

Composing Transformations to Optimize Linear Code

Thomas Noll and Stefan Rieger

RWTH Aachen University
Software Modeling and Verification Group
52056 Aachen, Germany
{noll,rieger}@cs.rwth-aachen.de

Abstract. We study the effect of an optimizing algorithm for straight-line code which first constructs a directed acyclic graph representing the given program and then generates code from it. We show that this algorithm produces optimal code with respect to the classical transformations known as Constant Folding, Common Subexpression Elimination, and Dead Code Elimination. In contrast to the former, the latter are also applicable to iterative code containing loops. We can show that the graph-based algorithm essentially corresponds to a combination of the three classical optimizations in conjunction with Copy Propagation. Thus, apart from its theoretical importance, this result is relevant for practical compiler design as it potentially allows to exploit the optimization potential of the graph-based algorithm for non-linear code as well.

1 Introduction

Literature on optimizing compilers describes a wide variety of code transformations which aim at improving the efficiency of the generated code with respect to different parameters. Most of them concentrate on specific aspects such as the elimination of redundant computations or the minimization of register usage. There are, however, also combined methods which integrate several optimization steps in one procedure.

In this paper we compare several classical optimizing transformations for straight-line code with a combined procedure which first constructs a directed acyclic graph (DAG) representing the given program and then generates optimized code from it. The basic version of the latter has been introduced in [3], and the authors claim that it produces optimal results¹ regarding the length of the generated code. However the DAG procedure cannot directly be applied to iterative code containing loops, which on the other hand is possible for most of the classical transformations.

Although our analysis is limited to straight-line code in this contribution we have plans to extend it to iterative programs in the future.

In this paper we first present a slightly modified version of the DAG algorithm that in addition supports constant folding. We do not consider algebraic

¹ The optimality is given w.r.t. strong equivalence, see Sec. 2.

optimizations though, since our framework is independent of the interpretation of operator and constant symbols. Our results thus hold for arbitrary interpretations over any domain. We show that the DAG algorithm integrates the following three classical transformations:

Constant Folding, corresponding to a partial evaluation of the program with respect to a given interpretation of its constant and operation symbols;
Common Subexpression Elimination, aiming to decrease the execution time by avoiding multiple evaluations of the same (sub-)expression; and
Dead Code Elimination, removing computations that do not contribute to the actual result of the program.

It will turn out that these transformations are not sufficient to completely encompass the optimizing effect of the DAG algorithm. Rather a fourth transformation, *Copy Propagation*, has to be added, which propagates values in variable-copy assignments. This does not have an optimizing effect on its own but generally enables other transformations such as Common Subexpression Elimination and Dead Code Elimination, thus it is treated separately in an extra section.

Our investigation will be carried out in a framework in which we develop formal definitions for concepts such as linear programs and their semantics, (optimizing) program transformations, their correctness and their equivalence, etc. These preliminaries will be presented in Sec. 2, followed by the definition of the three classical transformations and of the DAG-based algorithm in Sec. 3 and 4, respectively. The subsequent Sec. 5 constitutes the main part of this paper, establishing the equivalence between the DAG procedure and the composition of the classical transformations.

To support the experimenting with concrete examples, a web-based implementation of the optimizing transformations is available at the URL [8].

Moreover we would like to mention that this paper is a condensed version of our technical report [7] which presents complete proofs for all of the propositions and includes more details and examples regarding the algorithms presented here.

2 SLC-Programs and Their Properties

Straight-line code (SLC) constitutes the basic blocks of the intermediate code of iterative programs. In particular it is contained in loop bodies whose efficient execution is crucial.

Syntax

An SLC-program consists of a sequence of assignments using simple arithmetic expressions without branching or loops.

A *signature* is a pair $\Sigma = (F, C)$ consisting of a finite set of *function symbols* (or: *operation symbols*) $F := \bigcup_{i=1}^{\infty} F^{(i)}$ where $F^{(i)}$ denotes the set of i -ary function symbols, and a set of *constant symbols* C . Furthermore $V := \{x, y, z, \dots\}$ denotes a possibly infinite set of variables.

An *SLC-program* is a quadruple $\pi = (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta)$ with a signature $\Sigma = (F, C)$, a vector of *input variables* $\mathbf{v}_{in} = (x_1, \dots, x_s)$ ($x_i \in V$), a vector of *output*

variables $\mathbf{v}_{out} = (y_1, \dots, y_t)$ ($y_i \in V$) where all input and output variables are pairwise distinct, and a block $\beta = \alpha_1; \alpha_2; \dots; \alpha_n$ with instructions α_i of the form $x \leftarrow e$ where $x \in V$ and $e \in (V \setminus \{x\}) \cup C \cup \{f(u_1, \dots, u_r) \mid f \in F^{(r)} \text{ and } \forall j \in \{1, \dots, r\} : u_j \in V \cup C\}$.

In addition every program is assumed to be *complete* in the sense that every variable is defined before being used (with the understanding that every input variable x_i is defined in the beginning while every output variable y_j is used in the end).

Moreover we introduce the following denotations: V_{in} and V_{out} denote the sets of input/output variables of π , C_π and V_π the sets of constant symbols and variables occurring in π , respectively, and V_α and V_e stand for the set of variables in the instruction α and in the expression e , respectively. Finally, \mathcal{SLC} denotes the set of all SLC–programs.

Figure 1 shows a simple SLC–program over the signature $\Sigma_{\mathbb{Z}} := (\{+^{(2)}, *^{(2)}, -^{(2)}\}, \mathbb{Z})$. For simplicity we employed the usual infix notation.

Semantics

The semantics of an SLC–program depends on the domain of the variables and on the interpretation of the operators and constant symbols. These are formally given by a Σ –algebra $\mathfrak{A} := (A, \varphi)$ with domain (universe) A and interpretation function $\varphi : F \cup C \cup A \rightarrow \bigcup_{i=0}^\infty \{\delta \mid \delta : A^i \rightarrow A\}$ where $\varphi(f) : A^r \rightarrow A$ for every $f \in F^{(r)}$, $\varphi(c) \in A$ for every $c \in C$, and $\varphi(a) = a$ for every $a \in A$.²

The current state in the computation of an SLC–program can be expressed as a mapping of the program’s variables to their values: $\sigma : V_\pi \rightarrow A$. This induces the *state space* $S := \{\sigma \mid \sigma : V_\pi \rightarrow A\}$.

Now every instruction $\alpha = x \leftarrow e$ determines a transformation $\mathfrak{A}[\![\alpha]\!] : S \rightarrow S$ of one state into another: the variable x is associated with the value resulting from the evaluation of the expression e . By composing the transformations of the instructions in a program π we obtain its semantics as a function that maps a vector representing the input values to a vector that contains the values of the output variables, that is, $\mathfrak{A}[\![\pi]\!] : A^s \rightarrow A^t$.

Note that the semantics is defined independent of the variable names, only the order of the variables in the input/output vectors is relevant. Moreover due to the completeness conditions on programs it suffices to give the values of the input variables; non–input variables can be initialized arbitrarily.

Equivalence and Optimality

For optimizations program equivalence is of high significance since programs have to be transformed in such a way that their semantics is preserved.

² Requiring the latter will turn out to be useful for constant folding; see Sec. 3 for details.

```

 $\mathbf{v}_{in} : (x, y)$ 
 $\beta : u \leftarrow 3;$ 
 $v \leftarrow x - y;$ 
 $w \leftarrow u + 1;$ 
 $x \leftarrow x - y;$ 
 $v \leftarrow w - 1;$ 
 $u \leftarrow x - y;$ 
 $z \leftarrow u * w;$ 
 $u \leftarrow 2 * u;$ 
 $\mathbf{v}_{out} : (u, v)$ 
    
```

Fig. 1. An SLC–program

Two SLC–programs π_1 and π_2 over some signature Σ are called \mathfrak{A} –*equivalent* for a Σ –algebra \mathfrak{A} ($\pi_1 \sim_{\mathfrak{A}} \pi_2$) iff $\mathfrak{A}[\pi_1] = \mathfrak{A}[\pi_2]$. If this holds for all interpretations \mathfrak{A} then they are called *strongly equivalent* ($\pi_1 \sim \pi_2$).

Note that (strong) equivalence of two programs requires that both have the same number of input and of output variables. The \mathfrak{A} –equivalence of two programs is generally undecidable [5]. This, however, is not the case for strong equivalence [7].

Assessing the quality of an optimization requires a notion of cost. As standard cost functions we will use the number of instructions and the number of operations (that is, instructions of the form $x \leftarrow f(u_1, \dots, u_r)$).

In general the optimality of a given program w.r.t. a cost function is undecidable (this follows from the undecidability of \mathfrak{A} –equivalence). Therefore from now on we will concentrate on transformations that *improve* programs instead of really *optimizing* them. (Nevertheless we will still call these “optimizations”.)

3 Classical Optimizations

After discussing the formal basis we will now focus on optimization algorithms for SLC–programs. In this section we will introduce the “classical optimizations”. Those are algorithms that are widely known and used in today’s compilers. Optimizations typically consist of an *analysis* and a *transformation* step. For an optimizing transformation of SLC–programs we require the following properties:

Definition 3.1. *A function $T : \mathcal{SLC} \rightarrow \mathcal{SLC}$ is called an \mathfrak{A} –program transformation for an interpretation \mathfrak{A} if, for every $\pi \in \mathcal{SLC}$, $T(\pi) \sim_{\mathfrak{A}} \pi$ (correctness) and $T(T(\pi)) = T(\pi)$ (idempotency). If T is correct for every interpretation, then we call it a program transformation.*

Dead Code Elimination

Dead Code Elimination removes instructions that are dispensable because they do not influence program semantics. An instruction $x \leftarrow e$ represents *dead code* if x is not used until it is redefined or if it is used only in instructions which are themselves dead code.

The transformation is based upon the *Needed Variable Analysis* which determines, for each instruction, those variables whose values are still required. It is a *backward analysis*, i.e., starting from the set of output variables the set of needed variables is computed for each instruction.

Definition 3.2 (Needed Variable Analysis). *Let $\pi := (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta = \alpha_1; \dots; \alpha_n$. For every instruction $\alpha = x \leftarrow e$ we define the transfer function $t_\alpha : 2^{V_\pi} \rightarrow 2^{V_\pi}$ as follows:*

$$t_\alpha(M) := \begin{cases} (M \setminus \{x\}) \cup V_e & \text{if } x \in M \\ M & \text{else} \end{cases}$$

The $t_{\alpha_n}, \dots, t_{\alpha_1}$ determine, beginning with V_{out} , the sets of needed variables:

$$NV_n := V_{out} \quad \text{and} \quad NV_{i-1} := t_{\alpha_i}(NV_i) \quad \text{for } i \in \{n, \dots, 2\}.$$

Using the sets of needed variables computed during the analysis step we can now define Dead Code Elimination.

Definition 3.3 (Dead Code Elimination). For $\pi = (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLLC}$ with $\beta = \alpha_1; \dots; \alpha_n$, the transformation $T_{DC} : \mathcal{SLLC} \rightarrow \mathcal{SLLC}$ is given by:

$$T_{DC}(\pi) := (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta') \text{ with } \beta' := t_{DC}(\alpha_1); \dots; t_{DC}(\alpha_n)$$

$$t_{DC}(x \leftarrow e) := \begin{cases} x \leftarrow e & \text{if } x \in NV_i \\ \varepsilon & \text{else} \end{cases}$$

This means all instructions $x \leftarrow e$ for which x is not in the set of needed variables are removed. A computation of NV_0 could be used for removing dispensable input variables. This would however conflict with our definition of the program semantics.

In Tab. 1 the analysis sets and the computation result of a T_{DC} -application to the program from Fig. 1 are shown.

Table 1. Application of the classical transformations to the program from Fig. 1

i	α_i	NV_i	d.c.?	AE_i	$vr(i)$	$T_{CS}(\pi)$	$RD_i(u, v, w, x, y, z)$	$T_{CF}(\pi)$
1	$u \leftarrow 3;$	$\{u, x, y\}$	No	\emptyset	\emptyset	$u \leftarrow 3;$	$(\perp, \perp, \perp, \perp, \perp, \perp)$	$u \leftarrow 3;$
2	$v \leftarrow x - y;$	$\{u, x, y\}$	Yes	\emptyset	$\{4\}$	$t_2 \leftarrow x - y;$ $v \leftarrow t_2;$	$(3, \perp, \perp, \perp, \perp, \perp)$	$v \leftarrow x - y;$
3	$w \leftarrow u + 1;$	$\{w, x, y\}$	No	$\{2\}$	\emptyset	$w \leftarrow u + 1;$	$(3, \perp, \perp, \perp, \perp, \perp)$	$w \leftarrow 4;$
4	$x \leftarrow x - y;$	$\{w, x, y\}$	No	$\{2, 3\}$	\emptyset	$x \leftarrow t_2;$	$(3, \perp, 4, \perp, \perp, \perp)$	$x \leftarrow x - y;$
5	$v \leftarrow w - 1;$	$\{v, x, y\}$	No	$\{3\}$	\emptyset	$v \leftarrow w - 1;$	$(3, \perp, 4, \perp, \perp, \perp)$	$v \leftarrow 3;$
6	$u \leftarrow x - y;$	$\{u, v\}$	No	$\{3, 5\}$	\emptyset	$u \leftarrow x - y;$	$(3, 3, 4, \perp, \perp, \perp)$	$u \leftarrow x - y;$
7	$z \leftarrow u * w;$	$\{u, v\}$	Yes	$\{5, 6\}$	\emptyset	$z \leftarrow u * w;$	$(\perp, 3, 4, \perp, \perp, \perp)$	$z \leftarrow u * 4;$
8	$u \leftarrow 2 * u;$	$\{u, v\}$	No	$\{5, 6, 7\}$	\emptyset	$u \leftarrow 2 * u;$	$(\perp, 3, 4, \perp, \perp, \perp)$	$u \leftarrow 2 * u;$

T_{DC} is correct and idempotent and thus a program transformation [7]. It should be noted that there are *non-idempotent* variants of Dead Code Elimination based on a so-called ‘‘Live-Variable Analysis’’ (see e.g. [4,6]).

Common Subexpression Elimination

Unlike Dead Code Elimination, Common Subexpression Elimination is using a forward analysis, the *Available Expressions Analysis*, which computes for each instruction the (indices of the) operation expressions whose value is still available and whose repeated evaluation can be avoided therefore.

Definition 3.4 (Available Expressions Analysis). Let $\pi := (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLLC}$ with $\beta = \alpha_1; \dots; \alpha_n$ and $\alpha_i = x_i \leftarrow e_i$ for every $i \in \{1, \dots, n\}$. An expression e is available at position i if $e_j = e$ for some $j < i$ and $x_k \notin V_e$ for every $j \leq k < i$.

The $t_{\alpha_i} : 2^{\{1, \dots, n\}} \rightarrow 2^{\{1, \dots, n\}}$ are given by $t_{\alpha_i}(M) := \text{kill}_{\alpha_i} \circ \text{gen}_{\alpha_i}$ where

$$\text{gen}_{\alpha_i}(M) := \begin{cases} M \cup \{i\} & \text{if } e_i = f(u_1, \dots, u_r) \text{ and } \forall j \in M : e_j \neq e_i \\ M & \text{else} \end{cases}$$

$$\text{kill}_{\alpha_i}(M) := M \setminus \{j \in M \mid x_i \in V_{e_j}\}$$

This yields the sets of available expressions $AE_i \subseteq \{1, \dots, n\}$ for $i \in \{1, \dots, n\}$:

$$AE_1 := \emptyset \quad \text{and} \quad AE_{i+1} := t_{\alpha_i}(AE_i) \quad \text{for } i \in \{1, \dots, n\}$$

Definition 3.5 (Common Subexpression Elimination). For each instruction, the function $vr : \{1, \dots, n\} \rightarrow 2^{\{1, \dots, n\}}$ yields the valid recurrences of the corresponding expression: $vr(i) := \{j \in \{i+1, \dots, n\} \mid i \in AE_j, e_i = e_j\}$.

The program transformation³ $T_{CS} : \mathcal{SLLC} \rightarrow \mathcal{SLLC}$ works as follows: for every $i \in \{1, \dots, n\}$ with $vr(i) \neq \emptyset$, select $t_i \in V \setminus V_\pi$ and

1. replace $\alpha_i = x \leftarrow e$ by $t_i \leftarrow e$; $x \leftarrow t_i$ and
2. for all $j \in vr(i)$, replace $\alpha_j = y \leftarrow e$ by $y \leftarrow t_i$.

Table 1 shows both the available expressions, the valid recurrences, and the result of eliminating common subexpressions in the program from Fig. 1.

Again it is possible to show that Common Subexpression Elimination is a program transformation. Please refer to [7] for details.

Constant Folding

Constant Folding is a partial evaluation of the input program with constant propagation. It avoids the redundant evaluation of constant expressions at runtime. In contrast to Dead Code and Common Subexpression Elimination the optimization is incorporating the program semantics as this is necessary for evaluating constant expressions.

During the program analysis we determine for every instruction the known values of the variables. For this the definition of the semantics (Sec. 2) is extended to allow the “evaluation” of expressions with unknown variable values (represented by the symbol \perp in Tab. 1). If at least one argument of a function is an unknown variable also the evaluation result is unknown. Special properties of operations, such as $\forall x \in \mathbb{R} : 0 \cdot x = 0$, are ignored.

The evaluation of constant expressions potentially causes the introduction of new constants (not contained in C). Therefore the signature of the target program needs to be adapted.

Definition 3.6 (Constant Folding). For $\pi = (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLLC}$, $\Sigma = (F, C)$, $\beta = \alpha_1; \dots; \alpha_n$, $\alpha_i = x_i \leftarrow e_i$ and $\mathfrak{A} = (A, \varphi)$, the transformation $T_{CF} : \mathcal{SLLC} \rightarrow \mathcal{SLLC}$ is defined by:

$$T_{CF}(\pi) := ((F, A), \mathbf{v}_{in}, \mathbf{v}_{out}, \beta') \quad \text{with } \beta' := x_1 \leftarrow \overline{RD}_1(e_1); \dots; x_n \leftarrow \overline{RD}_n(e_n)$$

where \overline{RD}_i replaces all variables known to be constant by the respective values, and evaluates constant expressions (see [7] for the formal definition).

³ For a proof of this property refer to [7].

Example 3.7. When applying T_{CF} to the program from Fig. 1 employing the usual arithmetic interpretation we get the result depicted in Tab. 1. As we can see, Constant Folding potentially produces dead code: the first instruction is dispensable since u is not used anymore until its redefinition in the sixth instruction.

For establishing the correctness of Constant Folding the particular interpretation has to be taken into account; thus strong equivalence is generally not preserved. We show in [7]:

Lemma 3.8. *For every $\pi = (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLC}$ and every interpretation \mathfrak{A} of Σ , $\pi \sim_{\mathfrak{A}} T_{CF}(\pi)$ and $T_{CF}(T_{CF}(\pi)) = T_{CF}(\pi)$.*

4 DAG Optimization

The DAG optimization for SLC–programs is based on the construction of a *directed acyclic graph* (DAG). A basic version of this optimization has been introduced in [3]. We will present a modified version which, however, does not consider the register allocation problem since we focus on intermediate code.

Definition 4.1 (DAG). *A DAG is an acyclic graph $G = (K, L, lab, suc)$ with a set of nodes K and a set of labels L , a labeling function $lab : K \rightarrow L$ and a partially defined successor function $suc : \subseteq K \times \mathbb{N} \rightarrow K$.*

A DAG represents the result and the operands of expressions by (different) nodes that are linked by the successor function.

The DAG of an SLC–Program

The DAG of an SLC–program is a graphical representation of the program with *sharing* of identical subterms. Furthermore a partial evaluation of expressions (similar to Constant Folding) is performed (extending the algorithm in [3]).

In addition to the DAG we need a valuation function $val : \subseteq V_{\pi} \times \mathbb{N} \rightarrow K$ with $val(x, i) = k$ iff the subgraph rooted at k represents the value of the variable x after i computation steps.

Algorithm 4.2 (DAG Construction). *Let $\pi := (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLC}$ with $\Sigma = (F, C)$, $\beta = \alpha_1; \dots; \alpha_n$ and $\mathfrak{A} = (A, \varphi)$. The DAG G and the valuation function val are inductively constructed as follows where the set of labels is defined by $L := F \cup V_{in} \cup A$.⁴*

Select $K := V_{in} \cup \varphi(C_{\pi})$ with $lab(k) = k \ \forall k \in K$ as initial nodes⁵ and set $val(x, 0) := x$ for all $x \in V_{in}$ where $\varphi(C_{\pi}) := \{\varphi(c) \mid c \in C_{\pi}\}$.

Assuming that G and val are already constructed for $\alpha_1; \dots; \alpha_i$ and letting $\alpha_{i+1} = x \leftarrow e$, we distinguish different cases depending on the type of the expression e :

⁴ Nodes that represent complex expressions will be labeled by the corresponding function symbols whereas variable/constant nodes will be labeled by themselves.

⁵ Alternatively one could add the constants later “on demand”.

1. $e = y \in V$:

According to the induction hypothesis, $val(y, i) \in K$ is already representing the current value of y . Thus G is not extended; set

$$val(x, i + 1) := val(y, i) \quad \text{and} \quad val(x', i + 1) := val(x', i) \quad \text{for } x' \neq x$$

In the following **update**($x, i + 1, y$) will be used to abbreviate the above two assignments.

2. $e = c \in C$:

$\varphi(c) \in K$ already exists. Therefore only: **update**($x, i + 1, \varphi(c)$).

3. $e = f(u_1, \dots, u_r)$, $u_j \in V \cup C$, $f \in F^{(r)}$. We distinguish further subcases:

(a) for all $j \in \{1, \dots, r\}$, $u_j \in C$ or ($u_j \in V$ and $val(u_j, i) \in A$).

Then let $a := \varphi(f)(u'_1, \dots, u'_r) \in A$ with

$$u'_j := \begin{cases} \varphi(u_j) & \text{if } u_j \in C \\ val(u_j, i) & \text{if } u_j \in V \end{cases}$$

– $a \in \varphi(C_\pi)$: no extension of G , set **update**($x, i + 1, a$).

– $a \notin \varphi(C_\pi)$: $K := K \cup \{a\}$ and **update**($x, i + 1, a$).

(b) $\exists j \in \{1, \dots, r\}$ with $u_j \in V$ and $val(u_j, i) \notin A$.

– $\exists k \in K$ with $lab(k) = f$ and

$$suc(k, j) = \begin{cases} \varphi(u_j) & \text{if } u_j \in C \\ val(u_j, i) & \text{if } u_j \in V \end{cases}$$

No modification of G ; the value of e is already represented by k . Set **update**($x, i + 1, k$).

– Otherwise: insert a node k_{i+1} :

$$K := K \cup \{k_{i+1}\}$$

$$lab(k_{i+1}) := f$$

$$suc(k_{i+1}, j) := \begin{cases} \varphi(u_j) & \text{if } u_j \in C \\ val(u_j, i) & \text{if } u_j \in V \end{cases}$$

$$\mathbf{update}(x, i + 1, k_{i+1})$$

Example 4.3. Figure 2 shows the DAG of the program from Fig. 1. The square nodes are the nodes already present before processing the instructions. The others were created later according to the above definition. For better clarity the *val*-table only shows those entries that represent changes.

The DAG construction incorporates aspects of Common Subexpression Elimination (node sharing for expressions already represented by a node in the DAG) and Constant Folding (partial evaluation of expressions based on constant information, i.e. no node represents a constant expression).

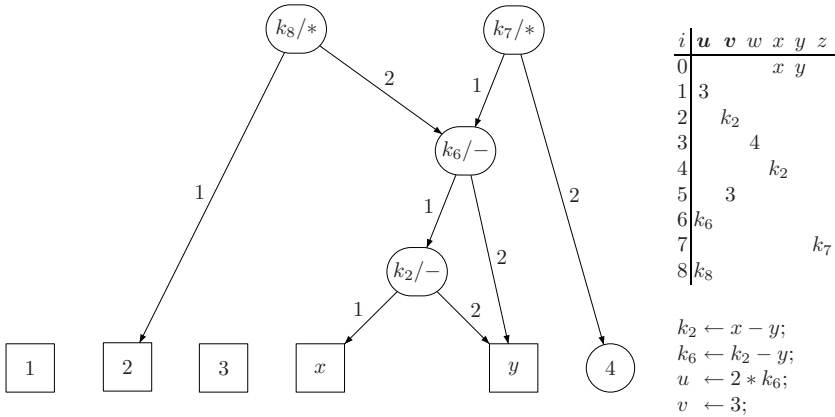


Fig. 2. DAG, *val*-function and optimized program for π from Fig. 1

Code Generation from a DAG

For generating code from a DAG only the nodes that are reachable from an “output node” are required. We will call these nodes *output relevant*. The others are not considered during code generation, thus implementing Dead Code Elimination.

With regard to the processing order of the nodes the following restriction has to be observed. Before creating an instruction for a node k all successors of k have to be processed first. In the following we will present a simple *nondeterministic* algorithm for code generation using our example.

Example 4.4. Let π be the program from Fig. 1 and G its DAG which is depicted in Fig. 2. Then code generation can be done as follows:

1. The set of output-relevant nodes is $\{2, 3, x, y, k_2, k_6, k_8\}$, as they are all reachable from k_8 (except 3 which itself is output-relevant).
2. Since the constants and input values are immediately available an operation node is the first node to process. This can only be k_2 because the other operation nodes depend on it. The instruction $k_2 \leftarrow x - y$ is created (according to the node label and the node’s successors).
3. The next node to process is k_6 because the node k_8 depends on it. For k_6 we obtain $k_6 \leftarrow k_2 - y$.
4. For the last remaining operation node k_8 we add $u \leftarrow 2 * k_6$. Here we do not use a temporary assignment variable but the output variable u because this will be the final value of u . Otherwise we would have to insert a copy instruction later on.
5. Now all the operation nodes are processed but the output variable v is still undefined. Since $val(v, 8) = 3$ we have to add the instruction $v \leftarrow 3$.

Thus we get the result shown in Fig. 2.

This “naive” code generation technique has several disadvantages increasing the complexity of the correctness proof:

- The algorithm is nondeterministic, multiple choices for the next node to process are possible. Hence the output depends on the node ordering, and thus the order of the assignments may differ between the input and the optimized program. (In our example, the order of the assignments to the variables u and v is reversed.)
- The idempotency is violated because after eliminating output-irrelevant nodes in the first application of T_{DAG} , the second application will introduce new node names.

In [7] an extended algorithm avoiding the above problems is introduced. It works similarly and will be denoted by T_{DAG} in the sequel.

In programs obtained from the DAG code generation for every assignment a new, previously undefined variable is used; this normal form is called *Static Single Assignment (SSA) form* in the literature.

Note that if the number of available registers is limited, the optimal code generation from a DAG is NP-complete [2]. The code generation from expression trees, however, is also in this case efficient [1].

Intuitively it is clear that the DAG algorithm is working correctly. Proving this correctness, however, is not trivial. We show in [7]:

Theorem 4.5. *Let $\pi = (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLLC}$ and \mathfrak{A} be an interpretation of Σ . Then $\pi \sim_{\mathfrak{A}} T_{DAG}(\pi)$ and $T_{DAG}(T_{DAG}(\pi)) = T_{DAG}(\pi)$.*

5 Composing the Simple Transformations

After formally introducing the different optimizing program transformations we will analyze the relations between them. We call two optimizations equivalent if every transformation is optimal with respect to the other:

Definition 5.1. *Let $T_1, T_2 : \mathcal{SLLC} \rightarrow \mathcal{SLLC}$ be program transformations.*

- T_2 is called T_1 -optimal ($T_1 \leq T_2$) iff $\forall \pi \in \mathcal{SLLC} : T_1(T_2(\pi)) = T_2(\pi)$.
- T_1 and T_2 are called equivalent ($T_1 \sim T_2$) iff $T_1 \leq T_2$ and $T_2 \leq T_1$.

In previous examples we have seen that the DAG algorithm has a higher “optimizing potential” than the classical transformations. A generalization of this observation yields (proven in [7]):

Theorem 5.2. *$T \leq T_{DAG}$ for every $T \in \{T_{DC}, T_{CS}, T_{CF}\}$.*

In the following we will examine the reverse direction, that is, the question whether the classical transformations can be applied in a certain order such that the result is equivalent to the DAG-optimized program.

Copy Propagation

Our first observation is that the classical transformations alone do not suffice for “simulating” the DAG optimization.

Theorem 5.3. *There exists a program $\pi \in \mathcal{SLC}$ such that $\pi = T(\pi)$ for every $T \in \{T_{DC}, T_{CS}, T_{CF}\}$, but $\pi \neq T_{DAG}(\pi)$.*

Proof. Consider the program depicted in Fig. 3. It is optimal w.r.t. all three classical transformations. An application of T_{DAG} , however, produces a much shorter version. \square

In the example Dead Code Elimination would yield an optimizing effect if one would replace e.g. the occurrences of u on the right-hand side of the instructions 2 and 3 by x . For this problem we will now introduce a new algorithm, *Copy Propagation*, which is no direct improvement but a *pre-processing* step enabling other optimizations.

Our version of Copy Propagation does not only substitute a variable used after a copy instruction with its original variable but also traces back transitive dependencies for copy chains. During the analysis step we collect the *valid copies* for each instruction. These are basically pairs of variables that have the same value at a given point (due to copy instructions). A third value – the *transitive depth* which represents the length of a copy chain – is used to guarantee determinism and idempotency of the transformation. Copy Propagation fulfills the requirements of a program transformation⁶.

$$\begin{aligned} & \mathbf{v}_{in} : (x, y) \\ & \beta : u \leftarrow x; \\ & \quad y \leftarrow u + y; \\ & \quad v \leftarrow u; \\ & \quad u \leftarrow v * y; \\ & \quad v \leftarrow u + v; \\ & \mathbf{v}_{out} : (u, v) \\ & \\ & \quad \downarrow T_{DAG} \\ & v_1 \leftarrow x + y; \\ & u \leftarrow x * v_1; \\ & v \leftarrow u + x; \end{aligned}$$

Fig. 3. Program with copy instructions and its DAG-optimized version

Definition 5.4 (Valid Copies). *Let $\pi := (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta := \alpha_1; \dots; \alpha_n$. For an instruction $\alpha = x \leftarrow e$ we define the transfer function $t_\alpha : 2^{V_\pi^2 \times \{1, \dots, n\}} \rightarrow 2^{V_\pi^2 \times \{1, \dots, n\}}$ by*

$$\begin{aligned} t_{x \leftarrow e}(M) := & \text{trans}(M \setminus \{(y, z, d) \in M \mid d \in \{1, \dots, n\}, x \in \{y, z\}\} \\ & \cup \{(x, e, 1) \mid e \in V_\pi \setminus \{x\}\}) \end{aligned}$$

Here $\text{trans} : 2^{V^2 \times \mathbb{N}} \rightarrow 2^{V^2 \times \mathbb{N}}$ computes the transitive closure of the argument relation, taking the transitive depth into account. Now the analysis sets $CP_i \subseteq 2^{V_\pi^2 \times \{1, \dots, n\}}$ can be computed inductively:

$$CP_1 := \emptyset \quad \text{and} \quad CP_{i+1} := t_{\alpha_i}(CP_i) \quad \text{for } i \in \{1, \dots, n-1\}$$

Based on the CP -sets we define the program transformation:

⁶ Proven in [7].

Definition 5.5 (Copy Propagation). *The transformation $T_{CP} : \mathcal{SLC} \rightarrow \mathcal{SLC}$ for $\pi = (\Sigma, V_{in}, V_{out}, \beta) \in \mathcal{SLC}$ with $\beta = x_1 \leftarrow e_1; \dots; x_n \leftarrow e_n$ and $\Sigma = (F, C)$ is given by*

$$T_{CP}(\pi) := (\Sigma, \mathbf{v}_{in}, \mathbf{v}_{out}, \beta') \text{ with } \beta' := x_1 \leftarrow \overline{CP}_1(e_1); \dots; x_n \leftarrow \overline{CP}_n(e_n)$$

where $\overline{CP}(e)$ with $CP \subseteq V_\pi^2 \times \{1, \dots, n\}$, $c \in C$, $x \in V_\pi$ and $f \in F^{(r)}$ is defined as follows:

$$\begin{aligned} \overline{CP}(c) &:= c \\ \overline{CP}(x) &:= \begin{cases} y & \text{if } \exists y \in V, \exists d \in \{1, \dots, n\} \text{ with } (x, y, d) \in CP \\ & \text{and } \forall (x, y', d') \in CP : d' \leq d \\ x & \text{else} \end{cases} \\ \overline{CP}(f(u_1, \dots, u_r)) &:= f(\overline{CP}(u_1), \dots, \overline{CP}(u_r)) \end{aligned}$$

The selection of the tuple with the highest transitive depth for the substitution ensures that the copy chains are traced back completely and that the algorithm works deterministically.

Example 5.6. Applying Copy Propagation to the program of Fig. 3 yields:

i	α_i	CP_i	new instruction α'_i
1	$u \leftarrow x;$	\emptyset	$u \leftarrow x;$
2	$y \leftarrow u + y;$	$\{(u, x, 1)\}$	$y \leftarrow x + y;$
3	$v \leftarrow u;$	$\{(u, x, 1)\}$	$v \leftarrow x;$
4	$u \leftarrow v * y;$	$\{(u, x, 1), (v, u, 1), (\mathbf{v}, \mathbf{x}, \mathbf{2})\}$	$u \leftarrow x * y;$
5	$v \leftarrow u + v$	$\{(v, x, 2)\}$	$v \leftarrow u + x;$

In instruction 4 the tuple printed in boldface results from the computation of the transitive closure. The transitive depth is needed to decide which substitution to use for v in instruction 4.

Execution Order

Now we will analyze the relations between the simple algorithms. Of particular interest is the following property:

Definition 5.7. *Let $T_i : \mathcal{SLC} \rightarrow \mathcal{SLC}$, $i \in \{1, 2\}$ be two program transformations. If there exists a $\pi \in \mathcal{SLC}$ such that*

$$T_1(\pi) = \pi \text{ and } T_1(T_2(\pi)) \neq T_2(\pi)$$

then T_2 is called T_1 -enabling ($T_2 \rightarrow T_1$).

Thus intuitively a transformation enables another if it “produces” additional optimization potential w.r.t. the other. The enabling relationships holding for our transformations are given in Fig. 4. Here an arrow means that the transformation labeling the row enables the transformation indexing the column, whereas a dash indicates the absence of an enabling effect.

	T_{DC}	T_{CS}	T_{CF}	T_{CP}
T_{DC}	-	-	-	-
T_{CS}	-	-	-	\rightarrow
T_{CF}	\rightarrow	\rightarrow	-	-
T_{CP}	\rightarrow	\rightarrow	-	-

Fig. 4. Enabling relations

- T_{DC} does not enable any of the other transformations because it eliminates instructions and therefore does not create new available expressions, copy instructions, or constant variables.
- T_{CS} enables Copy Propagation due to the insertion of copy instructions. It has no influence on Constant Folding and Dead Code Elimination.
- We have already seen in Ex. 3.7 that Constant Folding “produces” additional Dead Code. One can easily construct examples in which the substitution of variables by constants creates common subexpressions.
- From Ex. 5.6 it is clear that T_{CP} is T_{DC} -enabling. Similarly to Constant Folding the substitution of variables by others can create common subexpressions.

Between Copy Propagation and Common Subexpression Elimination there is a mutual dependence. Thus a repeated application of both is generally unavoidable.

Lemma 5.8. *There exists a sequence $(\pi_n)_{n \in \mathbb{N} \setminus \{0\}}$ such that $T := T_{CP} \circ T_{CS}$ has to be applied at least n times to π_n for reaching a fixed point.*

Proof. $\pi_n := (\Sigma, (x), (y, z), \beta_n)$ with $\Sigma = (F, \emptyset)$ and $F = \{f^{(2)}\}$ where β_n is given by

$$\begin{array}{ll} \beta_1 := y \leftarrow f(x, x); & \beta_{n+1} := \beta_n; \\ z \leftarrow f(x, x); & y \leftarrow f(y, x); \\ & z \leftarrow f(z, x); \end{array}$$

obviously fulfills the requirement. □

The maximum number of iterations is bounded by the number of instructions since every application of Common Subexpression Elimination that provokes a change reduces the number of operation expressions of the program, and since the number of operation expressions is limited by the number of instructions (see also [7]).

In practice it is certainly not advisable to “blindly” use the worst-case iteration number; rather a demand-driven method should be employed.

Definition 5.9. *For $\pi \in \mathcal{SLC}$ and $T := T_{CP} \circ T_{CS}$ the transformation $T_{CPCS} : \mathcal{SLC} \rightarrow \mathcal{SLC}$ is defined by:*

$$T_{CPCS}(\pi) := \begin{cases} \pi & \text{if } T(\pi) = \pi \\ T_{CPCS}(T(\pi)) & \text{else} \end{cases}$$

From the previous observations as represented in Tab. 4, we can now derive an order for the simple transformations to achieve a good optimization effect:

1. Constant Folding (cannot be enabled by the other transformations)
2. Application of Common Subexpression Elimination and Copy Propagation in alternation (T_{CPCS})
3. Dead Code Elimination (enabled by T_{CP} and T_{CF})

Table 2. Application of T_{COPT} to π from Fig. 1

$T_{CF}(\pi)$	$T_{CS}(T_{CF}(\pi))$	$T_{CP}(T_{CS}(T_{CF}(\pi)))$	$T_{DC}(T_{CP}(T_{CS}(T_{CF}(\pi))))$
$u \leftarrow 3;$	$u \leftarrow 3;$	$u \leftarrow 3;$	
$v \leftarrow x - y;$	$t_2 \leftarrow x - y;$	$t_2 \leftarrow x - y;$	$t_2 \leftarrow x - y;$
	$v \leftarrow t_2;$	$v \leftarrow t_2;$	
$w \leftarrow 4;$	$w \leftarrow 4;$	$w \leftarrow 4;$	
$x \leftarrow x - y;$	$x \leftarrow t_2;$	$x \leftarrow t_2;$	
$v \leftarrow 3;$	$v \leftarrow 3;$	$v \leftarrow 3;$	$v \leftarrow 3;$
$u \leftarrow x - y;$	$u \leftarrow x - y;$	$u \leftarrow t_2 - y;$	$u \leftarrow t_2 - y;$
$z \leftarrow u * 4;$	$z \leftarrow u * 4;$	$z \leftarrow u * 4;$	
$u \leftarrow 2 * u;$	$u \leftarrow 2 * u;$	$u \leftarrow 2 * u;$	$u \leftarrow 2 * u;$

Definition 5.10. The compositional transformation incorporating the classical optimizations and Copy Propagation is given by $T_{COPT} := T_{DC} \circ T_{CPCS} \circ T_{CF}$.

Example 5.11. Table 2 shows an exemplary computation of T_{COPT} for our example program from Fig. 1 starting from the T_{CF} -optimized program from Ex. 3.7. Already one application of $T_{CP} \circ T_{CS}$ suffices here to reach the fixed point (this is clear because in $T_{CP}(T_{CS}(T_{CF}(\pi)))$ all operation expressions are distinct).

The resulting program is identical to the DAG-optimized program from Fig. 2 except for variable naming and the instruction order⁷. Also for the example from Fig. 3 we would get the same results “modulo” variable names.

Simulation of the DAG Transformation

For achieving a T_{DAG} -optimal transformation we obviously need to rename the variables in the output program. This can be done via a variable renaming transformation. The T_{DAG} -equivalence is even then not fulfilled for the compositional T_{COPT} -optimization, there remain two minor issues:

- Under some circumstances Copy Propagation does not allow the propagation of a variable name even though the old assignment is still valid. This problem can be circumvented by transforming the input program in SSA form (T_{SSA} is formally defined in [7]) before applying T_{COPT} .
- Copy Assignments to output variables cannot be removed using the previously introduced transformations. They occur especially due to Common Subexpression Elimination. A specialized algorithm (T_{RC} ; see [7]) solves this problem.

Finally we obtain an extended compositional transformation:

Definition 5.12. For $\pi \in SLC$ the extended compositional transformation $T_{XOPT} : SLC \rightarrow SLC$ is defined by $T_{XOPT} := T_{SSA} \circ T_{RC} \circ T_{COPT} \circ T_{SSA}$.

⁷ The advanced DAG-algorithm does not have the reordering issue.

The final application of T_{SSA} is only required to ensure a variable naming “compatible” to the advanced DAG algorithm. Approximately we have $T_{XOPT} \approx T_{COPT}$ and according to [7]:

Theorem 5.13. $T_{XOPT} \sim T_{DAG}$.

6 Conclusion and Future Work

We have shown that the DAG procedure can essentially be characterized as a combination of Copy Propagation and the three classical transformations. More concretely it corresponds to a repeated application of Common Subexpression Elimination and Copy Propagation in alternation, preceded by Constant Folding and followed by Dead Code Elimination:

$$T_{DAG} \approx T_{DC} \circ (T_{CP} \circ T_{CS})^* \circ T_{CF}.$$

Apart from its theoretical importance, this result is also relevant for practical compiler design as it potentially allows to exploit the optimization potential of the DAG-based algorithm for non-linear code as well. Certainly our results cannot be transferred directly to iterative code, but the basic composition of the transformations should turn out to be effective. This matter is the main point for future investigation.

Furthermore it would be interesting to analyze whether Copy Propagation and Common Subexpression Elimination can be merged into one algorithm to avoid the iterative application procedure.

References

1. Aho, A.V., Johnson, S.C.: Optimal code generation for expression trees. *J. ACM* 23(3), 488–501 (1976)
2. Aho, A.V., Johnson, S.C., Ullman, J.D.: Code generation for expressions with common subexpressions. *J. ACM* 24(1), 146–160 (1977)
3. Aho, A.V., Sethi, R., Ullman, J.D.: A formal approach to code optimization. *ACM SIGPLAN Notices* 5(7), 86–100 (1970)
4. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, London, UK (1986)
5. Ibarra, O.H., Leinger, B.S.: The complexity of the equivalence problem for straight-line programs. In: *STOC’80. Proc. of the 12th Annual ACM Symp. on Theory of Computing*, pp. 273–280. ACM Press, New York, NY, USA (1980)
6. Nielson, F., Nielson, H.R., Hankin, C.: *Principles of Program Analysis*. Springer, Heidelberg (1999)
7. Noll, T., Rieger, S.: Optimization of straight-line code revisited. Technical Report 2005–21, RWTH Aachen University, Dept. of Computer Science, Germany (2005), <http://sunsite.informatik.rwth-aachen.de/Publications/AIB/>
8. Rieger, S.: SLC Optimizer - Web Interface (2005), <http://aprove.informatik.rwth-aachen.de/~rieger/>