Efficient Code Motion and an Adaption to Strength Reduction

Bernhard Steffen * Jens Knoop † Oliver Rüthing †

1 Introduction

Common subexpression elimination, partial redundancy elimination and loop invariant code motion, are all instances of the same general run-time optimization problem: how to optimally place computations within a program. In [SKR1] we presented a modular algorithm for this problem, which optimally moves computations within programs wrt *Herbrand equivalence*. In this paper we consider two elaborations of this algorithm, which are dealt with in Part I and Part II, respectively.

Part I deals with the problem that the full variant of the algorithm of [SKR1] may excessively introduce trivial redefinitions of registers in order to cover a single computation. Rosen, Wegman and Zadeck avoided such a too excessive introduction of trivial redefinitions by means of some practically oriented restrictions, and they proposed an efficient algorithm, which optimally moves the computations of acyclic flow graphs under these additional constraints (the algorithm is "*RWZ*-optimal" for acyclic flow graphs) [RWZ]. Here we adapt our algorithm to this notion of optimality. The result is a modular and efficient algorithm, which avoids a too excessive introduction of trivial redefinitions along the lines of [RWZ], and is *RWZ*-optimal for *arbitrary* flow graphs.

Part II modularly extends the algorithm of [SKR1] in order to additionally cover strength reduction. This extension generalizes and improves all classical techniques for strength reduction in that it overcomes their structural restrictions concerning admissible program structures (e.g. previously determined loops) and admissible term structures (e.g. terms built of induction variables and region constants). Additionally, the program transformation obtained by our algorithm is guaranteed to be safe and to improve run-time efficiency. Both properties are not guaranteed by previous techniques.

Structure of the Paper

After the preliminary definitions in Section 2, the paper splits into the following two parts, one for efficient code motion and one for strength reduction.

Part I starts with a motivation of our approach to efficient code motion in Section 3. Afterwards, we follow the structure of our code motion algorithm in Sections 4 and 5. Section 6 sketches the remaining part of the algorithm, which has already been presented in detail in [SKR1], and states our optimality result. Subsequently, Section 7 shows the complexity analysis of our algorithm.

Part II starts with a motivation of our extension to strength reduction in Section 8. This extension is illustrated by means of a small example in Section 9. Afterwards, we follow the structure of our strength reduction algorithm in Sections 10 and 11 and their subsections. Subsequently, we discuss the relationship to other approaches to strength reduction in Section 12.

Finally, Section 13 contains our conclusions.

*Lehrstuhl für Informatik II, Rheinisch-Westfälische Technische Hochschule Aachen, D-5100 Aachen

†Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, D-2300 Kiel 1 – The authors are supported by the Deutsche Forschungsgemeinschaft grant La 426/9-2

2 Preliminaries

This section contains the preliminary definitions for both parts. It is recommended to read the motivating sections of these parts first.

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

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

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. Nodes $n \in N$ represent *parallel assignments* of the form $(x_1, ., x_r) := (t_1, ., t_r)$, 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 $\mathbf{e}$.

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

$\delta_n : \mathbf{T} \to \mathbf{T}$ by $\delta_n(t) =_{df} t[t_1, ., t_r / x_1, ., x_r]$ 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 $j \in \{1, ., q-1\}$. $\mathbf{P}[m, n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}[m, n)$ the set of all finite paths from m to a predecessor of n. Additionally, ";" denotes 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 $p = (m \equiv n_1, \ldots, n_q \equiv n) \in \mathbf{P}[m, n]$, we define $\Delta_p : \mathbf{T} \to \mathbf{T}$ by

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

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

$$\Theta_p =_{df} \begin{cases} \theta_{n_1} & \text{if } q = 1 \\ \Theta_{(n_2, .., n_q)} \circ \theta_{n_1} & \text{otherwise} \end{cases}$$

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

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

Now, we can define:

Definition 2.1 (Herbrand Equivalence)
Let $t_1, t_2 \in \mathbf{T}$ *and* $n \in N$. *Then* t_1 *and* t_2 *are* Herbrand equivalent *at node* n *iff*

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

In order to deal with redundant computations this notion of equivalence must be generalized in order to cover terms occurring at different nodes.

Definition 2.2 (Partial Herbrand Redundancy)
Let $t_1, t_2 \in \mathbf{T}$, $m, n \in N$, $p_1 \in \mathbf{P}[\mathbf{s}, m)$, $p_2 \in \mathbf{P}[m, n)$, *and* $p =_{df} p_1; p_2$. *Then a computation of* t_1 *at* m *is* p-equivalent *to a computation of* t_2 *at* n *iff* $\mathbf{H}(t_1)(\Theta_{p_1}(\sigma_0)) = \mathbf{H}(t_2)(\Theta_p(\sigma_0))$. *A computation of* t_2 *at* n *is* partially Herbrand redundant *wrt a computation of* t_1 *at* m *iff there is a path* $p'' \in \mathbf{P}[m, n)$ *such that for all paths* $p' \in \mathbf{P}[\mathbf{s}, m)$ *the computations* t_1 *and* t_2 *are* $p'; p''$*-equivalent. In this case,* t_1 *and* t_2 *are also called* globally Herbrand equivalent[1].

We conclude this section with a technicality, which, however, is typical for code motion (cf. [RWZ]). Given an arbitrary flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, edges of G, leading from a node with more than one successor to a node with more than one predecessor are *critical*, since they may cause a "deadlock" during the code motion process, as can be seen in Figure 2.3(a):

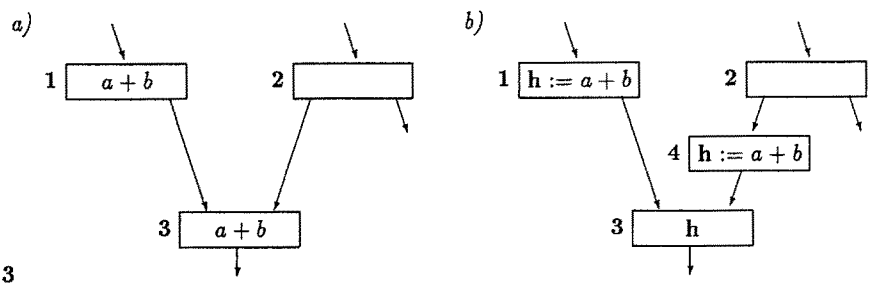

Figure 2.3

Here 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 its preceding nodes, 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 after the insertion of a synthetic node in the critical edge, as illustrated in Figure 2.3(b). We therefore assume that in the flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, which we consider as to be given for the formal development in this paper, a synthetic node has been inserted into every edge leaving a node with more than one successor. This certainly implies that all critical edges are eliminated. Moreover, it simplifies the analysis of the placement process, because one can now prove that all computation points are synthetic nodes, where it does not matter whether the initializations are inserted at the beginning or at the end (cf. Section 11).

[1]Note, global Herbrand equivalence is in general not an equivalence relation.

Part I: Efficient Code Motion[2]

3 Motivation

Common subexpression elimination ([Ki1, Ki2]), partial redundancy elimination ([MR, Ki1, Ki2]) and loop invariant code motion ([FKU]) are all instances of the same general run-time optimization problem: how to optimally place computations within a program. This can be illustrated by the following example:

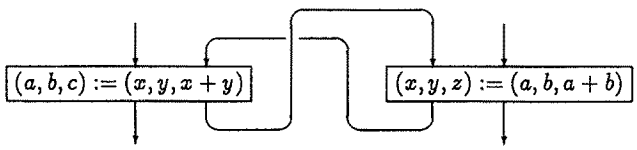

Figure 3.1

Here, the computation of "$x+y$" in the left hand block is *globally equivalent* to the computation of "$a+b$" in the right hand block. This justifies a placement of the computations, as it is shown below:

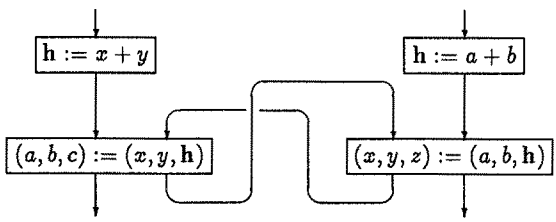

Figure 3.2

Two properties of this optimization are exceptional:

- It deals with arbitrary loop structures: note, the fragment above is not even reducible.
- It requires interrelated initialization statements that use syntactically different terms.

Two algorithms have been proposed to deal with code placement on this level of generality, which both abstract from costs of *trivial redefinitions*[3] as it is usual for code motion. First, an algorithm (cf. [SKR1]) that optimally moves computations wrt *Herbrand equivalence*[4]. However, this algorithm may introduce an arbitrary number of trivial redefinitions, just in order to cover a single computation of the program being optimized. Second, a more practically oriented algorithm (cf. [RWZ]), which is tailored to deal with a modified notion of optimality that we call *RWZ*-optimality. This algorithm avoids a too excessive introduction of trivial redefinitions by means of some practically oriented restrictions. However, it is structurally restricted: it is constrained to reducible flow graphs[5], and it is *RWZ*-optimal only for loop-free programs, i.e. it misses important optimizations in loop contexts, like for example the one presented above[6].

In this paper we will present a modification of the algorithm of [SKR1], which avoids a too excessive introduction of trivial redefinitions in the same way as the algorithm of [RWZ] does, but which is *RWZ*-optimal for arbitrary flow structures. Moreover, the algorithm is efficient, cleanly structured, and it allows a modular extension of its analysis and transformation power.

Essentially, this algorithm is obtained by splitting and reorganizing the first stage of the algorithm presented in [SKR1], which results in a three stage structure. As before, the algorithm depends on the *Value Flow Graph*, which serves as an interface between the second and third stage:

[2] [SKR2] is an extended version of Part I.

[3] A redefinition is called *trivial* if it is of the form $a := b$, where a and b are both variables (cf. [RWZ]).

[4] Herbrand equivalence is called transparent equivalence in [RWZ].

[5] We were told that their algorithm can be modified to overcome this constraint.

[6] A detailed illustration of the introductory example is given in [SKR2].

1. Determination of relevant terms: this step computes for every program point a finite set of "relevant" terms, which is sufficiently large in order to guarantee *RWZ*-optimality of our algorithm. (Section 4).

2. Computation of term equivalences:

   (i) Local equivalences: determining at every program point the Herbrand equivalence class for every relevant term by means of a modification of Kildall's algorithm (Section 5.1).

   (ii) Global equivalences: globalizing the local equivalence information determined in the previous step to an explicit representation of global term equivalences by constructing the *value flow graph* (Section 5.2).

3. *RWZ*-optimal placement of computations (Section 6):

   (i) Determining the optimal computation points by means of a modification of Morel/-Renvoise's algorithm. Our modification works on value flow graphs, which explicitly incorporate global equivalence information. This allows us to generalize Morel/Renvoise's technique, which only deals with term identity, to work for term equivalence (e.g. Herbrand equivalence).

   (ii) Placing the computations.

The worst case time complexity of our algorithm is limited by $O(n^4)$, where n is the number of nodes in the flow graph being optimized. This complexity is given by the Kildall-like step 2(i) of our algorithm, which is well-behaved in practice and therefore accepted for practical use. The other steps are of third order. In comparison, the worst case time complexity of the structurally restricted algorithm of [RWZ] is $O(n^3)$. Thus, except for its standard Kildall-like part, our algorithm is of the same worst case time complexity as the one proposed by Rosen, Wegman and Zadeck. As usual these estimations are based on the assumption of *constant branching* and *constant term depth*, i.e. on the assumption that the maximal number of successors of a node and the maximal depth of a program term is bounded by a constant.

Experience with an implementation of our algorithm, done in a joint project with the Norsk Data company, shows its practicality. In particular, all examples given are computed by means of this implementation.

An interesting feature of program transformations are their *second order effects*. Consider for example:

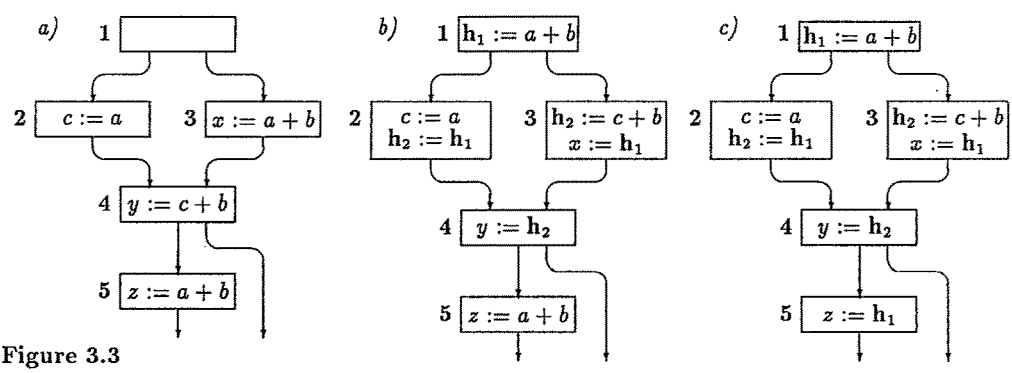

Figure 3.3

Here the computation of "$a + b$" in node 5 of Figure 3.3(a) cannot safely be moved to node 4, because this may introduce a new computation on the path leaving node 4 on the right branch.

However, after the program transformation displayed in Figure 3.3(b), which simultaneously moves the computations of "$c+b$" at node **4** and of "$a+b$" at node **3** to node **1**, the computation of "$a+b$" at node **5** can be replaced by a reference to the auxiliary variable $\mathbf{h}_1$ as it is illustrated in Figure 3.3(c).

Steffen [St] and Rosen, Wegman and Zadeck [RWZ] were the first who proposed algorithms dealing with such effects. Our algorithm here captures all second order effects in the sense of [RWZ].

4 Computation of Relevant Terms

The first stage of our algorithm computes for every program point a finite set of *relevant* terms. Essentially, a term t is relevant at a program point if its value *must* be computed on every continuation of a program execution passing this point. The equivalences wrt these terms are already sufficient for our placement procedure to determine a superset of the optimal computation points (cf. [SKR2]). However, the transformed program may still contain some full redundancies. This can be illustrated by means of Figure 3.3(a). The flow graph there would only be transformed into the one of Figure 3.3(b), missing to eliminate the redundancy of "$a+b$" at node **5**. In order to capture these redundancies as well, we subsequently enlarge the term closures mentioned above by means of a procedure which resembles the *question propagation* process of [RWZ].

This combined closure[7] guarantees that the placement procedure results in a *RWZ*-optimal flow graph. Moreover, it can be shown that the resulting placement is at least as good as a placement obtained by the techniques of Rosen, Wegman and Zadeck [RWZ], because the second closure step covers their *question propagation* completely.

5 Computation of Global Term Equivalences

5.1 Determining Local Term Equivalences

The semantic analysis of the first step of the second stage determines for every program point all Herbrand equivalence classes that contain (at least) one of the terms that have been associated with this point in the first stage (1. Optimality Theorem 5.5). Here, term equivalences are expressed by means of structured partition DAGs (cp. [FKU]). -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 5.1 *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* $r(\gamma)$.
- $\forall \gamma, \gamma' \in N_D.\ L_D(\gamma) \cap L_D(\gamma') \cap \mathbf{Op} \neq \emptyset \ \wedge\ l(\gamma)=l(\gamma') \ \wedge\ r(\gamma)=r(\gamma') \Rightarrow \gamma=\gamma'$.

Additionally, D *is called a* minimal structured partition DAG, *if all its root nodes satisfy* $|L_D(\gamma)| \geq 2$, *and it is called a* finite structured partition DAG, *if* N_D *is finite. The set of all structured partition DAGs is denoted by* $\mathcal{PD}$.

[7] An algorithm for its construction is given in [SKR2].

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

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

Then given a structured partition DAG two terms are equivalent iff they are represented by the same node of the DAG. This can be illustrated as follows:

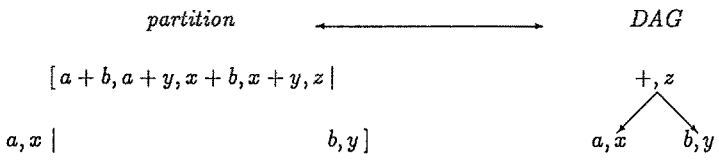

Figure 5.2

Thus a full DAG represents a partition (or equivalence relation) on:

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

Viewing DAGs as equivalence relations as it is suggested by Figure 5.2 makes the set of all structured partition DAGs a complete lattice, with inclusion defined set theoretically as usual. This guarantees the existence and well definedness of $\mathcal{H}(D)$ in:

Definition 5.3 *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.*

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:

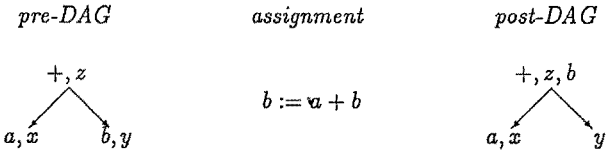

Figure 5.4

We have[8]:

Theorem 5.5 (1. Optimality Theorem)
Given an arbitrary flow graph, the analysis for determining local semantic equivalences[9] terminates with an annotation of finite structured partition DAGs, which syntactically characterize all Herbrand equivalence classes containing a relevant term.

Remark 5.6 Note that the corresponding algorithm of [SKR1], which determines a characterization of *all* Herbrand equivalence classes (rather than just the relevant ones), terminates with a different annotation. There we used *minimal* finite structured partition DAGs in order to reduce the complexity of the analysis. In fact, these DAGs provide the most concise DAG representation of Herbrand equivalence relations. In contrast, in this paper the complexity is limited by restricting the analysis to relevant equivalence classes (cf. Section 4). This restriction is essential for the complexity estimation in Section 7.

[8]The proof of this theorem is based on the Coincidence Theorem of [Ki2, KU].
[9]The corresponding algorithm is given in [SKR2].

5.2 The Value Flow Graph

The value flow graph (see Definition 5.7) represents global equivalence information explicitly. Essentially, its nodes represent term equivalence classes and its edges the data flow. For technical reasons, the nodes of a value flow graph are defined as pairs of equivalence classes. However, identifying these pairs with their second component leads back to the original intuition.

In the following let us assume that every node n of G is annotated by a pre-DAG $\mathbf{pre}(n)$ and a post-DAG $\mathbf{post}(n)$ according to the results of Section 5.1. For the sake of readability we abbreviate $\dot{\bigcup}_{n \in N}(N_{\mathbf{pre}(n)} \times N_{\mathbf{post}(n)})$ by Γ, and denote the flow graph node corresponding to a pair $(\gamma, \gamma') \in \Gamma$ by $\mathcal{N}(\gamma, \gamma')$. This allows to define the backward substitution relation $\xleftarrow{\delta} \subseteq \Gamma$ by:

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

where $(\gamma, \gamma') \in \xleftarrow{\delta}$ is abbreviated by $\gamma \xleftarrow{\delta} \gamma'$.

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 formal definition of the value flow graph for the DAG annotation under consideration is as follows (cf. [St, SKR1]):

Definition 5.7 *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*

$$\nu = (\gamma_1, \gamma_2) \in VFN \iff_{df} \begin{cases} \gamma_1 \xleftarrow{\delta} \gamma_2 & \text{if } \gamma_1 \neq \odot \wedge \gamma_2 \neq \odot \\ \dot{\exists} \gamma_3.\, \gamma_1 \xleftarrow{\delta} \gamma_3 & \text{if } \gamma_1 \neq \odot \wedge \gamma_2 = \odot \\ \dot{\exists} \gamma_3.\, \gamma_3 \xleftarrow{\delta} \gamma_2 & \text{if } \gamma_1 = \odot \wedge \gamma_2 \neq \odot \end{cases}$$

- *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 value, i.e. satisfy the inclusion $\mathbf{T}_{\mathbf{pre}(n)}(\gamma_1) \supseteq \{t \mid \exists t' \in \mathbf{T}_{\mathbf{post}(n)}(\gamma_2).\ t = \delta_n(t')\}$. 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. $\mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu'))}(\nu'\downarrow_1) \subseteq \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu))}(\nu\downarrow_2)$. Thus, edges of the value flow graph model the value flow along the branching structure of G and nodes the value flow over a single assignment statement. This is illustrated in Figure 5.8, which shows the important part of the value flow graph belonging to our introductory example.

Nodes of the value flow graph represent the value flow over the nodes of the flow graph: the term "x+y" ("a+b") which is represented by the first projection of the left (right) value flow graph node has the same value before the execution of the left (right) assignment as the terms which are represented by the second projection of the left (right) value flow graph node after the execution of this assignment.

Edges of the value flow graph represent the value flow along the edges of the flow graph: the terminal nodes of the two edges of the value flow graph below have first components "$\{x + y\}$" ("$\{a + b\}$"), which are contained in the second components of their initial nodes "$\{a + b, x + b, a + y, x + y, z\}$" ("$\{a + b, x + b, a + y, x + y, c\}$").

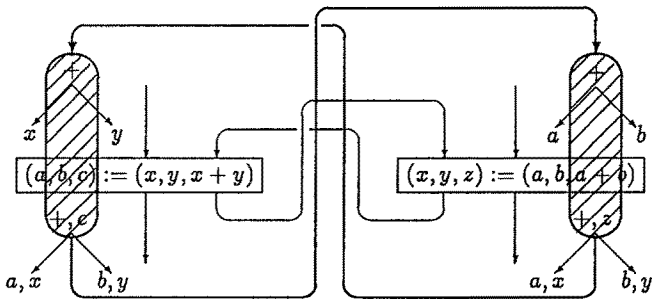

Figure 5.8

6 *RWZ*-Optimal Placement of Computations

The placement procedure of our three stage algorithm is exactly the same as the one introduced in [SKR1][10]. It places computations in a program relative to the equivalence information provided by a value flow graph. In this section we are going to show that the value flow graphs constructed in Section 5.2 lead to a placement satisfying an optimality criterion which was first considered in [RWZ][11].

Definition 6.1 *A flow graph G satisfies the RWZ-*criterion *iff every redundancy of a computation t_1 at a node w wrt a Herbrand equivalent computation t_2 at a node u on a path $p \in \mathbf{P}[u, w]$ is of one of the following two kinds:*

1. *path p goes through a node v and there exist two further paths: the first, p_1, from the start node through v to a predecessor of w along which no computation is performed that is p_1-equivalent to the computation of t_1 at w, and the second from v to the end node of G that does not contain a computation equivalent to that of t_1 at w,*

2. *path p goes through a node v and there exists another path from u to w through v on which the computation of t_1 at w is Herbrand equivalent to a computation of t_3 at v, and on neither path are the computations of t_3 at v and of t_2 at u Herbrand equivalent.*

The *RWZ*-criterion was introduced in [RWZ] in order to establish a notion of optimality for a placement procedure: a placement is "optimal" if the resulting program satisfies this criterion. Whereas the elimination of redundancies of the second kind may require an excessive introduction of trivial redefinitions, redundancies of the first kind cannot be eliminated without violating safety. However, there are programs, which cannot be improved by means of safe transformations and do not satisfy the *RWZ*-criterion:

[10] It can also be found in [SKR2].

[11] However, the definition of optimality there is erroneous. It does not cover the right intuitions, and in cases cannot be met without violating safety.

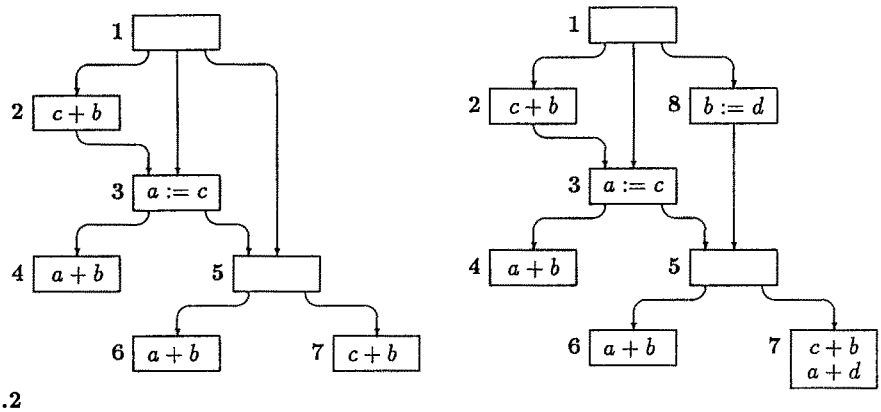

Figure 6.2

In both diagrams of Figure 6.2 the computation of "$a+b$" at node 4 is partially Herbrand redundant wrt the computation of "$c+b$" at node 2 and neither the first nor the second condition of Definition 6.1 holds. In fact, the second picture shows that the defect cannot be explained in terms of necessary computations on program paths: although the value of the computation "$a + b$" at node 4 is computed on every path through this program, no program path can safely be improved without impairing some other program paths. Thus, we need to consider a weaker notion of optimality, which we obtain by replacing the first condition of Definition 6.1 by the following:

- path p goes through a node v, and
  - there exists a further path p_1 from the start node through v to a predecessor of w along which no computation is performed that is p_1-equivalent to the computation of t_1 at w, and
  - a computation of the value of t_1 at v is not statically safe.

Intuitively, a computation t is *statically safe* at a node n, if every successor m of n satisfies:

- the value of t is computed at m or
- there is a term t', which is partially Herbrand redundant wrt t at n, whose computation at m is statically safe[12].

In fact, in contrast to the optimality result of [RWZ], the algorithm proposed there only satisfies this (meaningful) weaker notion of optimality on DAGs, which we call *RWZ*-optimality. In fact our algorithm satisfies this notion of optimality for arbitrary flow graphs.

Theorem 6.3 (*RWZ*-Optimality)
Every flow graph transformed by our three stage algorithm is RWZ-optimal.

7 Complexity

We estimate the worst case time complexity independently for every stage. As usual this estimation is based on the assumption of *constant branching* and *constant term depth*, and depends on 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. Note that $\mathtt{n} * \mu$ is an upper approximation of the number of nodes in the value flow graph, which we will abbreviate by ν. This

[12]Note, in both diagrams of Figure 6.2, a computation of "$a + b$" at node 3 is not statically safe, since its value is not computed at node 5 and neither a computation of "$a + b$" nor "$c + b$" is statically safe at node 5.

yields for the complexity of the five steps of our algorithm[13]:

1. Determination of relevant terms: $O(n^3)$. Using our assumption of constant branching and constant term depth, it can be shown that in the worst case the maximal number of terms a single flow graph node is annotated with is of order $O(n^2)$. Thus, the estimation by $O(n^3)$ is based on the very pessimistic assumption that this worst case occurs at every flow graph node. In practice, however, the set of relevant terms is much smaller. This should be kept in mind, because all the other estimations are based on this worst case assumption.

2. Computation of term equivalences:

   (i) Local equivalences: $O(n^2 * m)$. Here, "n^2" reflects the maximal length of a descending chain of annotations of a flow graph. In fact, the number of analysis steps to determine the local equivalences is linear in this chain length. This is achieved by adding those nodes to a workset whose annotations have been changed (rather than their successors). Then processing a worklist entry consists of updating the annotations of all its successors just wrt the change of annotation at the node being the entry. This can be done in $O(m)$ because of our assumption of constant branching.

   (ii) Global equivalences: $O(n * \mu)$. This estimation for the costs of constructing the value flow graph 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 $O(\mu)$.

3. Optimal placement of the computations:

   (i) Determination of the computation points: $O(\nu)$. The argument needed here is based on that of the first step, however, two additional problems arise. First, we do not have constant branching, and the algorithm here is bidirectional. Second, the predicates associated with a node contain a disjunction of properties of their successors[14]. However, using a "counted or" for this predicate, all nodes of the value flow graph can be updated once by executing only two constant time operations per edge of the value flow graph. Moreover, the number of edges of a value flow graph can be estimated by the number of its nodes $O(\nu)$ as well. Thus, the determination of the optimal computation points is linear in the number of nodes of the value flow graph.

   (ii) Placing the computations: $O(\nu)$. This is straightforward for our algorithm.

Using the fact that the maximal size of a set of relevant terms a single flow graph node is annotated with can be estimated by $O(n^2)$, we obtain that both m and μ can be approximated by $O(n^2)$ as well. While this is straightforward for the estimation of μ, the estimation for m exploits the fact that the meet of two structured partition DAGs can be computed essentially linearily in the size of the resulting DAG. This yields a worst case time complexity of $O(n^4)$ for the Kildall-like first step of the second stage of our algorithm, and of $O(n^3)$ for all other steps. Note that this estimation of the Kildall-like step is rendered possible only by its restriction to compute the Herbrand equivalence classes solely for relevant terms. However, even the standard approach, which we conjecture to be exponential in its worst case, is well-behaved in practice and therefore accepted for practical use.

Of independent interest is the estimation of the complexity of the third stage, yielding that the placement process is linear in the size of the value flow graph. The argumentation used here also applys to the classical algorithm of Morel and Renvoise [MR], showing that their algorithm is linear in the size of the flow graph. This improves all previous estimations we know of.

[13]The complete algorithms are given in [SKR2].

[14]See **PPOUT** in Equation System 11.1.

Part II: Strength Reduction[15]

8 Motivation

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

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

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

Note, if no updating assignments are inserted, this three step procedure performs *loop invariant code motion*. In fact, a clean realization of it should transform the flow graph of Figure 8.1(a) into the one displayed in Figure 8.1(b)[16]:

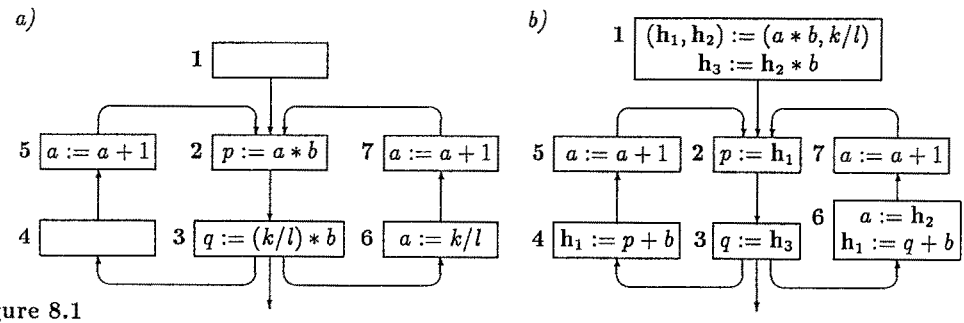

Figure 8.1

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

1. Construction of a value flow graph (Section 10):
   (i) Determining all Herbrand equivalences.
   (ii) Computing for every program point a finite set of "relevant" terms that allows to syntactically represent enough term equivalences in order to perform strength reduction.
   (iii) Constructing the corresponding value flow graph.
2. Placement of the computations:
   (i) Determining the computation points and computation forms wrt the value flow graph obtained in step 1(iii) (Section 11.1).
   (ii) Placing the computations (Section 11.2).

[15] [KS] is an extended version of Part II.

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

This algorithm performs strength reduction based on an optimal movement of the computations wrt Herbrand equivalence. The point of this approach is that it reduces strength reduction completely to the availability of values at the computation points. This allows to overcome all restrictions concerning admissible program structures (e.g. previously detected loops) and admissible term structures (e.g. terms built of induction variables and region constants) that are required by previous strength reduction techniques (cf. Section 12). Moreover, it is the key for proving that program transformations obtained by our algorithm are guaranteed to be *safe* and to *improve* run-time efficiency. Both properties can be violated by previous techniques (cf. Section 12). The power of our algorithm that generalizes and improves the classical algorithms for strength reduction, common subexpression elimination, partial redundancy elimination, and loop invariant code motion is illustrated in the example of Figure 8.1(a), where to the best of the authors' knowledge the algorithm presented here is unique in performing the optimization displayed in Figure 8.1(b).

9 Discussion of a Small Example

In this section we discuss the effects of the five steps of our two stage algorithm by means of the example of Figure 9.1(a), which will be transformed into the flow graph displayed in Figure 9.1(b):

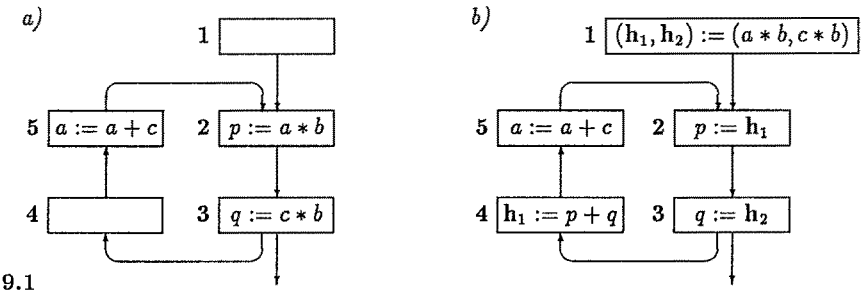

Figure 9.1

The semantic analysis of step 1(i) annotates the flow graph with partitions[17] that characterize all equivalences between terms wrt the Herbrand interpretation, i.e. all equivalences that are valid independently of specific properties of the term operators (Figure 9.2). In particular, this analysis detects the equivalence of p and $a * b$ and of q and $c * b$ after the execution of node **3** (cf. Section 10).

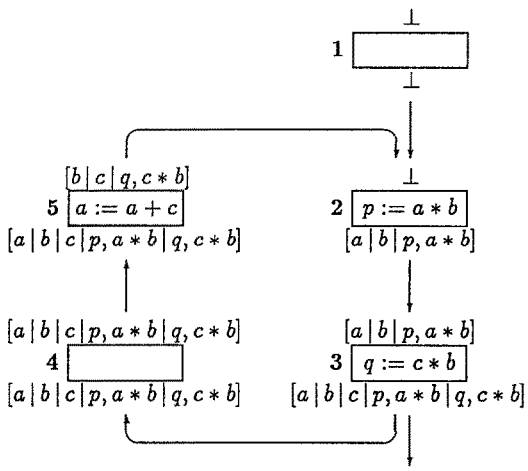

Figure 9.2

[17] Partitions are represented by means of structured partition DAGs (see Section 5.1 and 10).

Afterwards, step 1(ii) computes for every program point a finite set of "relevant" terms, which contains a representation system of those equivalence classes that express all necessary equivalences syntactically (cf. Section 10), and extends the node annotation computed in step 1(i) accordingly. This (straightforward) extension is necessary, because the placement process of the second stage only refers to term equivalences that are explicit in the value flow graph under consideration, i.e. two terms are equivalent at a program point if they are commonly represented by a node of the value flow graph at this point. In addition to the corresponding step of the algorithm of [SKR1], strength reduction requires to consider terms as relevant that arise from an application of arithmetic laws. The essence of classical strength reduction is to exploit the distributive law for sums and products: $(u+v)*w = u*w+v*w$. Therefore, whenever a term of the form $(u+v)*w$ is relevant, the terms $u*w$ and $v*w$ are also relevant[18].

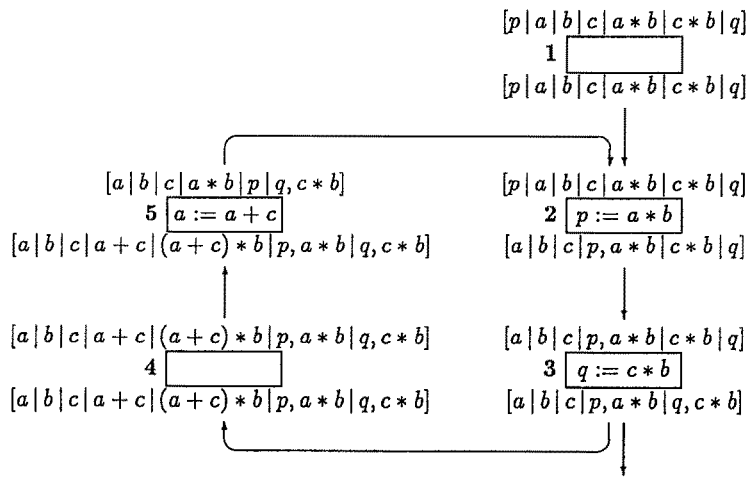

Figure 9.3

Step 1(iii) produces the corresponding value flow graph (cf. Section 10), whose relevant part is displayed in Figure 9.4:

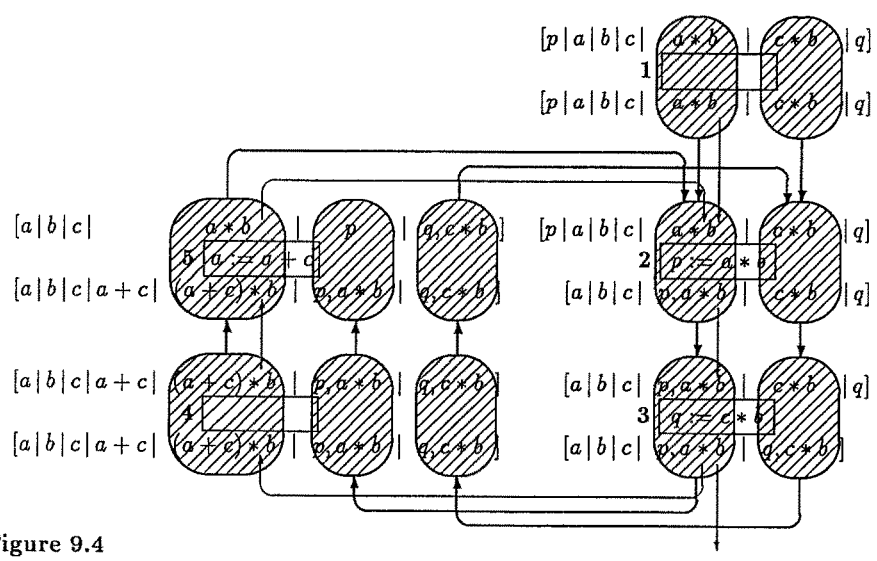

Figure 9.4

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

Applying a modification of Morel/Renvoise's algorithm (step 2(i), cf. Section 11.1) to the value flow graph above yields the computation points and computation forms. In addition to the corresponding step of the algorithm of [SKR1], the determination of computation forms here needs to exploit the distributive law in order to capture strength reduction. This is achieved by adding the predicate DISTR to the equation system (cf. Section 11.1). After this preparation, the placement procedure of step 2(ii) results in the following flow graph (cf. Section 11.2)[19]:

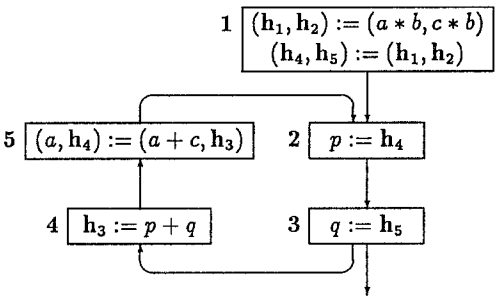

Figure 9.5

Subsequent *variable subsumption* [Ch, CACCHM] yields the desired result (Figure 9.1(b)).

10 Construction of a Value Flow Graph

In this section we follow [SKR1] in that we first compute all Herbrand equivalences and subsequently build an appropiate problem dependent term closure. This is in contrast to the approach of Part I, where the problem dependent term closure was computed first in order to gain efficiency.

1. Determining all Herbrand equivalences.
2. Computing for every program point a finite set of "relevant" terms that allows to syntactically represent enough Herbrand equivalences in order to perform strength reduction.
3. Constructing the corresponding value flow graph.

Since the procedures of the first and third step are essentially the same as the corresponding steps of Part I and [SKR1], we concentrate on the second step here[20]:

Computation of Relevant Terms

The placement process of our algorithm (Section 11.1) considers the pre-DAGs and post-DAGs of a flow graph annotation as purely syntactical objects, i.e. terms are considered equivalent iff they are syntactically equivalent (Definition 5.3(2)). Thus we need to extend the flow graph annotation constructed in the first step of the first stage, which characterizes Herbrand equivalence semantically (Definition 5.3(3)), to a sufficiently large syntactic representation. As in Section 4 and [SKR1], this is achieved by computing for every node n of G a finite set of *relevant* terms $T_{suf}(n)$ that contains a representative of all equivalence classes that are necessary at node n. However, in order to capture (classical) strength reduction, we additionally need to exploit algebraic laws. Remember, classical strength reduction essentially replaces computations of the form $u * (v + w)$ by $(u * v) + (u * w)$. This is safe and profitable, whenever the values of $u * v$ and $u * w$ are available. Therefore, we consider a term $(u * v) + (u * w)$ and its subterms as relevant here, whenever the term $u * (v + w)$ is

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

[20]Details can be found in [KS].

relevant in the sense of [SKR1]. Moreover, commutativity and associativity are necessary in order to evaluate subterms with constant operands, whose values can be computed already at compile time and therefore enlarge the number of available expressions.

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

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

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

Here we consider the basic strategy of [SKR1] for computing relevant terms, which determines for every program point the set of all terms whose value *must* be computed on every continuation of a program execution passing this point. Enhancing this strategy by means of the closure operator Φ it is already sufficient to uniformly capture the known strength reduction algorithms[21]. The complete closure algorithm can be found in [KS].

11 Placement of Computations

11.1 Determination of Computation Points and Forms

The determination of computation points is split into two steps. The first step coincides with the corresponding step of [SKR1]. It determines the computation points wrt the equivalence information that is expressed by the value flow graph under consideration. The second step, however, had to be extended. It determines the computation forms for the computation points computed in the first step. This has been trivial in [SKR1], where computation forms are simply minimal representatives of the Herbrand equivalence classes associated with the computation points. In the context of strength reduction, however, the choice of the computation forms is much more elaborate, because semantic equations need to be exploited to take care of replacing "expensive" by "cheap" operations (cf. Theorem 11.2).

Computation Points

The point of this step is the solution of the Boolean equation system 11.1, which was introduced in [SKR1]. It is tailored to work on value flow graphs rather than flow graphs directly, in order to capture semantic equivalence (cf. [SKR1] and Part I). Following [MR], the names of the predicates are acronyms for the properties *"local anticipability"*, *"availability"* and *"placement possible"*. Furthermore, the formal presentation of the equation system needs the following notation: given a value flow graph VFG, let

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

and

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

where $pred_{VFG}$ and $succ_{VFG}$ denote functions that map a node of VFG to its set of predecessors and successors, respectively. This allows:

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

Equation System 11.1 (Boolean Equation System)

- The Frame Conditions (Local Properties):

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

$$\mathbf{AVIN}(\nu) = \mathit{false} \text{ if } \nu \in \mathit{VFN}_{\mathbf{s}} \ \wedge \ \mathbf{T}_{\mathbf{pre}(\mathcal{N}(\nu))}(\nu\downarrow_1) \not\subseteq \mathbf{C}$$

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

- The Fixed Point Equations (Global Properties):

$$\begin{aligned}
\mathbf{AVIN}(\nu) &\Longleftrightarrow \prod_{\nu' \in \mathit{pred}(\nu)} \mathbf{AVOUT}(\nu') \\
\mathbf{AVOUT}(\nu) &\Longleftrightarrow \mathbf{AVIN}(\nu) \vee \mathbf{PPOUT}(\nu) \\
\mathbf{PPIN}(\nu) &\Longleftrightarrow \mathbf{AVIN}(\nu) \wedge (\mathbf{ANTLOC}(\nu) \vee \mathbf{PPOUT}(\nu)) \\
\mathbf{PPOUT}(\nu) &\Longleftrightarrow \prod_{m \in \mathit{succ}(\mathcal{N}(\nu))} \sum_{\substack{\nu' \in \mathit{succ}(\nu) \\ \mathcal{N}(\nu') = m}} \mathbf{PPIN}(\nu')
\end{aligned}$$

The greatest solution of this system[22] determines the computation points by means of

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

Computation Forms

In this step we determine for every value flow graph node ν satisfying the predicate **INSERT** an initialization term (computation form), i.e. a term with "minimal" executions costs that represents the value of the equivalence class $\nu\downarrow_2$. In the case of the Herbrand interpretation an initialization term is just a minimal representative of $\nu\downarrow_2$ (cf. [KS, SKR1, SKR2]). However, in order to capture the effects of strength reduction a more careful choice is necessary. We therefore introduce a new predicate **DISTR** (*"Distributivity"*) that establishes a relationship between candidates for strength reduction (given by terms of "$\nu_1\downarrow_2$" having "$*$" as top most operator) and values (given by terms of "$\nu_2\downarrow_2$" and "$\nu_3\downarrow_2$"), whose sum is equivalent to the value of the candidate:

$$\begin{aligned}
\mathbf{DISTR}(\nu_1, \nu_2, \nu_3) \Longleftrightarrow\ & \mathcal{N}(\nu_1) = \mathcal{N}(\nu_2) = \mathcal{N}(\nu_3) \ \wedge \\
& \mathbf{INSERT}(\nu_1) \wedge \mathbf{AVOUT}(\nu_2) \wedge \mathbf{AVOUT}(\nu_3) \ \wedge \\
& L_{\mathbf{post}(\mathcal{N}(\nu_1))}(\nu_1\downarrow_2) = \{*\} \ \wedge \\
& \exists\, t_2 \in \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu_2))}(\nu_2\downarrow_2)\ \exists\, t_3 \in \mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu_3))}(\nu_3\downarrow_2). \\
& \qquad (+, t_2, t_3) \in \Phi(\mathbf{T}_{\mathbf{post}(\mathcal{N}(\nu_1))}(\nu_1\downarrow_2))
\end{aligned}$$

For notational convenience we introduce the predicate **SRINS** which is derived from **DISTR**:

$$\mathbf{SRINS}(\nu) \Longleftrightarrow \exists\, \nu_1, \nu_2 \in \mathit{VFN}.\ \mathbf{DISTR}(\nu, \nu_1, \nu_2)$$

[22] An algorithm for determining this solution is given in [KS].

The intuitive meaning of the predicate **SRINS** (*"Strength Reduction Insertion"*) is the following: whenever a node ν of the value flow graph satisfies the predicate **SRINS**, then there exist two further nodes ν_1 and ν_2, which represent values, whose sum equals the value represented by $\nu\downarrow_2$. This allows us to choose as an initialization term a term having " + " as top most operator and initialization terms of ν_1 and ν_2 as operands, instead of the "standard" minimal representative of $\nu\downarrow_2$, which has " $*$ " as top most operator. Note, due to the availability of ν_1 and ν_2 this choice can be proved to be safe and to improve the efficiency, i.e. there is no path on which a new computation is introduced as a consequence of this replacement. In fact, we have:

Theorem 11.2 *The computation forms (initialization terms) are optimal wrt the convertibility relation and the local equivalence information expressed by the value flow graph under consideration.*

Every flow graph transformed by our two stage algorithm has the same computation points as the flow graph that results from the algorithm of [SKR1] applied to the same value flow graph. The transformed flow graphs differ only in the form and the computation costs of the initialization terms. This difference, which arises from the greater flexibility in the choice of the initialization terms here, leads to second order effects: replacing multiplications by summations according to the distributive law may introduce (partial) redundancies in the program. This is due to the fact, that the specific properties of " + " and " $*$ " are considered only by the second stage of the algorithm, but not during the semantic analysis of the first stage. Whereas a heuristic approach to this problem can be found in [KS], a systematic treatment is under investigation.

11.2 Placing the Computations

The placement procedure is a straightforward adaption of the placement procedure of [SKR1]. Essentially, it performs the following steps:

- Initializing auxiliary variables for every value flow graph node satisfying the predicate **INSERT** by means of an initialization term with minimal computation costs.
- Propagating the values of these auxiliary variables to the locations of original program terms and replacing them by references to their corresponding auxiliary variables.

The detailed placement procedure is given in [KS].

12 Related Work

Strength reduction was pioneered by Cocke and Kennedy [CK][23] and later on generalized and improved in particular by Allen, Cocke and Kennedy [ACK], and Joshi and Dhamdhere [JD1, JD2]. All these approaches, which characterize the state of the art, are:

- *Syntactic* : they optimize term by term, without exploiting semantic equivalences between syntactically different terms.
- *Locally updating*: they insert update assignments whenever an operand of a candidate expression for strength reduction is redefined, without investigating the global context for the necessity of this update. This may introduce terms, whose values are not computed in the original program. Thus, the resulting program transformation *cannot* be guaranteed to be *safe* or to *improve* run-time efficiency.

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

- *Structurally restricted*: [CK, ACK] work only for previously detected loops and terms built from induction variables and region constants, which excludes the optimization of more general program structures. In contrast, [JD1, JD2] work for arbitrary control flow structures and terms composed of variables and program constants. They pay for their ability to deal with general program structures by requiring an unnecessarily strong notion of admissible term structure.

In contrast, our algorithm is:

- *Semantic*: it exploits semantic equivalence between syntactically different terms.
- *Globally updating*: update assignments are only inserted, if they are required by the global context. This guarantees that the resulting program transformation is *safe* and that it *improves* the run-time efficiency of the original program.
- *General*: it works for arbitrary program structures without requiring additional constraints concerning admissible term structures[24].

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

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

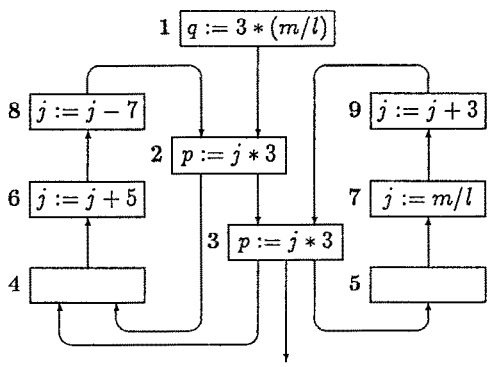

Figure 12.1

In the flow graph above the computation of $j * 3$ in node 2 and 3 is a candidate expression for strength reduction. Local updating means to insert for *every* redefinition of an operand of a candidate expression e a redefinition of the auxiliary variable $\mathbf{h}$ storing the value of e to preserve the value of e in $\mathbf{h}$. Therefore, the only nontrivial transformation a local updating algorithm can do to the flow graph above results in the flow graph shown in Figure 12.2(a)[25]. Note that local updating introduces a computation whose value is not computed in the original program, namely the value of the computation of $p + 15$ at node 6. Hence, the transformation is unsafe. Moreover, it even impairs the run-time efficiency: on path $(3, 5, 7, 9)$ one multiplication is saved, but a multiplication and an addition is inserted. And on path $(2, 4, 6, 8)$ a multiplication is saved on the costs of two

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

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

inserted additions, whose added computation costs may exceed those of the saved multiplication[26]. Our algorithm produces the flow graph of Figure 12.2(b).

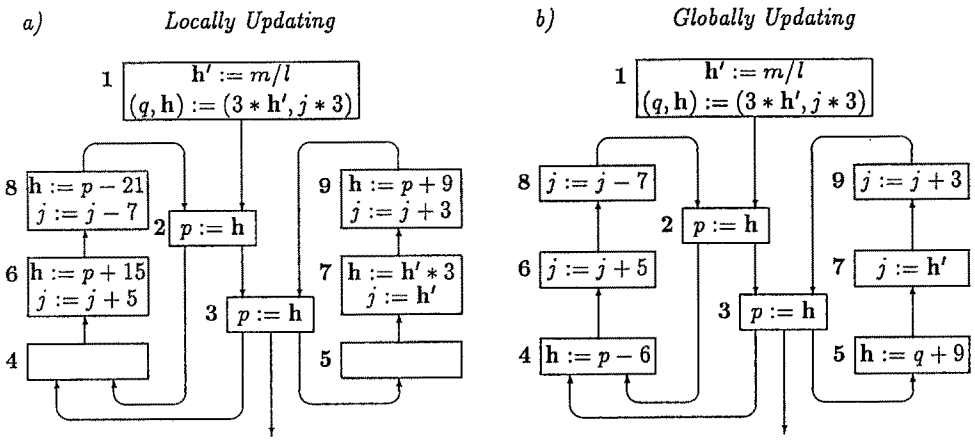

Figure 12.2

13 Conclusion

Based on the code motion algorithm of [SKR1], which optimally moves computations within programs wrt Herbrand equivalence, we developed two elaborations: first, an efficient algorithm for code motion that achieves the effect of the (in a sense optimal) algorithm of [RWZ] for arbitrary flow graphs, and second, a uniform extension to strength reduction.

The algorithm of [SKR1] may excessively introduce trivial redefinitions of variables in order to cover a single computation. This effect is limited along the lines of [RWZ] by the algorithm presented in Part I. The point of our algorithm is that it is *RWZ*-optimal without any restrictions on the flow structure of the flow graph being optimized, rather than just for DAGs, and that it is almost as efficient as the structurally restricted algorithm of [RWZ].

The algorithms of [SKR1] and Part I generalize and improve previous techniques for common subexpression elimination, partial redundancy elimination, and loop invariant code motion. In addition, the algorithm presented in Part II also improves on all classical techniques for strength reduction in that it overcomes their restrictions concerning admissible program structures (previously detected loops) and admissible term structures (built of induction variables and program constants).

The development of both algorithms profited from the modular structure of the underlying code motion algorithm (cf. [SKR1]). This modularity, which is due to the strict separation of the local and global equivalence analysis, the computation of relevant terms, and the placement procedure, has been maintained. Thus further extensions are supported. For example, both algorithms can be extended to cover further optimization goals like *constant propagation* and *constant folding* ([SK]) by strengthening the capacity of determining local equivalences between terms.

[26]The algorithm of [JD2] deals with these problems using a machine dependent heuristic: assumed that the computation costs of two additions are less expensive than those of a multiplication, it would insert within the loop the assignments $\mathbf{h} := j * 3$ on the edge leaving node **9**, $\mathbf{h} := \mathbf{h} - 21$ at node **8**, and $\mathbf{h} := \mathbf{h} + 15$ at node **6**. This transformation would improve the "left" part of the loop construct. However, without this assumption it would insert the assignment $\mathbf{h} := j * 3$ at node 4 instead of the insertions at node **6** and **8**. And in this case, there would be no positive effect on the run-time efficiency at all.

Acknowledgements

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

References

[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, 1981

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

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

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

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

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

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

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

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

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

[KS] J. Knoop and B. Steffen. *"Strength Reduction based on Code Motion: A Uniform Approach"*. Extended version of Part II of this paper. Bericht Nr. 9103, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 1991

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

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

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

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

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

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

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

[SK] 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 – An extended version is to appear in TCS 80(1), April 1991

[SKR1] B. Steffen, J. Knoop and O. Rüthing. *"The Value Flow Graph: A Program Representation for Optimal Program Transformations"*. 3^{rd} ESOP, Copenhagen, Denmark, LNCS 432, 389 - 405, 1990 – Extended version available as: Bericht Nr. 9004, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 1990

[SKR2] B. Steffen, J. Knoop and O. Rüthing. *"Optimal Code Motion within Flow Graphs: A Practical Approach"*. Extended version of Part I of this paper. Bericht Nr. 9102, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 1991