Expansion-Based Removal of Semantic Partial Redundancies

Jens Knoop, Oliver Rüthing, and Bernhard Steffen

Universität Dortmund
Baroper Str. 301, D-44221 Dortmund, Germany
{knoop,ruething,steffen}@ls5.cs.uni-dortmund.de
http://sunshine.cs.uni-dortmund.de/

Abstract. We develop an expansion-based algorithm for *semantic partial redundancy elimination* (*SPRE*), which overcomes the central drawbacks of the state-of-the-art approaches, which leave the program structure invariant: they fail to eliminate all partial redundancies even for acyclic programs. Besides being optimal for acyclic programs, our algorithm is unique in eliminating *all* partial k-redundancies, a new class of redundancies which is characterised by the number k of loop iterations across which values have to be kept. These optimality results come at the price of an in the worst case exponential program growth. The new technique is thus tailored for optimizing the typically considerably small computational "hot" spots of a program. Here it is particularly promising because its structural simplicity supports extensions to uniformly capture further powerful optimisations like constant propagation or strength reduction in a mutually supportive way.

1 Motivation

Background *Partial redundancy elimination* (*PRE*) is an important, widely used optimisation for performance improvement. Particularly powerful are approaches aiming at the elimination of *semantically* partially redundant computations (for short: SPRE). Intuitively, these are computations whose *values* are computed more than once along some program paths. Instead of recomputing such values, SPRE aims at replacing their (re-) computations by references to the respective values where possible. Intuitively, this is achieved by storing the values of computations for later reuse in temporaries as illustrated in the program fragment of Figure 1 showing the motivating example of [16]: as long as control stays inside the loop-like construct of Figure 1(a), the computations of $x + y$ and $a + b$ always yield the same value meaning that they are *semantically* (not syntactically[1]) partially redundant with respect to each other. These redun-

[1] Most commercial implementations consider equivalences on a syntactic basis as proposed e.g. in [13].

S. Jähnichen (Ed.): CC'99, LNCS 1575, pp. 91–107, 1999.

dancies can be eliminated by computing the values before entering the loop and replacing the original computations by references to the stored values as shown in Figure 1(b).

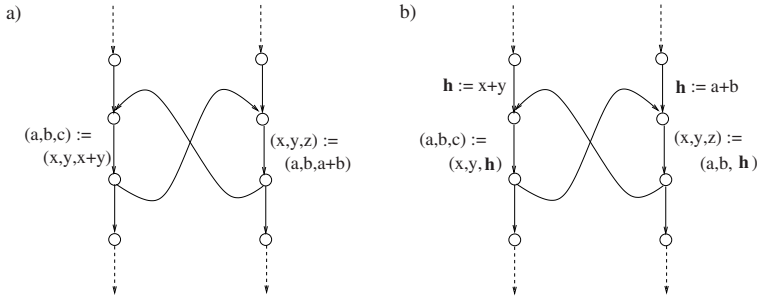

Fig. 1. Illustrating the essence of SPRE.[2]

State-of-the-art algorithms for SPRE (cf. [3,12,15,18,19]) are characterised by two common major design decisions made on the conceptual and technical side:

- *Conceptually:* the *flow graph* structure must not be modified,
- *Technically:* computations are *moved to* (rather than *placed at*) earlier program points for later reuse of their values.

Unfortunately, this has drawbacks already for *acyclic* programs:

Firstly, under the constraint of structural invariance certain redundancies cannot safely be eliminated at all, i.e., without impairing other program paths. This is illustrated in Figure 2(a). Note that such redundancies are by no means uncommon. Their most prominent manifestation is given by loop invariant code placed on a backedge as depicted in Figure 2(b). Even worse, there are situations like the one in Figure 2(c). Here, even a computation whose value has been computed before on every program path cannot be eliminated. In terms of Figure 2(c) this is because depending on the program path taken, the computation of $a' + b$ is either redundant with respect to the computation of $a + b$ or with respect to the computation of $c + b$. Thus, $a' + b$ cannot statically be replaced by a reference to the value of $a + b$ or $c + b$ as the one actually reused cannot be determined at compile time.

Secondly, SPRE-techniques limited to *moving* computations to earlier program points are inherently weaker than SPRE-techniques being capable of arbitrarily *placing* computations at appropriate program points. This was discussed in detail in [12], and here we only recall an example for illustration. In the program fragment of Figure 3(a) all redundancies can safely be eliminated as shown

[2] For the sake of presentation we allow parallel assignments $(x_1, \ldots, x_r) := (t_1, \ldots, t_r)$ as a shorthand for an appropriate sequence of assignments.

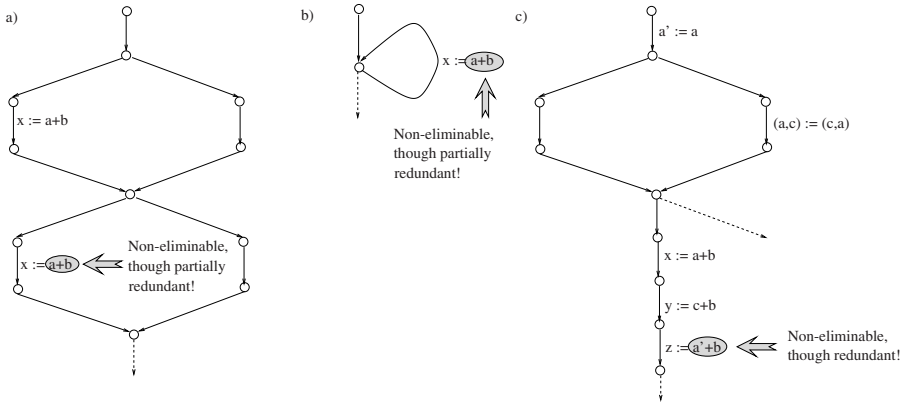

Fig. 2. *Conceptual* design constraint causes non-eliminable redundancies.

in Figure 3(c). However, motion-based SPRE-techniques are not capable of doing this and get stuck with the result of Figure 3(b). This is because neither $a+b$ nor $c+b$ are *anticipable* (*down-safe*) at the join point of control, and hence none of them can be hoisted across this point. In [12] we gave reasons why, under the

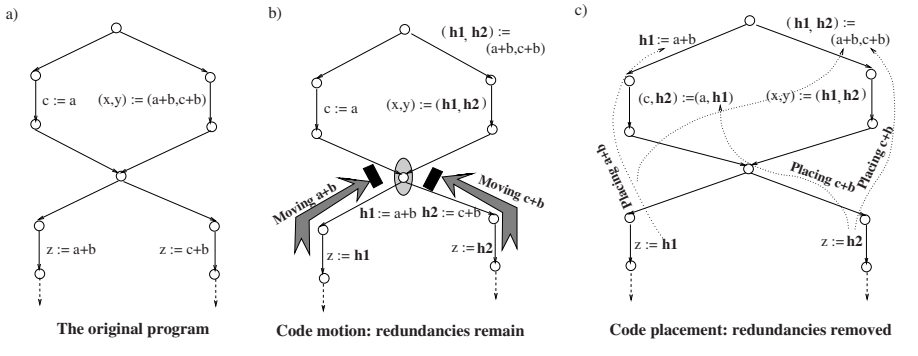

Fig. 3. *Technical* design constraint causes non-eliminable redundancies.

constraint of structural invariance, the problem of designing a general placement-based SPRE-technique cannot satisfactorily be solved: even in acyclic programs there are redundancies which can only be eliminated at the cost of introducing other ones, which excludes optimality in general. Though the heuristically based extension of a motion-based SPRE-algorithm proposed by Bodík and Anik can be considered a step towards placement-based algorithms (cf. [3]), the general problem of computing an optimal solution where one exists, is still unsolved.

Property-Oriented Expansion In this article, we present an *expansion-based* algorithm, which preserves the *flow tree* structure, i.e., it preserves the branching structure, but duplicates nodes in order to avoid "destructive joins" (see e.g. Figures 2 and 3 for examples of such joins).

To understand the basic idea behind our approach, note that on the execution tree of a program every redundancy is total and can thus successfully be eliminated. Of course, this insight is merely of theoretical interest because the execution tree is infinite for cyclic programs. However, this observation yields an easy access to the informal understanding of our algorithm. Basically, it works by *expanding* (i.e., unrolling) the program in a *demand-driven* fashion. The expansion is controlled by means of on-the-fly computed semantic equivalence information of program terms. Hence, the algorithm works by means of *property-oriented expansion* in the sense of [17].

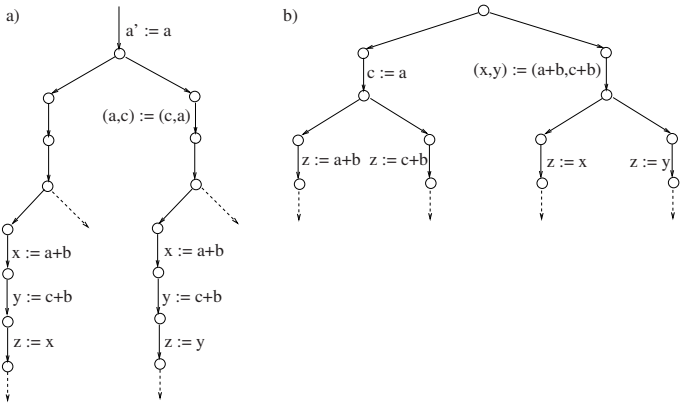

Fig. 4. Optimisations achieved by our approach for the examples of Figure 2(c) and Figure 3(a).

Conceptually, this approach has previously been applied to the *complete* elimination of *syntactic* partial redundancies (cf. [17]), a result which has later been optimized in order to avoid unnecessary code duplication (cf. [4]). The approach here parallels the syntactic approach under a *semantic* perspective leading to substantially stronger optimality results. Its unique power, which will be discussed in detail in the course of this article, is illustrated in Figures 4, 7, and 8.

Contribution While in the syntactic case partial redundancies can be eliminated completely [17] there is no chance for such a result in the semantic setting. Figure 5(a) shows a program containing an unbounded number of redundancies which can only partly be captured by unrolling the program.[3]

[3] In fact, every process ending with a finite program will fail.

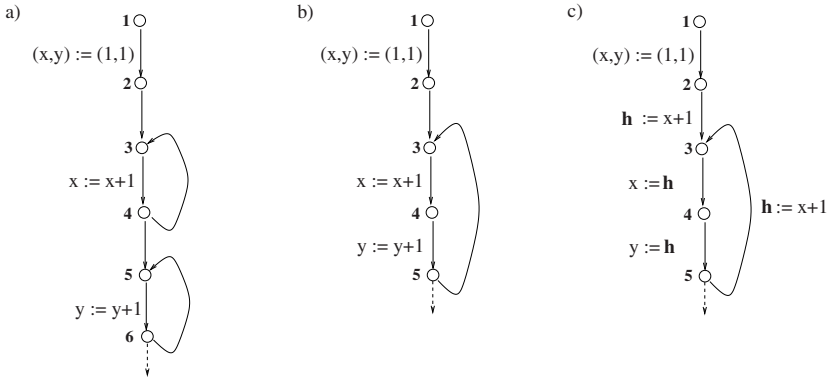

Fig. 5. a) Cyclic program with non-eliminable redundancies b) Cyclic program with eliminable redundancies c) Code motion transformation of b).

The point of this example is that the ith computation of $y+1$ in the second loop has the same value as x has after executing i iterations of the first loop. Hence, eliminating all partial redundancies would require to unroll both loops infinitely often. On the other hand, there are also situations where redundancies in cyclic programs can be fully eliminated. Figure 5(b) gives an example where even "classical" semantic code motion techniques succeed (cf. Figure 5(c)). The essential difference between both situations is that in case of Figure 5(b) the value of $x+1$ is immediately available for a usage before it is recomputed while in Figure 5(a) the value is possibly rewritten an unbounded number of times while still being usable. The notion of partial k-redundancies (cf. Definition 2) is tuned for taking this phenomenon into account. In essence, a k-redundancy can be eliminated by keeping a value for at most k loop iterations.[4]

In our demand-driven expansion process this is reflected in a mechanism which keeps the semantic equivalence information finite, while being tailored for eliminating all partial k-redundancies. In particular, this assures the termination of the expansion process. Moreover, in acyclic programs every partial redundancy is a partial 0-redundancy. Thus, for *acyclic* programs our algorithm eliminates *all* partial redundancies. This is achieved while simultaneously all the complications associated with semantic code motion and placement are avoided. All this is out of the scope of any structure-preserving[5] approach as demonstrated by the example of Figure 2.

The enormous optimizing power of our algorithm comes at the price of an in the worst case exponential program growth. The new technique is therefore not meant to completely replace previous SPRE-algorithms. Rather it is an ex-

[4] Note that except for the redundancy of Figure 5(a) all other redundancies occurring in the examples discussed so far are partial 0-redundancies.

[5] Here and in the following "structure-preserving" is used as a shorthand for *flow-graph-structure* preserving.

tremely powerful means for optimizing the "hot" spots of a program, which can be assumed to be considerably small. Here, our approach is particularly promising because its conceptual and structural simplicity make it easily extensible to uniformly capture further powerful optimisations like partially redundant assignment elimination, constant propagation, or strength reduction.

2 Preliminaries

We consider procedures of imperative programs, which we represent by means of directed edge-labelled *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. We assume that all statements are either the *empty statement* "skip" or a *3-address assignment* of the form $x := y$ or $x := y_1\ op\ y_2$ where x, y, y_1, y_2 are variables and op a binary operator. However, an extension to assignments involving complex right-hand side terms is straightforward. Unlabelled edges are assumed to represent "skip."

For a flow graph G, let $pred(n)$ and $succ(n)$ denote the set of all immediate predecessors and successors of a node n. Similarly, 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 $dest(e_q) = n$. $\mathbf{P}[m, n]$ denotes the set of all finite paths from m to n, and for $p \in \mathbf{P}[n, m]$ and $q \in \mathbf{P}[m, n']$ the concatenation of p and q will be written as $p; q$. Without loss of generality we assume that every node $n \in N$ lies on a path from $\mathbf{s}$ to $\mathbf{e}$. An edge of a path p which is labelled by an assignment is called a *constituent* of p. The set of constituents of a path p is denoted by $\mathcal{C}(p)$. Finally, $\#_e(p)$ denotes the number of occurrences of edge e on a path p.

The *semantics* of terms is induced by the *Herbrand interpretation* $\mathbf{H} = (\mathbf{T}, \mathbf{H}_0)$, where the data domain is given by the set of terms $\mathbf{T}$ which are inductively composed of variables, constants, and operators, and $\mathbf{H}_0$ is the function which maps every constant c to c and every operator op to the total function $\mathbf{H}_0(op) : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ defined by $\mathbf{H}_0(op)(t_1, t_2) =_{df} op(t_1, t_2)$. $\Sigma = \{\sigma \mid \sigma : \mathbf{V} \to \mathbf{T}\}$ denotes the set of all *Herbrand states* and σ_0 the distinct *start state* which is the identity on $\mathbf{V}$. The *semantics* of terms $t \in \mathbf{T}$ is given by the *Herbrand semantics* $\mathbf{H} : \mathbf{T} \to (\Sigma \to \mathbf{T})$ which is inductively defined by:

$$\mathbf{H}(t)(\sigma) =_{df} \begin{cases} \sigma(v) & \text{if } t = v \text{ is a variable} \\ \mathbf{H}_0(c) & \text{if } t = c \text{ is a constant} \\ \mathbf{H}_0(op)(\mathbf{H}(t_1)(\sigma), \mathbf{H}(t_2)(\sigma)) & \text{if } t = op(t_1, t_2) \end{cases}$$

With every edge e a state transformation and a backward substitution function are associated. If $e \equiv x := t$ the corresponding state transformation is defined

by $\theta_e(\sigma) =_{df} \sigma[\mathbf{H}(t)(\sigma)/x]$, and the backward-substitution of e for a term t' is defined by $\delta_e(t') =_{df} t'[t/x]$. If e represents skip, the two functions are the identity on their domain. These definitions can inductively be extended to finite paths. The following result describes their relationship:

Lemma 1. $\forall \sigma \in \Sigma\ \forall\ t \in \mathbf{T}.\ \mathbf{H}(t)(\theta_e(\sigma)) \ = \ \mathbf{H}(\delta_e(t))(\sigma)$

Now, we can define the notion of (semantically) partially redundant computations:

Definition 1. *Let* $e_1 \equiv x_1 := t_1$ *and* $e_2 \equiv x_2 := t_2$ *be labelled edges and* $n_i =_{df} source(e_i)$, $m_i =_{df} dest(e_i)$ $(i = 1, 2)$.

1. t_2 *at* e_2 *is* partially redundant *with respect to* t_1 *at* e_1 *iff*

$$\exists p \in \mathbf{P}[m_1, n_2]\ \exists q \in \mathbf{P}[\mathbf{s}, n_1].\ \mathbf{H}(t_1)(\theta_q(\sigma_0)) = \mathbf{H}(t_2)(\theta_{q;\langle e_1\rangle;p}(\sigma_0))$$

2. t_2 *at* e_2 *is* strong partially redundant *with respect to* t_1 *at* e_1 *iff*

$$\exists p \in \mathbf{P}[m_1, n_2]\ \forall q \in \mathbf{P}[\mathbf{s}, n_1].\ \mathbf{H}(t_1)(\theta_q(\sigma_0)) = \mathbf{H}(t_2)(\theta_{q;\langle e_1\rangle;p}(\sigma_0))$$

3. t_2 *at* e_2 *is* (strong) totally redundant *iff every path* $p \in \mathbf{P}[\mathbf{s}, n_2]$ *contains an edge* $e_3 \equiv x_3 := t_3$ *such that* t_2 *at* e_2 *is* (strong) partially redundant *with respect to* t_3 *at* e_3.

Obviously, strong partial redundancy implies partial redundancy, and strong total redundancy total redundancy. Figure 2 demonstrated that structure-invariant approaches fail to eliminate some partial redundancies and even some total non-strong ones. In fact, they are only complete for the class of strong total redundancies (cf. Definition 1(3)).

As discussed in Section 1 it is impossible to eliminate all partial redundancies because the number of values to be stored for later reuse can be unbounded. The notion of k-redundancy introduced next is tuned to take this into account. Intuitively, k-redundancies can be eliminated by keeping values for at most k loop iterations. This notion of redundancy is not only reasonably general, it is also scalable due its parameterisation in k. In fact, the choice of k provides an easy handle to control the trade-off between power and performance of our algorithm.

Definition 2. *Let* $k \in I\!N$. *In the situation of Definition 1 we call* t_2 *at* e_2 *partially* k*-redundant with respect to* t_1 *at* e_1 *iff*

$$\begin{aligned}\exists p \in &\mathbf{P}[m_1, n_2].\ \#_{e_1}(p) \leq k\ \wedge\ \exists q \in \mathbf{P}[\mathbf{s}, n_1]\\ &\forall q' \in \mathbf{P}[\mathbf{s}, n_1].\ \mathcal{C}(q) = \mathcal{C}(q') \Rightarrow \mathbf{H}(t_1)(\theta_{q'}(\sigma_0)) = \mathbf{H}(t_2)(\theta_{q';\langle e_1\rangle;p}(\sigma_0))\end{aligned}$$

Essentially, the condition $\#_{e_1}(p) \leq k$ ensures that the value of t_1 has to be preserved for at most k loop iterations. Like the notion of strong partial redundancy also the one of partial k-redundancy rests on a set of paths entering n_1. While, however, in the definition of strong partial redundancy p must fit for all paths

entering n_1, p in Definition 2 must only fit for the subset of paths sharing the same constituents. Thus, for acyclic control flow partial k-redundancy coincides for every k with the "weak" variant of partial redundancy of Definition 1(1).
Two expressions t_1 and t_2 are called *Herbrand equivalent* at node n, in symbols $t_1 \overset{\mathbf{H}}{\sim}_n t_2$, iff $\forall p \in \mathbf{P}[\mathbf{s}, n]$. $\mathbf{H}(t_1)(\theta_p(\sigma_0)) = \mathbf{H}(t_2)(\theta_p(\sigma_0))$. Herbrand equivalences (or *transparent equivalences* (cf. [15])) can be compactly represented by means of structured partition DAGs (SPDAGs). In essence, an SPDAG is a DAG (directed acyclic graph) whose nodes are labelled with constants, operators and variables such that[6]

- leaf nodes are labelled with at most one constant and (possibly many) variables, and
- inner nodes are labelled with at most one operator and (possibly many) variables, and
- constants and variables are assigned to at most one node, and
- two inner nodes with the same operator have different left or right children.

An SPDAG D induces a unique partition on the terms T_D that are represented by D. Figure 6 depicts the partition view of the leftmost SPDAG. We write $t_1 \equiv_D t_2$ to indicate that $t_1, t_2 \in T_D$ are in the same class of D. An instruc-

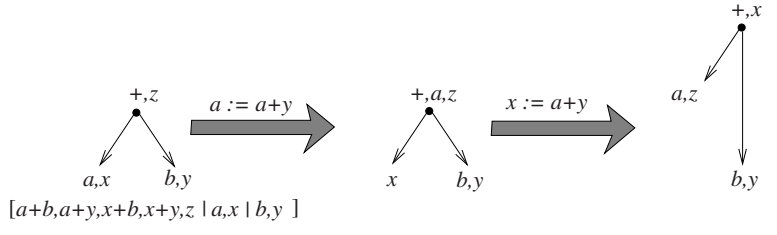

Fig. 6. Sequence of assignments and the associated transformations on SPDAGs

tion ι of the flow graph induces a flow function f_ι on SPDAGs which for a given SPDAG D computes the SPDAG that represents the equalities after executing ι. While $f_{\texttt{skip}}$ is the identity on SPDAGs, computing the effect of $f_{x := t}$ comprises the following three steps:

i) expanding D by t if it is not yet present in D,
ii) changing the variable position of x, and
iii) eliminating unlabelled leaves and their ingoing edges.[7]

Figures 6 illustrates this process by showing the impact of two assignments to an initial SPDAG.

[6] A formal definition can be found in [18].
[7] This may generate nodes with operator labelling whose operands are (partly) missing. At such nodes the operator labels and outgoing edges are removed. As this possibly generates new unlabelled leaves, the process has to be iterated.

3 The Algorithm

3.1 Overview

Conceptually, our algorithm consists of two stages:

- *Expansion stage*: in this phase the program model is partly unrolled according to a guided demand-driven expansion process. During this stage variables are renamed according to a naming discipline which keeps track of the values computed.
- *Elimination stage*: in this phase redundant computations are replaced by a reference to a variable storing the relevant value.

Note, in an implementation, both stages can be combined, i.e., replacing redundant computations can be done on-the-fly while unrolling. In the following we will describe both stages in detail.

The Expansion Stage As in the syntactic case the process of expansion is guided by a property-oriented duplication of the original program points (cf. [17]). While, however, in the syntactic setting the properties guiding this process are redundancy sets containing patterns of program assignments, the properties here are given in terms of SPDAGs. The expansion process for programs proceeds in four steps:

1. Set $(\mathbf{s}, \bot)$ to be a reachable node of the expanded program, where $\bot$ denotes the empty SPDAG.
2. Choose a yet non-processed node (n, D) of the expanded program and mark it as processed.
3. For each edge $e = (n, m)$ in the original flow graph (cf. Section 3.2 for details):
   (a) construct a modified edge label ι_{mod} from the label ι of e.
   (b) consider the pair $(m, f^k_{\iota_{mod}}(D))$, and add it to the set of reachable nodes of the expanded flow graph, whenever it is new. $f^k_{\iota_{mod}}$ is the k-restricted flow-function associated with instruction ι_{mod}.
   (c) Draw an edge with label ι_{mod} between (n, D) and $(m, f^k_{\iota_{mod}}(D))$.
4. Continue with the second step until all reachable nodes are processed.

The Elimination Stage In this stage every labelling $x := x_1 \; op \; x_2$ of an edge e is replaced by $x := y$ if the SPDAG D annotating $source(e)$ contains information on the equality of y and $x_1 \; op \; x_2$, i.e., $y \equiv_D x_1 \; op \; x_2$. In the case that x equals y the complete assignment can be removed.

3.2 Details

The application of the flow functions f_ι as introduced in Section 2 results easily in the construction of annotating SPDAGs of unbounded size when applied to programs with loops. However, by restricting the flow functions it is possible to

limit the growth of the SPDAGs while retaining enough information in order to capture partial k-redundancy completely. In essence, this is achieved by using a well-suited naming discipline to keep track of old values, and by carefully cutting SPDAGs.

Naming Discipline The name space for values is restricted to the variable names of the original program plus a set of fresh indexed variables $x^{(e,i)}$, where x is a variable, e an edge of the original program, and $0 \leq i \leq k$ a counter.[8] Intuitively, fresh variables are used to store values generated by different instances of the same statement during the expansion process. To keep track on the current names of variables each generated node (n, D) of the expanded flow graph is associated with

- a mapping $cnt_{(n,D)}[\bullet] : E \to [0, k]$ that maintains for each edge e of the original program the current counter value used to produce the name of the next instance.
- a mapping $curr_{(n,D)}[\bullet]$ from the name space of the original program to the name space of the expanded program. For a variable x of the original program $curr_{(n,D)}[x]$ denotes the current instance of x that is valid at node (n, D).

Initially, for $(\mathbf{s}, \bot)$ all counters $cnt_{(\mathbf{s},\bot)}[e]$ are set to 0 and $curr_{(\mathbf{s},\bot)}[x] = x$ for every variable x. Whenever a new edge $e' = ((n, D), (m, D'))$ is constructed the value of its edge counter associated with $e = (n, m)$ is incremented in a cyclic fashion:

$$cnt_{(m,D')}[e] = (cnt_{(n,D)}[e] + 1) \bmod (k + 1)$$

Moreover, if $x^{(e,i)}$ is the left-hand side variable of the edge label of e' then the current instance is updated accordingly:[9]

$$curr_{(m,D')}[x] = x^{(e,i)}$$

All other counters and values of current variables are unalteredly adopted from (n, D). It should be noted that this process is similar to the SSA-naming discipline (cf. [8,15]). However, it totally avoids the necessity of introducing ϕ-functions.

Modifying Edge Annotations While processing an edge $e = (n, m)$ of the original flow graph labelled with $x := t$, the constructed transition in the expanded model is labelled with a renamed assignment $x^{(e,i)} := t'$. This happens in two steps. Firstly, all variables in the right-hand side expression t are replaced by their current instances associated with (n, D), i.e., $t' = curr_{(n,D)}[t]$.[10] Secondly, the instance of the left-hand side variable is determined by the associated e-counter at (n, D), i.e., $i = cnt_{(n,D)}[e]$.

[8] For $k = 0$ the second parameter is sometimes omitted for the sake of presentation.

[9] Note that $x^{(e,i)}$ is computed exploiting information on the counters (see paragraph "Modifying edge annotations").

[10] $curr_{(n,D)}[t]$ denotes the straightforward extension of $curr_{(n,D)}[\bullet]$ to terms.

***k*-restricted Flow Functions** The previous steps alone are not sufficient to avoid unbounded growth of SPDAGs. This is achieved by a modification of the flow functions that eliminates "unreferencable" parts of SPDAGs. Let (n, D) be a constructed node of the expanded flow graph and ι an instruction. The k-restricted flow function $f_\iota^k(D)$ is realized through a two-step procedure.

1. *Computing the value flow:* D is subjected to the standard flow function f_ι.
2. *Cutting the resulting SPDAG:* nodes carrying only an operator label are eliminated together with their ingoing and outgoing edges. This may leave some operator labels at leaf nodes which must also be removed.

We demonstrate our algorithm for $k = 0$ using the example of Figure 3(a) for illustration. After the expansion phase we get the program depicted in Figure 7, annotated with SPDAGs at its program points. Instances of a node **n** of the original program are numbered **n-1**,**n-2**, etc. Moreover, e_1 to e_4 refer to the four edges labelled with $c := a$, $(x, y) := (a + b, c + b)$, $z := a + b$ and $z := c + b$, respectively. The results from the elimination phase are emphasized by the grey arrows. Note that the equality of x and $a + b$ at node **7-2** and of y and $c + b$ at node **9-2** allows us to replace the right-hand side expressions on the edges (**7-2**,**8-2**) and (**9-2**,**10-2**), which cannot be eliminated by motion-based redundancy elimination techniques.

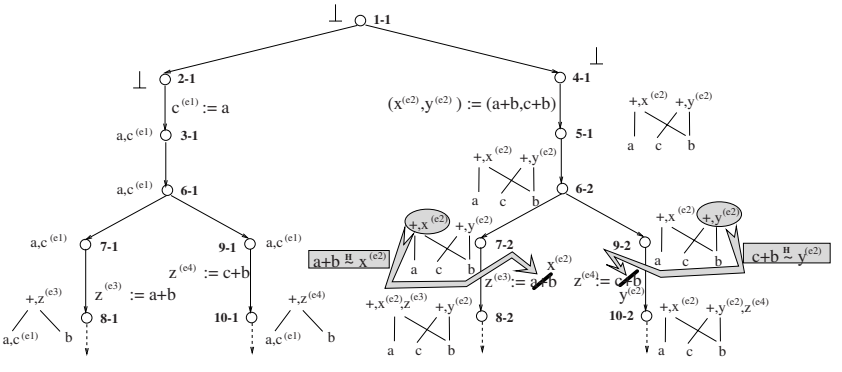

Fig. 7. Illustrating the algorithm for Figure 3(a).

3.3 Results

The algorithm is correct, i.e., it preserves the semantics of its argument programs. The proof of this property benefits from Lemma 2. Its first part is fundamental for proving the correctness of the expansion phase, while the $\Leftarrow$-direction of its second part guarantees the correctness of the replacements of the elimination phase.

The lemma requires the following projections relating the expanded and the original program. A path p in the expanded program has a unique corresponding path p_o in the original one, and a variable x which is part of an edge label in the expanded flow graph has a unique corresponding variable x_o in the original flow graph. It simply results from removing the optional superscript; a process, which applies naturally to terms, too.

Lemma 2. *Let (n, D) be a node in the expanded program (before the elimination phase), and $x := t$ the edge annotation of an outgoing edge of (n, D). Then we have:*

1. $\forall p \in \mathbf{P}[(\mathbf{s}, \bot), (n, D)].\ \mathbf{H}(t_o)(\theta_{p_o}(\sigma_0)) \ = \ \mathbf{H}(t)(\theta_p(\sigma_0))$
2. $\forall t_1 \in T_D\ \forall t_2 \in \mathbf{T}.\ t_1 \overset{\mathbf{H}}{\sim}_{(n,D)} t_2 \iff t_1 \equiv_D t_2$

Both parts can be proved by induction, the first one on the length of path p and the second one on the length of a shortest path leading to (n, D). By means of Lemma 2, we can now prove the main result of this article:

Theorem 1 (k-Optimality). *For a given $k \in \mathbb{N}$ the procedure of Section 3 terminates after eliminating* all *partial k-redundancies of the argument program.*

Suppose there is an instance of a partial k-redundancy in the expanded program before the elimination step has been performed, say between t_1 at edge e_1 and t_2 at edge e_2. The definition of partial k-redundancy ensures that this partial redundancy is a strong one in the expanded program. This is because paths with distinct constituents lead to the construction of different SPDAGs, and hence to different nodes in the expanded program. Let p be the intermediate path between $dest(e_1) =_{df} (n, D_1)$ and $source(e_2) =_{df} (m, D_2)$, and $x^{(e_1,i)}$ the left-hand side variable associated with e_1.

Exploiting that the partial redundancy is strong, Lemma 1 yields that $\delta_p(t_2)$ is Herbrand-equivalent with t_1 at (n, D_1). Hence the $\Rightarrow$-direction of Lemma 2(2) ensures that $\delta_p(t_2)$ is contained in D_1 with $\delta_p(t_2) \equiv_{D_1} x^{(e_1,i)}$. Moreover, the definition of k-redundancy ensures that $x^{(e_1,i)}$ is not rewritten on p. With the definition of the flow functions f^k and with $D_2 = f^k_p(D_1)$ it is easy to see that $\delta_p(t_2) \in T_{D_1}$ implies that $t_2 \in T_{D_2}$. This finally grants $t_2 \equiv_{D_2} x^{(e_1,i)}$ which makes t_2 eliminable at e_2. It can be replaced by $x^{(e_1,i)}$.

Recall that for acyclic programs the notion of partial 0-redundancy coincides with partial redundancy. Thus, we get as an immediate corollary of Theorem 1:

Corollary 1. *On acyclic programs the procedure of Section 3 terminates for $k = 0$ after eliminating* all *partial redundancies of the argument program.*

In particular, this guarantees that our approach resolves for acyclic programs the motion/placement-anomalies of structure-invariant approaches. Moreover, the definition of strong partial redundancies immediately yields:

Corollary 2. *There exists a $k \in \mathbb{N}$ such that the procedure of Section 3 terminates after eliminating* all *strong partial redundancies of the argument program.*

4 Discussion

In this section we discuss some significant characteristics of our algorithm and illustrate them by examples which simultaneously demonstrate the power of the approach.

Initialisation Freedom One of the most significant characteristics of our algorithm is that it does not lengthen any execution sequence of the original program. This is in contrast to the structure-invariant setting where optimisations come at the price of inserting initialisation statements. This reflects an abstract cost model, where initialisations are for free. Rosen, Wegman and Zadeck criticised such a model as impractical [15], since even in acyclic programs the elimination of some redundancies requires reinitialisation chains of arbitrary length, whose execution costs may easily exceed those of the saved computations.

Loop Unrolling It is well-known that loop-invariant computations like those in the example of Figure 8(a) can be eliminated by unrolling the loop once because the previously partially redundant computations become totally redundant and can therefore be eliminated. Figure 8(b) shows the corresponding final result of this conventional approach. It is worth comparing this result with the

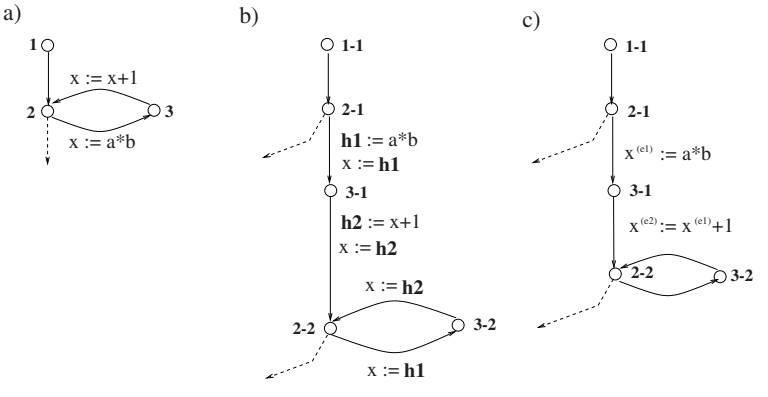

Fig. 8. Loop unrolling vs. expansion.

result of our approach depicted in Figure 8(c). Based on the for code motion typical assumption that reinitialisations are for free, the "classical result" introduces additional assignments to temporaries on program paths. In contrast, our solution (for $k = 0$) does not have any additional assignments and is therefore optimal independently of any simplifying assumptions. In particular, the assignments $x^{(e_1)} := a * b$ and $x^{(e_2)} := x^{(e_1)} + 1$ labeling the edges inside of the

loop after the expansion phase are completely removed as they reduce to assignments $x^{(e_1)} := x^{(e_1)}$ and $x^{(e_2)} := x^{(e_2)}$, respectively. In the case of Figure 8(b) obtaining an equivalent effect requires a postprocess, e.g. a combination of dead code elimination [1] and redundant assignment elimination [11].

Loop-Carried Redundancies "Loop-carried" partial redundancies reveal the function of the parameter k in our algorithm.

The example of Figure 9(a) shows a program that has a loop structure where the computation of $y+1$ is (strongly) redundant with respect to the computation of $x+1$ of the previous iteration. In the structure invariant setting this "loop-carried" redundancy can be eliminated by means of an additional reinitialisation statement as shown in Figure 9(b). By inserting a higher number of x-increments in front of the loop in Figure 9(a) one can easily construct examples where the elimination of a loop carried redundancy requires reinitialisation sequences of arbitrary length.

In contrast, the expansion-based approach is capable of removing redundancies carried over k loop iterations without introducing any additional assignment on a program path. Figure 9(c) shows the result of our algorithm for $k=1$. Essentially, the expansion process automatically results in a program where two loop iterations of the original program are combined to a single loop with two exits.

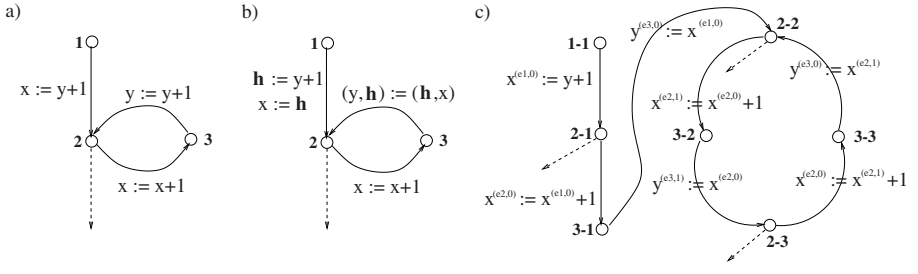

Fig. 9. Elimination of loop-carried (strong) redundancies.

5 Related Work

Semantic redundancy elimination has been pioneered by the *value numbering* approach of Cocke and Schwartz [7], which, however, was tailored for basic block optimisation. Algorithms aiming at the *global* elimination of semantic partial redundancies have been first proposed in [15,16,18,19], and more recently in [3,12].[11] Common to all of these algorithms is to respect the constraint of

[11] The "global" algorithms proposed in [5,14] have a slightly different accent as they concentrate on motion to dominators.

structural invariance, and their phase structure. As the impact of structural invariance has been investigated in the previous sections, we here focus on the second point, the phase structure. The algorithm of [12], for instance, consists of three stages each with up to three substeps which require considerably complex analyses.

In comparison, the algorithm proposed here consists only (of a simplified variant) of one of the substeps. This step, the computation of equivalence information by means of an algorithm matching the pattern of Kildall's algorithm [10], is significantly simpler.[12] The computation of the meet of equivalence information at merge points, a computational bottleneck, is replaced by the much simpler test of equality, which is used to trigger the node splitting. Hence, our extremely powerful optimisation algorithm is very easy to implement. This comes at the price of an in the worst case exponential program growth. Thus, as discussed below, the practical application of this aggressive algorithm requires some care.

6 Conclusions and Perspectives

Previous approaches for eliminating semantic partial redundancies are all designed under the maxim of structural invariance, i.e., not to affect the program structure. Unfortunately, under this constraint the SPRE-problem lacks a satisfying solution even for acyclic programs. Dropping the constraint, and trading size against efficiency, we developed a new SPRE-algorithm working by property-oriented expansion, which eliminates all partial k-redundancies. In acyclic programs it eliminates even all partial redundancies.

These optimality results come at the price of an in the worst case exponential program growth. The new technique is therefore not meant to completely replace previous SPRE-algorithms. Rather it is an extremely powerful means for optimizing the "hot" spots of a program, which can be assumed to be considerably small. Here, our approach is particularly promising, because of its structural simplicity the new algorithm is open for extensions. Optimisations like partially redundant assignment elimination (cf. [11,17]), constant propagation (cf. [9,10]), or strength reduction (cf. [2,6]) can uniformly be integrated which results in an extremely powerful expansion-based algorithm, where the integrated techniques mutually profit from each other. The integration boils essentially down to exploiting arithmetic properties of term operators and evaluating terms accordingly in the fashion of the strength reduction algorithm of [19]. All this can be done on-the-fly during the expansion process. In this manner it is possible to obtain a demand-driven uniform algorithm capturing the strong interdependencies and their corresponding problem of good application orders (cf. [20]).

[12] Note that redundancy elimination boils down to locally look up the required value in the respective SPDAG.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985. 104
2. F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3, pages 79 – 101. Prentice Hall, Englewood Cliffs, New Jersey, 1981. 105
3. R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *Conf. Rec. 25th Symp. on Principles of Programming Languages* (*POPL'98*), pages 237 – 251. ACM, NY, 1998. 92, 93, 104
4. R. Bodík, R. Gupta, and M.-L. Soffa. Complete removal of redundant expressions. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl.* (*PLDI'98*), volume *33* of *ACM SIGPLAN Not.*, pages 1 – 14, 1998. 94
5. C. Click. Global code motion/global value numbering. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl.* (*PLDI'95*), volume *30*,6 of *ACM SIGPLAN Not.*, pages 246–257, 1995. 104
6. J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Comm. ACM*, 20(11):850 – 856, 1977. 105
7. J. Cocke and J. T. Schwartz. Programming languages and their compilers. Courant Inst. Math. Sciences, NY, 1970. 104
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451 – 490, 1991. 100
9. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305 – 317, 1977. 105
10. G. A. Kildall. A unified approach to global program optimization. In *Conf. Rec. 1st Symp. Principles of Prog. Lang.* (*POPL'73*), pages 194 – 206. ACM, NY, 1973. 105
11. J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* (*PLDI'95*), volume *30*,6 of *ACM SIGPLAN Not.*, pages 233 – 245, 1995. 104, 105
12. J. Knoop, O. Rüthing, and B. Steffen. Code motion and code placement: Just synomyms? In *Proc. 7th European Symp. on Programming* (*ESOP'98*), LNCS 1381, pages 154 – 169. Springer-V., 1998. 92, 93, 104, 105
13. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96 – 103, 1979. 91
14. J. H. Reif and R. Lewis. Symbolic evaluation and the global value graph. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'77*), pages 104 – 118. ACM, NY, 1977. 104
15. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th Symp. Principles of Prog. Lang.* (*POPL'88*), pages 2 – 27. ACM, NY, 1988. 92, 98, 100, 103, 104
16. B. Steffen. Optimal run time optimization - Proved by a new look at abstract interpretations. In *Proc. 2nd Int. Conf. Theory and Practice of Software Development* (*TAPSOFT'87*), LNCS 249, pages 52 – 68. Springer-V., 1987. 91, 104
17. B. Steffen. Property-oriented expansion. In *Proc. 3rd Stat. Analysis Symp.* (*SAS'96*), LNCS 1145, pages 22 – 41. Springer-V., 1996. 94, 99, 105
18. B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proc. 3rd Europ. Symp. Programming* (*ESOP'90*), LNCS 432, pages 389 – 405. Springer-V., 1990. 92, 98, 104

19. B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proc. 4th Int. Conf. Theory and Practice of Software Development* (*TAPSOFT'91*), LNCS 494, pages 394 – 415. Springer-V., 1991. 92, 104, 105
20. D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (*PPOPP'90*), volume *25*,3 of *ACM SIGPLAN Not.*, pages 137 – 147, 1990. 105