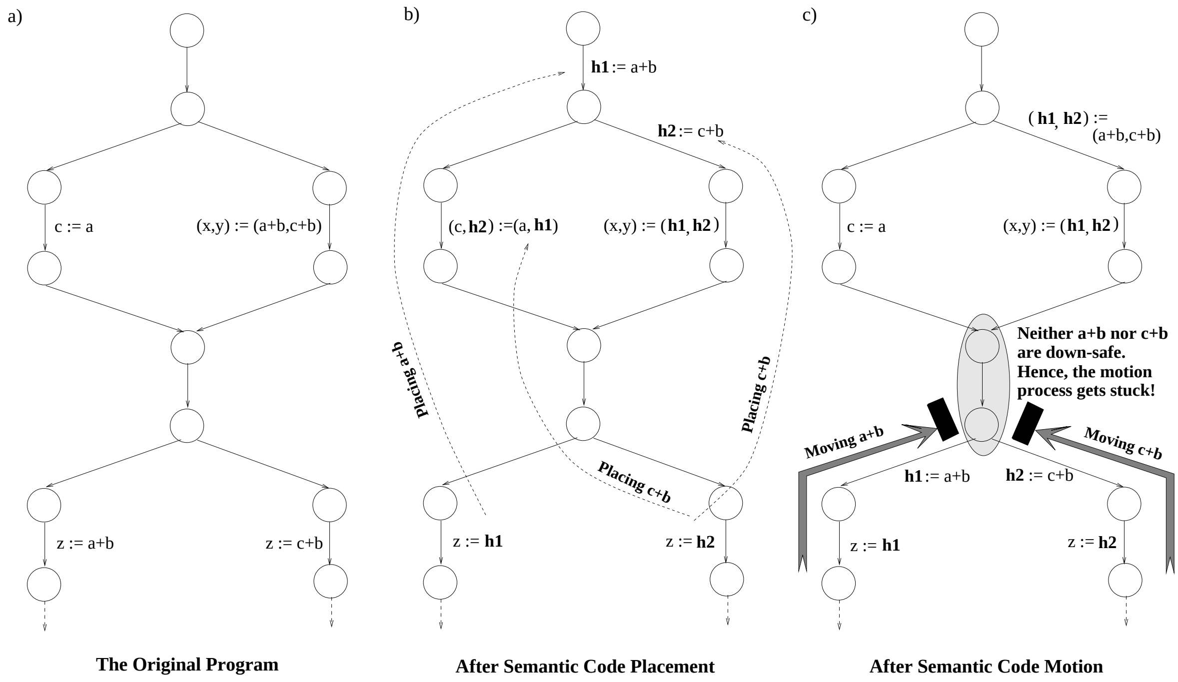

Figure 4: Illustrating the Difference: Sem. Code Placement vs. Sem. Code Motion

Safety – The Backbone of Code Motion: Key towards the understanding of the conceptual difference between syntactic and semantic CM is the notion of *safety* of a program point for some computation: intuitively, a program point is safe for a given computation, if the execution of the computation at this point results in a value that is guaranteed to be computed by every program execution passing the point.

Similarly, *up-safety* (*down-safety*) is defined by requiring that the computation of the value is guaranteed before meeting the program point for the first time (after meeting the program point for the last time).

As properties like these are undecidable in the standard interpretation (cf. [22]), decidable approximations have been considered. Prominent are the abstractions leading to the syntactic and semantic approach considered in this article. Concerning the safety notions established above the following result is responsible for the simplicity and elegance of the syntactic CM-algorithms (cf. [19]):

Theorem 1.1 (Syntactic Safety) *Safe* = *Up-safe* $\vee$ *Down-safe*

It is the failure of this equality in the semantic setting, which causes most of the problems of semantic CM, because the decomposition of safety in up-safety and down-safety is essential for the elegant syntactic algorithms. Figure 5 illustrates this failure as follows: placing the computation of $a + b$ at the boldface join-node is (semantically) safe, though it is neither up-safe nor down-safe.

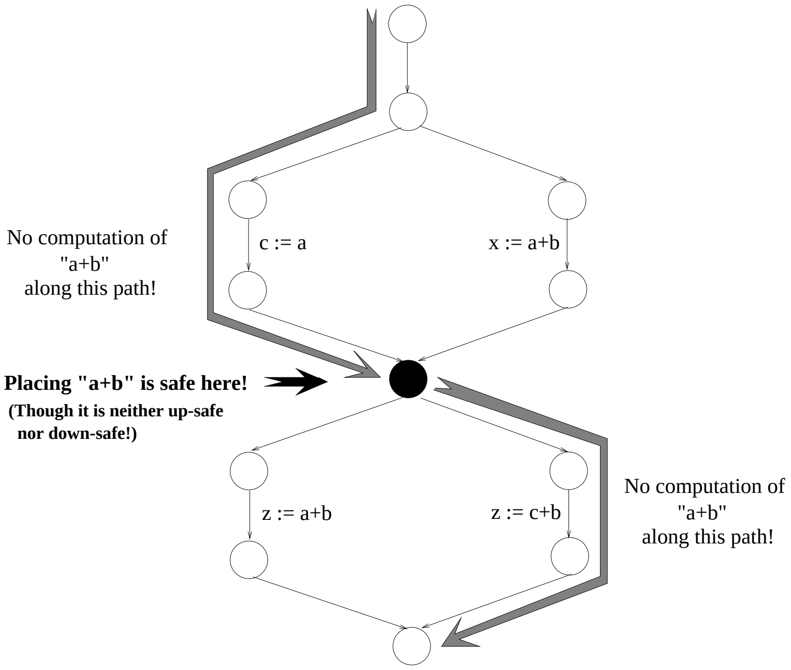

Figure 5: Safe though neither Up-Safe nor Down-Safe

As a consequence, simply transferring the algorithmic idea of the syntactic case to the semantic setting without caring about this equivalence results in an algorithm for CM with *second-order effects* (cf. [5, 24]). These can be avoided by defining a *motion-oriented* notion of safety, which allows to reestablish the equality for a hierarchically defined notion of up-safety: the algorithm resulting from the use of these notions captures all the second-order effects of the "naively transferred" algorithm as well as the results of the original bidirectional version for semantic CM of [28].

Motion versus Placement: The step from the motion-oriented notion of safety to "full safety" can be regarded as the step from *motion*-based algorithms to *placement*-

based algorithms: in contrast to CM, CP is characterized by allowing arbitrary (safe) placements of computations with subsequent (*total*) *redundancy elimination* (*TRE*). As illustrated in Figure 4, not all placements can be realized via motion techniques, which are characterized by allowing the code movement only within areas where the placement would be correct. The power of arbitrary placement leads to a number of theoretic and algorithmic complications and anomalies (cf. Section 5), which we conjecture, can only be solved by changing the graph structure of the argument program, e.g. along the lines of [27]. Cytron et al. [5] and similarly Ferrante et al. [12] proposed two approaches of how to move computations across points where a straightforward motion would not be safe. Basically, the idea is to move complete control-structures instead of right-hand side computations, if a computation like $c+p$ is blocked by a modification like $p := b$ in Figure 2(a). However, on the one hand side, this is a different class of transformations, and on the other hand side it still stays to the syntactic setting, and does not contribute to the problem displayed in Figure 4(c).

Retrospectively, the fact that all CM-algorithms arise from notions of safety, which collapse in the syntactic setting, suffices to explain that the syntactic algorithm does not have any second-order effects, and that there is no difference between "motion" and "placement" algorithms in the syntactic setting.

Theorem 1.2 (Syntactic CM and Syntactic CP)
In the syntactic setting, CM is as powerful as CP (*and vice versa*).

Structure of the Article: After the preliminary Section 2, we recall in Section 3 the basic structure and the essential concepts underlying the semantic CM-algorithm of [28], in particular the concept of a *value-flow graph*, which gives us a handle to represent global equivalence information syntactically. Central are then the Sections 4.1, 4.2, and 5, where it is demonstrated that we obtain, depending on the considered notions of safety, three algorithms of different characteristics and increasing power. Finally, we draw our conclusions in Section 6.

2 Preliminaries

We consider procedures of imperative programs, which we represent by means of directed edge-labeled *flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, a unique *start node* $\mathbf{s}$ and *end node* $\mathbf{e}$, which are assumed to have no incoming and outgoing edges, respectively. The edges of G represent both the statements and the nondeterministic control flow of the underlying procedure, while the nodes represent program points only. Statements are *parallel assignments* of the form $(x_1, \ldots, x_r) := (t_1, \ldots, t_r)$, where x_i are pairwise disjoint variables, and t_i terms of the set $\mathbf{T}$, which as usual are inductively composed of variables, constants, and operators. For $r=0$, we obtain the empty statement "skip". Unlabeled edges are assumed to represent "skip". Without loss of generality we assume that edges starting in nodes with more than one successor are labeled by "skip".[6]

We remark that the source and the destination node of a flow-graph edge corresponding to a node n of a traditional node-labeled flow graph represent the usual distinction between the *entry* and the *exit* point of n explicitly. This simplifies the formal development of the theory significantly, particularly the definition of the value-flow graph in Section 3.1 because the implicit treatment of this distinction, which, unfortunately is usually necessary for the traditional flow-graph representation, is obsolete here; a point which is intensely discussed in [17].

For a flow graph G, let $pred_G(n) =_{df} \{ m \mid (m,n) \in E \}$ and $succ_G(n) =_{df} \{ m \mid (n,m) \in E \}$ denote the set of all immediate predecessors and successors of a node n, and let $source(e)$ and $dest(e)$ denote the source and the destination node of an edge e. A *finite path* in G is a sequence $\langle e_1, \ldots, e_q \rangle$ of edges such that $dest(e_j) = source(e_{j+1})$ for $j \in \{1, \ldots, q-1\}$. It is a path from m to n, if $source(e_1) = m$ and $dest(e_q) = n$. Additionally, $p[i,j]$, where $1 \leq i \leq j \leq q$, denotes the *subpath* $\langle e_i, \ldots, e_j \rangle$ of p. Finally, $\mathbf{P}_G[m,n]$ denotes the set of all finite paths from m to n. Without loss of generality we assume that every node $n \in N$ lies on a path from $\mathbf{s}$ to $\mathbf{e}$.

3 Semantic Code Motion

In essence, the reason making semantic CM much more intricate than syntactic CM is that in the semantic setting safety is not the sum of up-safety and down-safety (i.e., does not coincide with their disjunction). In order to illustrate this in more detail, we consider as in [24] *transparent equivalence* of terms, i.e., we fully treat the effect of assignments, but we do not exploit any particular properties of term operators.[7] Moreover, we essentially base our investigations on the semantic CM-algorithm of [28] consisting of two major phases:

- *Preprocess:* a) Computing transparent equivalences (cf. Section 3.1).
  b) Constructing the *value-flow graph* (cf. Section 3.1).
- *Main Process:* Eliminating semantic redundancies (cf. Section 4).

After computing the transparent equivalences of program terms for each program point, the preprocess globalizes this information according to the value flow of the program. The value-flow graph is just the syntactic representation for storing the global flow information. Based on this information the main process eliminates semantically redundant computations by appropriately placing the computations in the program. In the following section we sketch the preprocess as far it is necessary for the development here, while we investigate the main process in full detail. In comparison to [28] it is completely redesigned.

[6] Our approach is not limited to a setting with scalar variables. However, we do not consider subscripted variables here in order to avoid burdening the development by alias-analyses.

[7] In [28] transparent equivalence is therefore called *Herbrand equivalence* as it is induced by the Herbrand interpretation.

3.1 The Preprocess: Constructing the Value-Flow Graph

The first step of the preprocess computes for every program point the set of all transparently equivalent program terms. The corresponding algorithm is fairly straightforward and matches the well-known pattern of Kildall's algorithm (cf. [16]). As a result of this algorithm every program point is annotated by a *structured partition* (*SP*) *DAG* (cp. [13]), i.e., an ordered, directed, acyclic graph whose nodes are labeled with at most one operator or constant and a set of variables (cf. Figure 7). The SPDAG attached to a program point represents the set of all terms being transparently equivalent at this point: two terms are transparently equivalent iff they are represented by the same node of an SPDAG. For example, the SPDAG of Figure 6 is an economical representation of the term equivalences $[g, h, 2-m, 2-n \,|\, e \,|\, k, g*e, h*e, (2-m)*e, (2-n)*e, \,|\, z, x+42, \ldots \,|\, \ldots]$. Figure 7, subsequently, shows the SPDAG-annotation resulting from the first step of our algorithm in the context of a sample program. It will be discussed in more detail in Section 4.2.

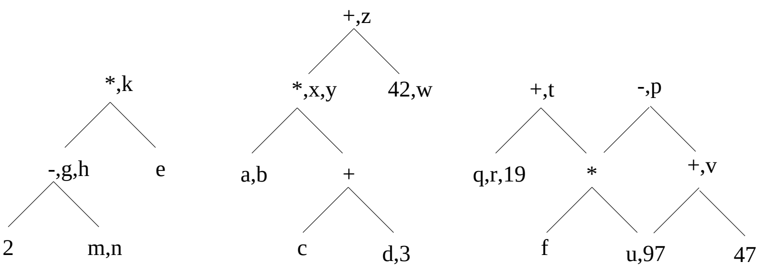

Figure 6: A Structured Partition DAG

Afterwards, the *value-flow graph* is constructed. Intuitively, it connects the

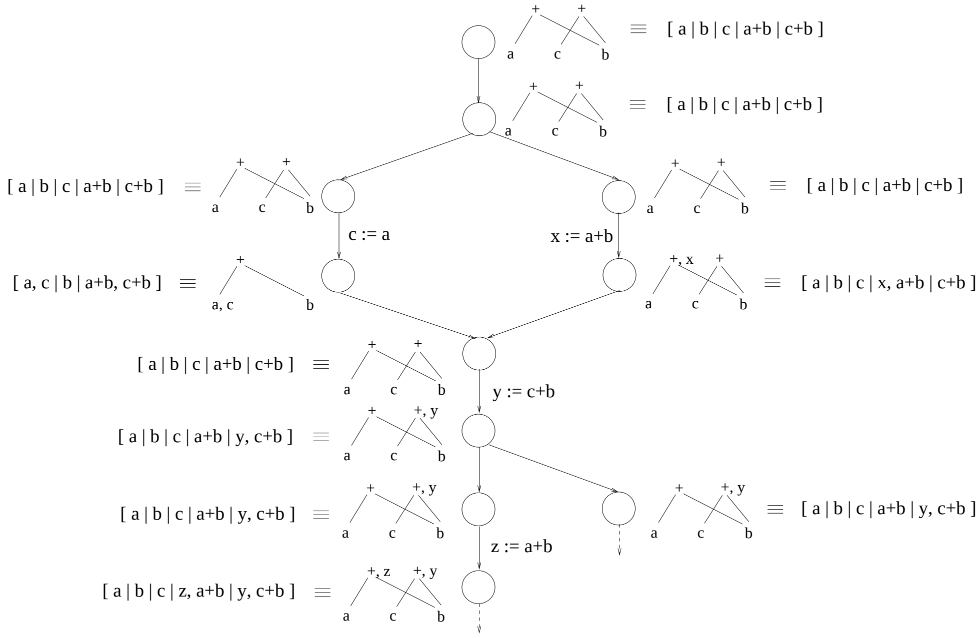

Figure 7: The SPDAG-Annotation of an Illustrating Sample Program

nodes, i.e., the term equivalence classes of the SPDAG-annotation $\mathcal{D}$ computed in the previous step according to the data flow. Thus, its nodes are the classes of transparently equivalent terms, and its edges are the representations of the data flow: if two nodes ν and ν' of a value-flow graph are connected by a value-flow graph edge (ν, ν'), then the terms represented by ν' evaluate to the same value after executing the flow-graph edge e corresponding to (ν, ν') as the terms represented by ν before executing e (cf. Figure 8). This is made precise by means of the relation $\xleftarrow{\delta}$ defined next. To this end, let Γ denote the set of all nodes of the SPDAG-annotation $\mathcal{D}$. Moreover, let $\mathit{Terms}(\nu)$ denote the set of all terms represented by node ν of Γ, and let $\mathcal{N}(\nu)$ denote the flow-graph node ν is related to. Central is the definition of the *backward substitution* δ, which is defined for each flow-graph edge $e \equiv (x_1, \ldots, x_r) := (t_1, \ldots, t_r)$ by:

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

where $t[t_1, \ldots, t_r / x_1, \ldots, x_r]$ stands for the simultaneous replacement of all occurrences of x_i by t_i in t, $i \in \{1, \ldots, r\}$.[8] The relation $\xleftarrow{\delta}$ on Γ is now defined by (in the following $\nu \xleftarrow{\delta} \nu'$ stands for $(\nu, \nu') \in \xleftarrow{\delta}$):

$$\forall\, (\nu, \nu') \in \Gamma.\ \nu \xleftarrow{\delta} \nu' \iff_{df} \exists\, e \in E.\ e \equiv (\mathcal{N}(\nu), \mathcal{N}(\nu')) \wedge \mathit{Terms}(\nu) \supseteq \delta_e(\mathit{Terms}(\nu'))$$

The value-flow graph for a SPDAG-designation $\mathcal{D}$ is then as follows:

Definition 3.1 (Value-Flow Graph)
The value-flow graph *with respect to $\mathcal{D}$ is a pair $VFG = (VFN, VFE)$ consisting of*

- *a set of nodes $VFN =_{df} \Gamma$ called* abstract values *and*
- *a set of edges $VFE \subseteq VFN \times VFN$ with $VFE =_{df} \xleftarrow{\delta}$.*

Figure 8 shows the value-flow graph resulting from the SPDAG-annotation of Figure 7 for illustration.[9] We recall that the nodes of the value-flow graph represent classes of transparently equivalent terms, whereas its edges represent the global data flow, i.e., the value represented by the terms of the start and end node of a value-flow graph edge is maintained along the flow-graph edge it is related to.[10] It is worth noting that the value-flow graph definition given above is technically much simpler than its original version of [28]. This is simply a consequence of representing procedures here by edge-labeled flow graphs instead of node-labeled flow graphs as in [28].

VFG-**Redundancies**: In order to define the notion of (partial) redundancies with respect to a value-flow graph *VFG*, we need to extend the local predicate *Comp* for terms known from syntactic CM to the abstract values represented by value-flow graph nodes. In the syntactic setting $\mathit{Comp}_t(e)$ expresses that the term t is computed at the edge e under consideration, i.e., it is a sub-term of some right-hand

[8] Note, for edges labeled by "skip" the function δ_e equals the identity on terms $Id_{\mathbf{T}}$.
[9] The safety properties will be discussed later.
[10] In the syntactic setting, this is called the value is "transparent" along this edge.

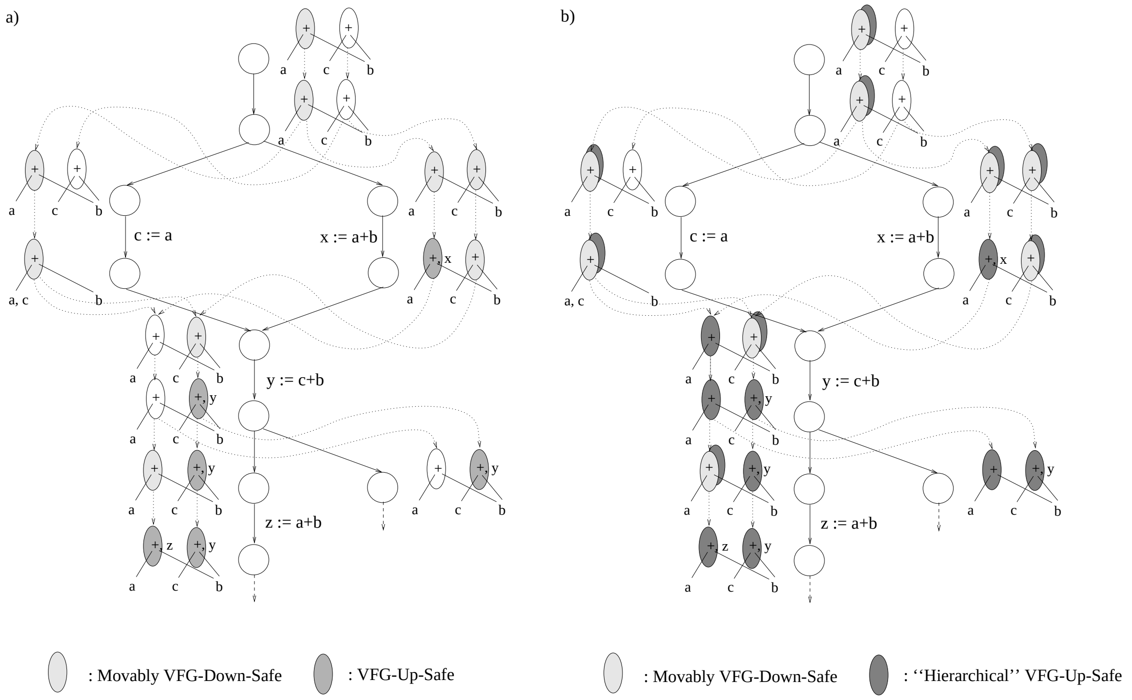

Figure 8: The Value-Flow Graph for the Sample Program of Figure 7

side term of the statement of the flow-graph edge under consideration. Analogously, for every abstract value $\nu \in \mathit{VFN}$ the local predicate $\mathit{Comp}(\nu)$ expresses that the statements of the corresponding outgoing flow-graph edges compute a term represented by ν:[11]

$$\mathit{Comp}(\nu) \iff \forall\, e \in E.\ \mathit{source}(e) = \mathcal{N}(\nu).\ \mathit{Terms}(\nu) \cap \mathit{Terms}(e) \neq \emptyset$$

Here $\mathit{Terms}(e)$ denotes the set of all terms occurring on the right-hand side of the statement of edge e.

In addition, we need the notion of *correspondence* between value-flow graph paths and flow-graph paths. Let $p = \langle e_1, \ldots, e_q \rangle \in \mathbf{P}_G[m, n]$ be a path in the flow graph G, and let $p' = (\varepsilon_1, \ldots, \varepsilon_r) \in \mathbf{P}_{VFG}[\nu, \mu]$ be a path in a value-flow graph VFG of G. Then p' is a *corresponding VFG-prefix* of p, if for all $i \in \{1, \ldots, r\}$ holds: $\mathcal{N}(\mathit{source}(\varepsilon_i)) = \mathit{source}(e_i)$ and $\mathcal{N}(\mathit{dest}(\varepsilon_i)) = \mathit{dest}(e_i)$. Analogously, the notion of a *corresponding VFG-postfix* p' of p is defined. We can now define:

Definition 3.2 (*VFG*-Redundancy)
Let VFG be a value-flow graph, let n be a node of G, and t be a term of $\mathbf{T}$. Then t is

1. partially *VFG*-redundant *at n, if there is a path $p = \langle e_1, \ldots, e_q \rangle \in \mathbf{P}_G[\mathbf{s}, n]$ with a corresponding VFG-postfix $p' = \langle \varepsilon_1, \ldots, \varepsilon_r \rangle \in \mathbf{P}_{VFG}[\nu, \mu]$ such that $\mathit{Comp}(\nu)$ and $t \in \mathit{Terms}(\mu)$ holds.*

[11] Recall that edges starting in nodes with more than one successor are labeled by "skip". Thus, we have (cf. [20]): $\forall\, n \in N.\ \ |\{\, e \mid \mathit{source}(e) = n \,\}| > 1 \Rightarrow \mathit{Terms}(\{\, e \mid \mathit{source}(e) = n \,\}) = \emptyset$. Hence, the truth value of the predicate $\mathit{Comp}(\nu)$ depends actually on a single flow-graph edge only.

2. (totally) *VFG*-redundant *at n, if t is partially VFG-redundant along each path $p \in \mathbf{P}_G[\mathbf{s}, n]$.*

4 The Main Process: Eliminating Semantic Redundancies

The nodes of a value-flow graph represent semantic equivalences of terms *syntactically*. In the main process of eliminating semantic redundancies, they play the same role as the lexical term patterns in the elimination of syntactic redundancies by syntactic CM. We demonstrate this analogy in the following section.

4.1 The Straightforward Approach

In this section we extend the analyses underlying the syntactic CM-procedure to the semantic situation in a straightforward fashion. As recalled above the point of the extension is that the value-flow graph nodes representing classes of transparently equivalent terms play the role the term patterns do in the syntactic case. We demonstrate this analogy by recalling the equation system characterizing up-safety in the syntactic setting first. Recall that in the syntactic setting up-safety of a term pattern t at a program point n means that t is computed on every program path reaching n without an intervening modification of some of its operands.

Equation System 4.1 (Syntactic Up-Safety for a Term Pattern t)

$$\texttt{Syn-US}(n) = \begin{cases} \mathit{false} & \text{if } n = \mathbf{s} \\ \prod_{m \in pred_G(n)} (\texttt{Syn-US}(m) \vee \mathit{Comp}(\,(m,n)\,) \wedge \mathit{Transp}(\,(m,n)\,)\,) & \text{otherwise} \end{cases}$$

The corresponding equation system for *VFG*-up-safety is given next. Note that there is no predicate like "VFG-Transp" corresponding to the predicate *Transp*. In the syntactic setting, the transparency predicate $\mathit{Transp}_t(e)$ is required for checking that the value of the term t under consideration is maintained along a flow-graph edge e, i.e., that none of its operands is modified. The essence of the value-flow graph is that transparency is modeled by the edges: two value-flow graph nodes are connected by a value-flow graph edge iff the value they represent is maintained along the corresponding flow-graph edge.

Equation System 4.2 (*VFG*-Up-Safety)

$$\texttt{VFG-US}(\nu) = \begin{cases} \mathit{false} & \text{if } \nu \in \mathit{VFN}_{\mathbf{s}} \\ \prod_{\mu \in pred_{VFG}(\nu)} (\texttt{VFG-US}(\mu) \vee \mathit{Comp}\,(\mu)) & \text{otherwise} \end{cases}$$

The roles of the start node and end node of the flow graph are here played by the "start nodes" and "end nodes" of the value-flow graph. They are defined by:

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

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

where $pred_{VFG}(\nu)$ and $succ_{VFG}(\nu)$ denote the predecessor and successor nodes of a value-flow graph node ν.

Down-safety is the dual counterpart of up-safety. However, the equation system for the *VFG*-version of this property is technically more complicated, as we have two graph structures, the flow graph and the value-flow graph, which must both be taken separately into account: as in the syntactic setting (or for up-safety), we need safety universally along all flow-graph edges, which is reflected by a value-flow graph node ν being down-safe at the program point $\mathcal{N}(\nu)$, if a term represented by ν is computed on all *flow-graph* edges leaving $\mathcal{N}(\nu)$ (i.e., $Comp\,(\nu)$),[11] or if it is down-safe at all successor points (i.e., $\prod_{m \in succ_G(\mathcal{N}(\nu))} \ldots$). However, we may justify safety along a flow-graph edge e by means of the *VFG* in *various* ways, as, in contrast to up-safety concerning the forward flow (or the syntactic setting), a value-flow graph node may have *several* successors corresponding to e (for illustration see the lower *VFG*-node in the left branch of Figure 8), and it suffices to have safety only along one of them. This is formally described by $\sum_{\substack{\mu \in succ_{VFG}(\nu) \\ \mathcal{N}(\mu) = m}} \texttt{VFG-MDS}(\mu)$.

Equation System 4.3 (*VFG*-Motion-Down-Safety)

$$\texttt{VFG-MDS}(\nu) \;=\; \begin{cases} \mathit{false} & \text{if } \nu \in VFN_{\mathbf{e}} \\ Comp\,(\nu) \vee \prod\limits_{m \in succ_G(\mathcal{N}(\nu))} \sum\limits_{\substack{\mu \in succ_{VFG}(\nu) \\ \mathcal{N}(\mu) = m}} \texttt{VFG-MDS}(\mu) & \text{otherwise} \end{cases}$$

The Transformation (of the Straightforward Approach)

Let $\texttt{VFG-US}^*$ and $\texttt{VFG-MDS}^*$ denote the greatest solutions of the Equation Systems 4.2 and 4.3. The analogy with the syntactic setting is continued by the specification of the semantic CM-transformation, called $Sem\text{-}CM_{Strght}$. It is essentially given by the set of insertion and replacement points. *Insertion* of an abstract value ν means to initialize a fresh temporary with a minimal representative t' of ν, i.e., containing a minimal number of operators, on the flow-graph edges leaving node $\mathcal{N}(\nu)$. *Replacement* means to replace an original computation by a reference to the temporary storing its value.

$$\begin{aligned} \texttt{Insert}(\nu) \;&\Longleftrightarrow\; \texttt{VFG-MDS}^*(\nu) \wedge \\ &\qquad \begin{cases} \mathit{true} & \text{if } \nu \in VFN_{\mathbf{s}} \\ \sum\limits_{\mu \in pred_{VFG}(\nu)} \neg(\texttt{VFG-MDS}^*(\mu) \vee \texttt{VFG-US}^*(\mu)) & \text{otherwise} \end{cases} \\ \texttt{Replace}(\nu) \;&\Longleftrightarrow\; Comp\,(\nu) \end{aligned}$$

Managing and reinitializing temporaries: In the syntactic setting, there is for every term pattern a unique temporary, and these temporaries are not interacting with each other: the values computed at their initialization sites are thus propagated to all their use sites, i.e., the program points containing an original occurrence of the corresponding term pattern, without requiring special care. In the semantic setting propagating these values requires usually to reset them at certain points to values of other temporaries as illustrated in Figure 2(b). The complete process including managing temporary names is accomplished by a straightforward analysis starting at the original computation sites and following the value-graph edges in the opposite direction to the insertion points. The details of this procedure are not recalled here as they are not essential for the point of this article. They can be found in [28, 29].

Second-Order Effects: In contrast to the syntactic setting, the straightforward semantic counterpart $\textit{Sem-CM}_{Strght}$ has second-order effects. This is illustrated in Figure 9: applying $\textit{Sem-CM}_{Strght}$ to the program of Figure 9(a) results in the program of Figure 9(b). Repeating the transformation again, results in the optimal program of Figure 9(c). This can easily be checked by means of Figure 8, which displays the corresponding value-flow graphs and the set of up-safe and down-safe nodes.

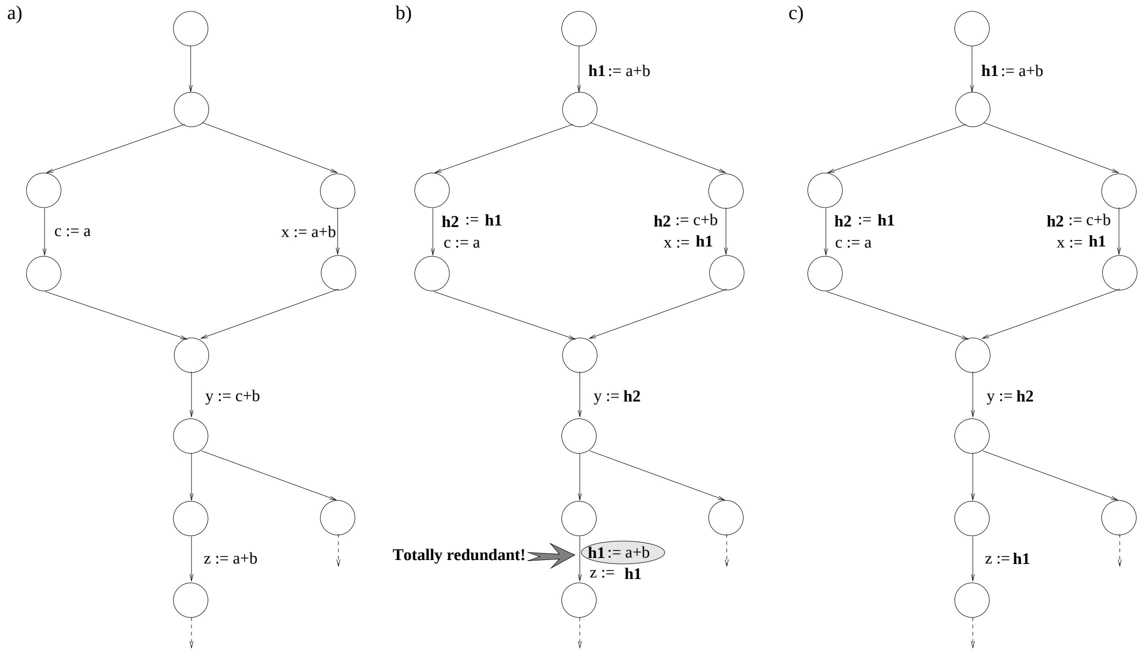

Figure 9: Illustrating Second-Order Effects

Intuitively, the reason that $\textit{Sem-CM}_{Strght}$ has second-order effects is that safety is no longer the sum of up-safety and down-safety. Above, up-safety is only computed with respect to original computations. While this is sufficient in the syntactic case, it is not in the semantic one as it does not take into account that the placement of computations as a result of the transformation may make "new" values available, e.g. in the program of Figure 9(b) the value of $a + b$ becomes available at the end of the program fragment displayed. As a consequence, total redundancies like the one in Figure 9(b) can remain in the program. Eliminating them (TRE) as illustrated in Figure 9(c) is sufficient to capture the second-order effects completely. We have:

Theorem 4.4 (*Sem-CM*$_{Strght}$)

1. *Sem-CM*$_{Strght}$ *relies on only* two uni-directional analyses.

2. *Sem-CM*$_{Strght}$ has second-order effects.

3. *Sem-CM*$_{Strght}$ *followed by TRE achieves* computationally motion-optimal results wrt the Herbrand interpretation *like its bidirectional precursor of [28].*

4.2 Avoiding Second-Order Effects: The Hierarchical Approach

The point of this section is to reestablish as a footing for the algorithm a notion of safety which can be characterized as the sum of up-safety and down-safety. The central result here is that a notion of safety tailored to capture the idea of "motion" (in contrast to "placement") can be characterized by down-safety together with the following hierarchical notion of up-safety:

Equation System 4.5 ("Hierarchical" *VFG*-Up-Safety)

$$\mathtt{VFG\text{-}US}_M(\nu) \;=\; \mathtt{VFG\text{-}MDS}^*(\nu) \vee \begin{cases} \mathit{false} & \text{if } \nu \in \mathit{VFN}_\mathbf{s} \\ \prod\limits_{\mu \in pred_{VFG}(\nu)} \mathtt{VFG\text{-}US}_M(\mu) & \text{otherwise} \end{cases}$$

In fact, the phenomenon of second-order effects can now completely and elegantly be overcome by the following hierarchical procedure:[12]

1. Compute down-safety (cf. Equation System 4.3).

2. Compute the modified up-safety property on the basis of the down-safety computation (cf. Equation System 4.5).

The transformation *Sem-CM*$_{Hier}$ of the Hierarchical Approach is now defined as before except that $\mathtt{VFG\text{-}US}_M{}^*$ is used instead of $\mathtt{VFG\text{-}US}^*$. Its results coincide with those of an exhaustive application of the CM-procedure proposed in Subsection 4.1, as well as of the original (bidirectional) CM-procedure of [28]. However, in contrast to the latter algorithm, which due to its bidirectionality required a complicated correctness argument, the transformation of the hierarchical approach allows a rather straightforward link to the Herbrand semantics, which drastically simplifies the correctness proof. Besides this conceptual improvement, the new algorithm is also easier to comprehend and to implement as it does not require any bidirectional analysis. We have:

Theorem 4.6 (*Sem-CM*$_{Hier}$)

1. *Sem-CM*$_{Hier}$ *relies on only* two uni-directional analyses sequentially ordered.

2. *Sem-CM*$_{Hier}$ *is* free of second-order effects.

3. *Sem-CM*$_{Hier}$ *achieves* computationally motion-optimal results wrt the Herbrand interpretation *like its bidirectional precursor of [28].*

---

[12]The "down-safety/earliest" characterization of [18] was also already hierarchical. However, in the syntactic setting this is not relevant as it was shown in [19].

5 Semantic Code Placement

In this section we are concerned with crossing the frontier marked by semantic CM towards to semantic CP, and giving theoretic and practic reasons for the fact that no algorithm has gone beyond this frontier so far. On the theoretical side (I), we prove that computational optimality is impossible for semantic CP in general. On the practical side (II), we demonstrate that down-safety, the handle for correctly and profitably placing computations by syntactic and semantic CM, is inadequate for semantic CP.

(I) No Optimality in General: Semantic CP cannot be done "computationally placement-optimal" in general. This is a significant difference to syntactic and semantic CM, which have a least element with respect to the relation "computationally better" (cf. [19, 28]). In semantic CP, however, we are faced with the phenomenon of incomparable minima. This is illustrated in the example of Figure 10 showing a slight modification of the program of Figure 4. Both programs of Figure 10 are of incomparable quality, since the "right-most" path is improved by impairing the "left-most" one.

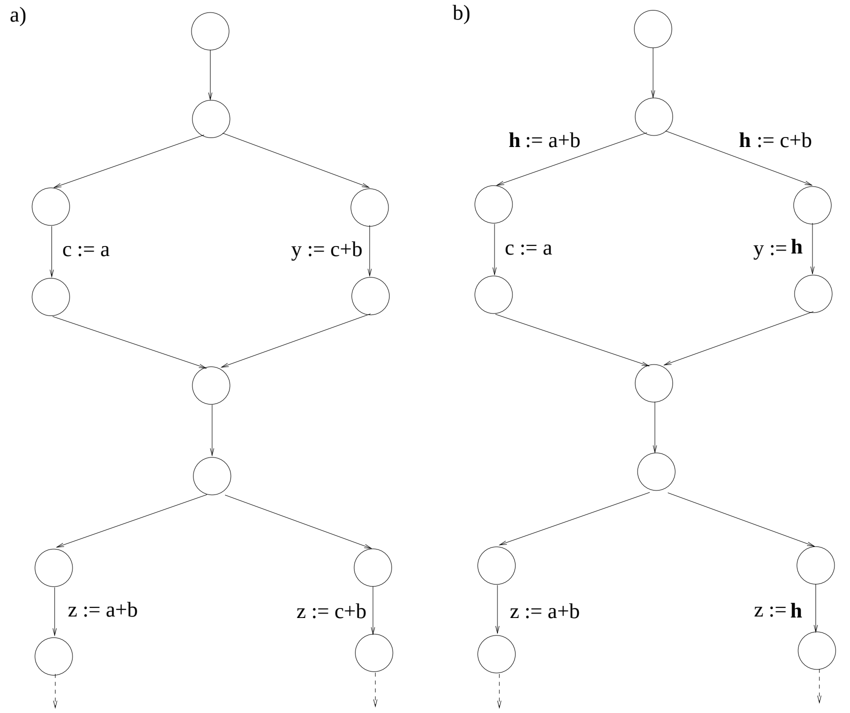

Figure 10: No Optimality in General: An Incomparable Result due to CP

(II) Inadequateness of Down-safety: For syntactic and semantic CM, down-safe program points are always legal insertion points. Inserting a computation at a down-safe point guarantees that it can be used on every program continuation. This, however, does not hold for semantic CP. Before showing this in detail, we illustrate the difference between "placable" *VFG*-down-safety (*VFG-DownSafe*) and "movable" *VFG*-down-safety (*VFG-M-DownSafe*) by means of the value-flow graph corresponding to the example of Figure 4.

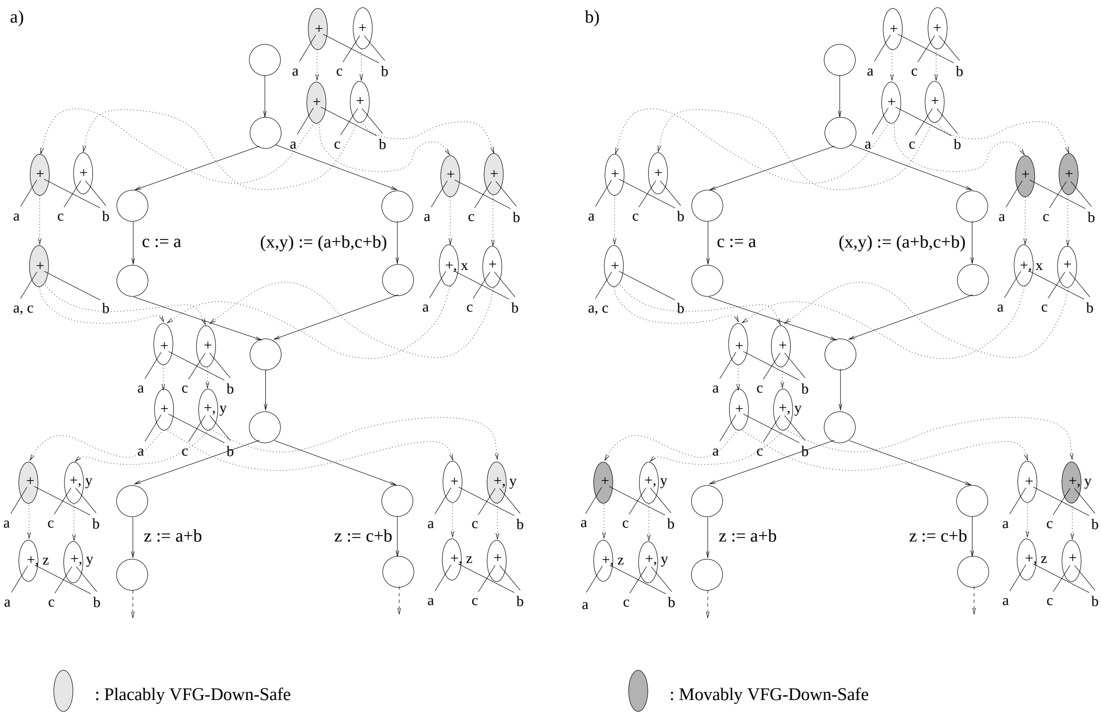

Figure 11: The Corresponding Value-Flow Graph

We remark that the predicate *VFG-DownSafe* is decidable. However, the point here is to demonstrate that this property is insufficient anyhow in order to arrive (straightforwardly) by an algorithm for semantic CP. This is illustrated by Figure 12 showing a slight modification of the program of Figure 10. Though the placement of $\mathbf{h} := a + b$ is perfectly down-safe, it cannot be used at all. Thus, impairing the program.

Summary: The examples of Figure 4 and of this section commonly share that they are invariant under semantic CM, since $a + b$ cannot safely be moved to (and hence not across) the join node in the mid part. However, $a + b$ can safely be placed in the left branch of the upper branch statement. In the example of Figure 4 this suffices to show that semantic CP is in general strictly more powerful than semantic CM. On the other hand, Figure 10 demonstrates that "computational optimality" for semantic CP is impossible in general. While this rules out the possibility of an algorithm for semantic CP being uniformly superior to every other semantic CP-algorithm, the inadequateness of down-safety revealed by the second example of this section gives reason for the lack even of heuristics for semantic CP: this is because down-safety, the magic wand of syntactic and semantic CM, loses its magic for semantic CP. In fact, naive adoptions of the semantic CM-procedure to semantic CP would be burdened with the placing anomalies of Figures 10 und 12. We conjecture that a satisfactory solution to these problems requires structural changes of the argument program. Summarizing, we have:

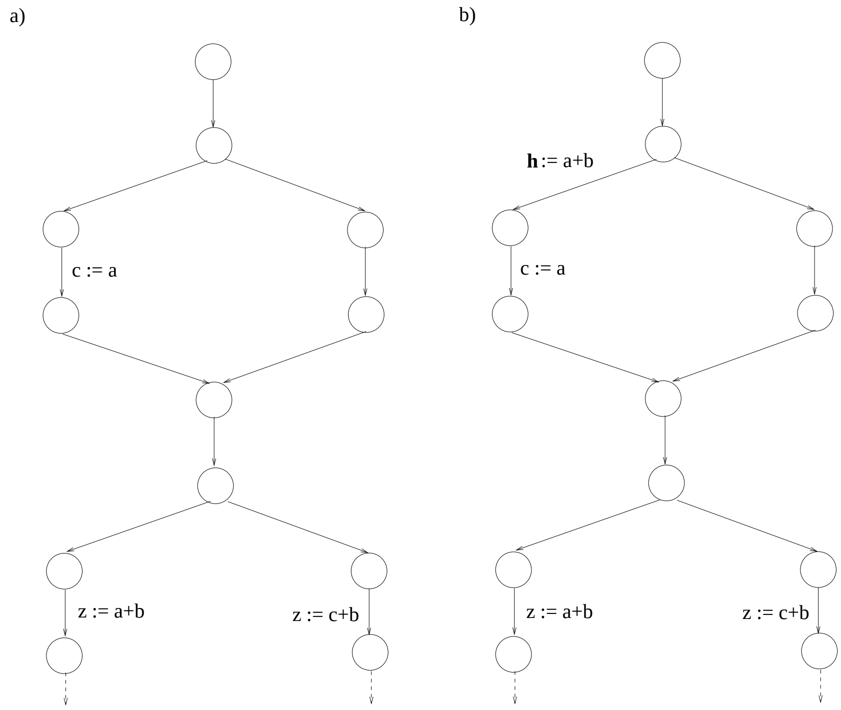

Figure 12: Inadequateness of Down-Safety: Degradation through Naive CP

Theorem 5.1 (Semantic Code Placement)

1. *Semantic CP is strictly more powerful than semantic CM.*

2. *Computational placement-optimality is* impossible *in general.*

3. *Down-safety is* inadeqate *for semantic CP.*

6 Conclusion

We have re-investigated semantic CM under the perspective of the recent development of syntactic CM, which has clarified the essential difference between the syntactic and the semantic approach, and uncovered the difference of CM and CP in the semantic setting. Central for the understanding of this difference is the role of the notion of *safety* of a program point for some computation. Modification of the considered notion of safety is the key for obtaining a transfer of the syntactic algorithm to the semantic setting which captures the full potential of motion algorithms. However, in contrast to the syntactic setting, motion algorithms do not capture the full potential of code placement. Actually, we conjecture that there does not exist a satisfactory solution to the code placement problem, unless one is prepared to change the structure of the argument program.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the 15th Annual Symposium on Principles of Programming Languages* (*POPL'88*) (*San Diego, California*), pages 1 – 11. ACM, New York, 1988.

[2] F. Chow. *A Portable Machine Independent Optimizer – Design and Measurements*. PhD thesis, Stanford University, Dept. of Electrical Engineering, Stanford, California, 1983. Published as Tech. Rep. 83-254, Computer Systems Lab., Stanford University.

[3] C. Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'95*) (*La Jolla, California*), volume *30*,6 of *ACM SIGPLAN Notices*, pages 246–257, 1995.

[4] J. Cocke and J. T. Schwartz. Programming languages and their compilers. Courant Institute of Mathematical Sciences, New York, 1970.

[5] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the 13th Annual Symposium on Principles of Programming Languages* (*POPL'86*) (*St. Petersburg Beach, Florida*), pages 70 – 85. ACM, New York, 1986.

[6] D. M. Dhamdhere. Characterization of program loops in code optimization. *Journal of Computer Languages*, 8(2):69 – 76, 1983.

[7] D. M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices*, 23(10):172 – 180, 1988.

[8] D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *International Journal of Computer Mathematics*, 27:1 – 14 (+ 31 – 32), 1989.

[9] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291 – 294, 1991. Technical Correspondence.

[10] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, California*), volume *27*,7 of *ACM SIGPLAN Notices*, pages 212 – 223, 1992.

[11] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635 – 640, 1988. Technical Correspondence.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319 – 349, 1987.

[13] A. Fong, J. B. Kam, and J. D. Ullman. Application of lattice algebra to loop optimization. In *Conference Record of the 2nd Annual Symposium on Principles of Programming Languages* (*POPL'75*) (*Palo Alto, California*), pages 1 – 9. ACM, New York, 1975.

[14] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization – Part I. *International Journal of Computer Mathematics*, 11:21 – 41, 1982.

[15] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization – Part II. *International Journal of Computer Mathematics*, 11:111 – 126, 1982.

[16] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the 1st Annual Symposium on Principles of Programming Languages* (*POPL'73*) (*Boston, Massachusetts*), pages 194 – 206. ACM, New York, 1973.

[17] J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? Technical Report MIP–9715, Fakultät für Mathematik und Informatik, Universität Passau, Germany, 1997.

[18] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, California*), volume *27*,7 of *ACM SIGPLAN Notices*, pages 224 – 234, 1992.

[19] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[20] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Bitvector analyses $\rightarrow$ No state explosion! In *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'95*) (*Aarhus, Denmark*), Lecture Notes in Computer Science, vol. 1019, pages 264 – 289. Springer-Verlag, Heidelberg, Germany, 1995.

[21] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96 – 103, 1979.

[22] J. H. Reif and R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the 4th Annual Symposium on Principles of Programming Languages* (*POPL'77*) (*Los Angeles, California*), pages 104 – 118. ACM, New York, 1977.

[23] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal on Computing*, 11:81 –93, 1981.

[24] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the 15th Annual Symposium on Principles of Programming Languages* (*POPL'88*) (*San Diego, California*), pages 2 – 27. ACM, New York, 1988.

[25] A. Sorkin. Some comments on a solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 11(4):666 – 668, 1989. Technical Correspondence.

[26] B. Steffen. Optimal run time optimization - Proved by a new look at abstract interpretations. In *Proceedings of the 2nd International Joint Conference on Theory and Practice of Software Development* (*TAPSOFT'87*) (*Pisa, Italy*), Lecture Notes in Computer Science, vol. 249, pages 52 – 68. Springer-Verlag, Heidelberg, Germany, 1987.

[27] B. Steffen. Property-oriented expansion. In *Proceedings of the 3rd Static Analysis Symposium* (*SAS'96*) (*Aachen, Germany*), Lecture Notes in Computer Science, vol. 1145, pages 22 – 41. Springer-Verlag, Heidelberg, Germany, 1996.

[28] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proceedings of the 3rd European Symposium on Programming* (*ESOP'90*) (*Copenhagen, Denmark*), Lecture Notes in Computer Science, vol. 432, pages 389 – 405. Springer-Verlag, Heidelberg, Germany, 1990.

[29] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development* (*TAPSOFT'91*) (*Brighton, United Kingdom*), Lecture Notes in Computer Science, vol. 494, pages 394 – 415. Springer-Verlag, Heidelberg, Germany, 1991.