RWTH Aachen

Department of Computer Science
Technical Report

Optimization of Straight-Line Code Revisited

Thomas Noll and Stefan Rieger

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2005-21

RWTH Aachen · Department of Computer Science · November 2005

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Optimization of Straight–Line Code Revisited

Thomas Noll and Stefan Rieger

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

Abstract. In this report 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 such 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 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 report we compare certain 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 [ASU70], and the authors claim that it produces optimal results[1] 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.

In this paper we first present a slightly modified version of the DAG algorithm, denoted by T_{DAG}, which in addition supports constant folding. We then show that it integrates the following three classical transformations:

Constant Folding (T_{CF}), which corresponds to a partial evaluation of the program with respect to a given interpretation of its constant and operation symbols;

Common Subexpression Elimination (T_{CS}), which aims to decrease the execution time of the program by avoiding multiple evaluations of the same expression; and

Dead Code Elimination (T_{DC}), which removes computations that do not contribute to the actual result of the program.

[1] The optimality is given w.r.t. strong equivalence, see Sec. 2.4.

It will then 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. In fact we will show that the DAG procedure can essentially be characterized as a combination of Copy Propagation and the first three transformations. ("Essentially" here means that some additional minor modifications are required to make the two resulting programs syntactically equal.)

More concretely we will show that, under the above restriction, the DAG algorithm corresponds to a repeated application of Common Subexpression Elimination and Copy Propagation in alternation, preceded by Constant Folding and followed by Dead Code Elimination. Formally this relation can be depicted as follows:

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

Apart from its theoretical importance, this result is also relevant for practical compiler design as it allows to exploit the optimization potential of the DAG–based algorithm for non–linear code as well.

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 first three classical transformations and of the DAG–based algorithm in Sec. 3 and 4, respectively. The following Sec. 5 constitutes the main part of this report, establishing the equivalence between the DAG procedure and the composition of the classical transformations.

To support the experimenting with concrete examples, also a web–based implementation of the optimizing transformations is available at the web page [Rie05a].

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.

2.1 Syntax

An SLC–program consists of a sequence of assignments using simple arithmetic expressions without branchings or loops, a vector of input variables whose values are initialized before program execution and a vector of output variables whose values form the "result" of the computation after program termination.

For the formal description of the syntax of such programs we need some preliminary definitions.

Definition 2.1 (Signature). 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 (not necessarily finite[2]) set of *constant symbols* C.

Furthermore let $V := \{x, y, z, ...\}$ be the (infinite) set of *all* variables.

Example 2.2. $\Sigma_{\text{arithm}} = (\{+, *, -\}, \mathbb{Z})$ with $+, * \in F^{(2)}$ and $- \in F^{(1)}$ is an example of a signature. A simplified notation for this is $\Sigma_{\text{arithm}} = (\{+^{(2)}, *^{(2)}, -^{(1)}\}, \mathbb{Z})$, in which for each operator the arity is given as a superscript. Observe that $\mathbb{Z}$ in this case contains only constant *symbols* whose interpretation can differ from the usual one (see also Sec. 2.2).

Now we can define the syntax of an SLC–program:

Definition 2.3 (SLC–Program). An *SLC–program* is a quadruple $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ with

– a signature $\Sigma = (F, C)$,
– a vector of pairwise distinct *input variables* $\vec{v}_{in} = (x_1, ..., x_s),\ x_i \in V$ and
– a vector of pairwise distinct *output variables* $\vec{v}_{out} = (y_1, ..., y_t),\ y_i \in V$, and
– a *block* $\beta = \alpha_1; \alpha_2; ...; \alpha_n$ with *instructions* α_i of the form $x \leftarrow e$ where $x \in V$ and $e \in V \cup C \cup \{f(u_1, ..., u_r) \mid f \in F^{(r)}$ and $\forall j \in \{1, .., r\} : u_j \in V \cup C\}$. It is not allowed to assign a variable to itself directly or indirectly via other variables (using copy instructions of the form $x \leftarrow y$ where $x, y \in V$).
– In correspondence to $\vec{v}_{in}$ and $\vec{v}_{out}$ let $V_{in} := \{x_1, ..., x_s\}$ and $V_{out} := \{y_1, ..., y_t\}$ denote the *sets* of input/output variables.
– $V_{in} \cap V_{out} = \emptyset$ must hold[3].

For an SLC–program π we introduce the following denotations:

– C_π is the set of constant symbols occurring in π,
– V_π the set of variables occurring in in π,
– V_α the set of the variables occurring in the instruction α and
– V_e the set of the variables in the expression e.

Finally we let $\mathcal{SLC}$ denote the set of all SLC–programs.

Example 2.4. Figure 1 shows a simple SLC–program with signature $\Sigma_{\text{arithm}'} := (\{+^{(2)}, *^{(2)}, -^{(2)}\}, \mathbb{Z})$. Deviating from the above definition we employed the usual infix notation. Later we will often use infix notation when we refer to the "classical arithmetic".

From now on we will always assume *completeness* for SLC–programs:

Definition 2.5 (Completeness). $\pi \in \mathcal{SLC}$ is called *complete* if every variable is defined before being used[4]. Formally: let $DV_i \subseteq V_\pi$ denote the defined variables up to instruction α_i. These sets can be computed as follows:

$$DV_0 := V_{in}$$
$$DV_i := DV_{i-1} \cup \{x \in V \mid \alpha_i = x \leftarrow e\} \text{ for } i \in \{1, ..., n\}$$

Now the following holds: π is complete iff

[2] In the program only finitely many of these can occur.
[3] This is necessary for the correctness of the DAG optimization (Sec. 4.3).
[4] The occurrence of a variable in $\vec{v}_{in}/\vec{v}_{out}$ corresponds to a definition/use, respectively.

$$
\begin{aligned}
\vec{v}_{in} :&\ (x, y)\\
\beta :&\ u \leftarrow 3;\\
&\ v \leftarrow x - y;\\
&\ w \leftarrow u + 1;\\
&\ x \leftarrow x - y;\\
&\ v \leftarrow w - 1;\\
&\ u \leftarrow x - y;\\
&\ z \leftarrow u * w;\\
&\ u \leftarrow 2 * u;\\
\vec{v}_{out} :&\ (u, v)
\end{aligned}
$$

Fig. 1. SLC–program $\pi = (\Sigma_{\mathrm{arithm}'}, \vec{v}_{in}, \vec{v}_{out}, \beta)$

- if $\alpha_i = x \leftarrow e$ then $V_e \subseteq DV_{i-1}$ for all $i \in \{1, ..., n\}$ and
- $V_{out} \subseteq DV_n$.

From the definition one can construct a simple iterative algorithm for testing completeness of SLC–programs. Only for complete programs it can be ensured that the result depends only on the input and, thus, that the program is deterministic. The completeness of the program π in Fig. 1 is obvious.

2.2 Semantics

So far we only dealt with the structure and not with the meaning of SLC–programs. The semantics of an SLC–program depend on the domain of the variables as well as the interpretation of the operators and constant symbols.

Definition 2.6 (Interpretation). An *interpretation* of a signature $\Sigma = (F, C)$ is a Σ–algebra $\mathfrak{A} := (A, \varphi)$ with domain (universe) A and interpretation function

$$
\begin{aligned}
&\varphi : F \cup C \cup A \rightarrow \bigcup_{i=0}^{\infty} \{\delta \mid \delta : A^i \rightarrow A\} \text{ with}\\
&\varphi(f) : A^r \rightarrow A \text{ for every } f \in F^{(r)}\\
&\varphi(c) \in A \text{ for every } c \in C\\
&\varphi(a) = a \text{ for every } a \in A
\end{aligned}
$$

Thus φ assigns to every function symbol f a function $\varphi(f)$ over A and to every constant symbol a value in A. For simplifying later notations values in A are mapped to themselves (this is e.g. useful for constant folding; see Sec. 3.3).

Example 2.7. For instance Σ_{arithm} could be interpreted by $\mathfrak{A} = (\mathbb{Z}, \varphi)$ where φ assigns the operators their "usual" meaning over $\mathbb{Z}$ and every constant *symbol* in $\mathbb{Z}$ itself as a constant *value*. A different interpretation could be the "Boolean arithmetic":

$$\mathfrak{A} = (\{0,1\}, \varphi) \text{ with } \begin{aligned}[t] &\varphi(+)(a,b) = a \vee b \\ &\varphi(*)(a,b) = a \wedge b \\ &\varphi(-)(a) = 1 - a \\ &\varphi(z) = \begin{cases} 0 & \text{if } z = 0 \\ 1 & \text{else} \end{cases} \quad \text{for } z \in \mathbb{Z} \end{aligned}$$

Intuitively a program can be understood as a function that maps a vector representing the input *values* (*input vector*) to a vector that contains the *values* of the output variables (*output vector*).

The current state at an arbitrary program position can be expressed as a mapping of variables to their values at this specific point. A state is therefore representable as a *valuation function* $\sigma : V_\pi \to A$.

Definition 2.8 (State Space). The *state space* of an SLC–program $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ with $\Sigma := (F, C)$ and interpretation $\mathfrak{A} := (A, \varphi)$ of Σ is given by

$$S := \{\sigma \mid \sigma : V_\pi \to A\}$$

From the semantic point of view every instruction α determines a transformation $t_\alpha : S \to S$ of one state into another. The program semantics are inductively based on this transformation.

Definition 2.9 (Semantics of an SLC–Program). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\vec{v}_{in} := (x_1, ..., x_s)$, $\vec{v}_{out} := (y_1, ..., y_t)$ and $\mathfrak{A} := (A, \varphi)$ an interpretation of Σ.

The input vector $\vec{in} := (in_1, ..., in_s) \in A^s$ determines the initial valuation

$$\sigma_0(v) := \begin{cases} in_i & \text{if } v = x_i \\ a & \text{else} \end{cases}$$

with arbitrary[5] $a \in A$.

The semantics $\mathfrak{A}[\![\pi]\!]$ of π w.r.t. $\mathfrak{A}$ can now be inductively defined as follows:

- The semantics $\mathfrak{A}[\![\alpha]\!] : S \to S$ of an instruction α are[6] $\mathfrak{A}[\![\alpha]\!]\sigma := \sigma[x/\mathfrak{A}[\![e]\!]\sigma]$. For $\mathfrak{A}[\![e]\!]\sigma$ we distinguish the following cases:
  1. $e = y \in V_\pi \quad \Rightarrow \quad \mathfrak{A}[\![e]\!]\sigma := \sigma(y)$
  2. $e = c \in C_\pi \quad \Rightarrow \quad \mathfrak{A}[\![e]\!]\sigma := \varphi(c)$
  3. $e = f(u_1, ..., u_r) \quad \Rightarrow \quad \mathfrak{A}[\![e]\!]\sigma := \varphi(f)(\mathfrak{A}[\![u_1]\!]\sigma, ..., \mathfrak{A}[\![u_r]\!]\sigma)$
- Given a block $\beta = \alpha_1; ...; \alpha_n$ we define the block semantics as[7] $\mathfrak{A}[\![\beta]\!] = \mathfrak{A}[\![\alpha_n]\!] \circ ... \circ \mathfrak{A}[\![\alpha_1]\!]$.
- The semantics $\mathfrak{A}[\![\pi]\!] : A^s \to A^t$ of π are then

$$\mathfrak{A}[\![\pi]\!](\vec{in}) = ((\mathfrak{A}[\![\beta]\!]\sigma_0)(y_1), ..., \underbrace{(\mathfrak{A}[\![\beta]\!]\sigma_0)(y_j)}_{\text{value of the output variable } y_j}, ..., (\mathfrak{A}[\![\beta]\!]\sigma_0)(y_t))$$

[5] Because of the completeness of π non–input variables can be initialized arbitrarily.

[6] $[f[x/a](y) := \begin{cases} a & \text{if } y = x \\ f(y) & \text{else} \end{cases}$ is the modification of a function for an input value.

[7] $\circ$ is the function composition $(f \circ g)(x) = f(g(x))$.

We see that the semantics are defined independent of the variable names; only the order of the variables in the input/output vectors is relevant. Thus we have *functional semantics.*

Example 2.10. Let us examine the semantics for a short example program:

$$\pi_{\text{short}} := (\Sigma_{\text{arithm}}, (x), (y, z), \beta) \text{ with } \beta := \begin{array}{l} y \leftarrow -x; \\ z \leftarrow 2 * y; \end{array}$$

We interpret the operators $*$ and $-$ with their usual meanings on $\mathbb{Z}$. The program semantics for $in = (4)$ (i.e., $\sigma_0 = \{x \mapsto 4, y \mapsto 0, z \mapsto 0\}$) are computed as follows:

$$\begin{aligned}
\mathfrak{A}[\![\beta]\!]\sigma_0 &= \mathfrak{A}[\![z \leftarrow 2 * y]\!] \circ \mathfrak{A}[\![y \leftarrow -x]\!]\sigma_0 \\
&= \mathfrak{A}[\![z \leftarrow 2 * y]\!]\sigma_0[y/\mathfrak{A}[\![-x]\!]\sigma_0] \\
&= \mathfrak{A}[\![z \leftarrow 2 * y]\!]\sigma_0[y/\varphi(-)(\mathfrak{A}[\![x]\!]\sigma_0)] \\
&= \mathfrak{A}[\![z \leftarrow 2 * y]\!]\sigma_0[y/\varphi(-)(\sigma_0(x))] \\
&= \mathfrak{A}[\![z \leftarrow 2 * y]\!]\sigma_0[y/\varphi(-)(4)] \\
&= \mathfrak{A}[\![z \leftarrow 2 * y]\!]\underbrace{\sigma_0[y/-4]}_{=:\sigma_1} \\
&= \sigma_1[z/\mathfrak{A}[\![2 * y]\!]\sigma_1] \\
&= \sigma_1[z/\varphi(*)(\mathfrak{A}[\![2]\!]\sigma_1, \mathfrak{A}[\![y]\!]\sigma_1)] \\
&= \sigma_1[z/\varphi(*)(\varphi(2), \sigma_1(y))] \\
&= \sigma_1[z/\varphi(*)(2, -4)] \\
&= \sigma_1[z/-8] \\
&= \{x \mapsto 4, y \mapsto -4, z \mapsto -8\} \\
&= \{y \mapsto -4, z \mapsto -8\}
\end{aligned}$$

We obtain:

$$\mathfrak{A}[\![\pi_{\text{short}}]\!](4) = ((\mathfrak{A}[\![\beta]\!]\sigma_0)(y), (\mathfrak{A}[\![\beta]\!]\sigma_0)(z))(4) = (-4, -8)$$

Thus the "execution" of the program works "as expected", the variables on the left–hand side of an instruction are instantiated with the value resulting from the evaluation of the expressions on the right–hand side. For this evaluation the constant and function symbols of an expression have to be interpreted. Finally the last variable valuation determines the program output.

2.3 Term Representation of SLC–Programs

The computation determining the final value of an output variable can be represented as a *term*. This is especially useful for proofs.

Definition 2.11. We define the *set of terms* $T_\Sigma(X)$ for a signature $\Sigma = (F, C)$ and a set of variables $X \subseteq V$ inductively:

1. $C \subseteq T_\Sigma(X)$,
2. $X \subseteq T_\Sigma(X)$, and
3. $f(t_1, ..., t_r) \in T_\Sigma(X)$ if $t_1, ..., t_r \in T_\Sigma(X)$ and $f \in F^{(r)}$.

A term without variables $t \in T_\Sigma(\emptyset) =: T_\Sigma$ is also called a *ground term.*

Therefore also the expressions on the right–hand side of instructions are terms for which no nesting is allowed. For computing the term representation of an SLC–program it seems reasonable to symbolically execute the program by backward substituting the expression on the right–hand side of an assignment for the left–hand side.

Definition 2.12. The *term representation* $t_\pi(y) \in T_\Sigma(V_{in})$ *of an SLC–program* $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ *with* $\beta := \alpha_1; ...; \alpha_n$ *w.r.t. an output variable* $y \in V_{out}$ can be computed as follows:

$$\begin{aligned} t_\pi^{(n)}(y) &:= y \\ t_\pi^{(i-1)}(y) &:= t_\pi^{(i)}(y)[x/e] \text{ for } \alpha_i = x \leftarrow e \text{ and } i \in \{1, ..., n\} \\ t_\pi(y) &:= t_\pi^{(0)}(y) \end{aligned}$$

Here $[x/t] : T_\Sigma(X) \to T_\Sigma(X)$ denotes the substitution of every occurrence of the variable $x \in X$ with the term $t \in T_\Sigma(X)$.

During computation variables in the current term are successively replaced until finally only input variables remain. Substituting these with the input values yields a ground term.

Example 2.13. Now we will compute the term representation $t_\pi(u)$ of the program π from Fig. 1 w.r.t. the output variable u:

i	α_i	$t_\pi^{(i)}$
0		$t_\pi^{(1)}[u/3] = *(2, -(-(x, y), y))$
1	$u \leftarrow 3;$	$t_\pi^{(2)}[v/-(x, y)] = *(2, -(-(x, y), y))$
2	$v \leftarrow x - y;$	$t_\pi^{(3)}[w/+(u, 1)] = *(2, -(-(x, y), y))$
3	$w \leftarrow u + 1;$	$t_\pi^{(4)}[x/-(x, y)] = *(2, -(-(x, y), y))$
4	$x \leftarrow x - y;$	$t_\pi^{(5)}[v/-(w, 1)] = *(2, -(x, y))$
5	$v \leftarrow w - 1;$	$t_\pi^{(6)}[u/-(x, y)] = *(2, -(x, y))$
6	$u \leftarrow x - y;$	$t_\pi^{(7)}[z/*(u, w)] = *(2, u)$
7	$z \leftarrow u * w;$	$t_\pi^{(8)}[u/*(2, u)] = *(2, u)$
8	$u \leftarrow 2 * u;$	u

Thus we obtain $t_\pi(u) = *(2, -(-(x, y), y))$.

Using the term representation as defined above we can give an alternate definition of the semantics:

Lemma 2.14. *Let* $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ *with* $\Sigma := (F, C)$, $\vec{v}_{in} := (x_1, ..., x_s)$ *and* $\vec{v}_{out} := (y_1, ..., y_t)$. *Then it holds:*

$$\mathfrak{A}[\![\pi]\!](\vec{in}) = (\mathfrak{A}[\![t_\pi(y_1)[x_1/in_1, ..., x_s/in_s]]\!], ..., \mathfrak{A}[\![t_\pi(y_t)[x_1/in_1, ..., x_s/in_s]]\!])$$

where the (ground) term semantics $\mathfrak{A}[\![t]\!]$ *for a ground term* $t \in T_{\Sigma'}$ *and a signature* $\Sigma' = (F, C \cup A)$ *are given by:*

1. $\mathfrak{A}[\![a]\!] := a$ *for* $a \in A$

2. $\mathfrak{A}[\![c]\!] := \varphi(c)$ *for* $c \in C$
3. $\mathfrak{A}[\![f(t_1, ..., t_r)]\!] := \varphi(f)(\mathfrak{A}[\![t_1]\!], ..., \mathfrak{A}[\![t_n]\!])$ *for* $t_i \in T_{\Sigma'}$

One inserts simply the semantics of the term representations of π w.r.t. the output variables into the result vector where the input variables in the terms are replaced by the input values (this results in a term over the new signature Σ'). The term semantics are the successive application of the interpretations of the operators on the particular subterms (constants are interpreted by themselves and constant symbols mapped by φ). The correctness of this approach is obvious.

2.4 Equivalence

For optimizations program equivalence is of high significance since programs have to be transformed in a way such that their semantics are preserved. Otherwise the optimization algorithm would be incorrect.

Definition 2.15 (Equivalence). Two SLC–programs π_1 and π_2 over some signature Σ are called $\mathfrak{A}$*–equivalent* for an interpretation $\mathfrak{A}$ of Σ ($\pi_1 \sim_{\mathfrak{A}} \pi_2$) if $\mathfrak{A}[\![\pi_1]\!] = \mathfrak{A}[\![\pi_2]\!]$. If this holds for all interpretations $\mathfrak{A}$ then they are called *strongly equivalent* ($\pi_1 \sim \pi_2$).

Two equivalent programs are therefore computing the same function. It is evident that strong equivalence is a sufficient condition for (weak) equivalence. With the aid of the term representation we get a decidability result for strong equivalence:

Theorem 2.16. *Let* $\pi_i := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_i) \in \mathcal{SLC},\ i \in \{1, 2\}$ *with*[8] *w.l.o.g.* $\vec{v}_{in} = (x_1, ..., x_s)$ *and* $\vec{v}_{out} = (y_1, ..., y_t)$*. Then:*

$$\pi_1 \sim \pi_2 \quad \Leftrightarrow \quad \forall j \in \{1, ..., t\} : t_{\pi_1}(y_j) = t_{\pi_2}(y_j)$$

Proof. According to La. 2.14 the semantics of $\pi_i,\ i \in \{1, 2\}$ are:

$$\mathfrak{A}[\![\pi_i]\!](\vec{in}) = (\mathfrak{A}[\![t_{\pi_i}(y_1)[x_1/in_1, ..., x_s/in_s]]\!], ..., \mathfrak{A}[\![t_{\pi_i}(y_t)[x_1/in_1, ..., x_s/in_s]]\!]).$$

If $t_{\pi_1}(y_j) = t_{\pi_2}(y_j)$ holds for all $j \in \{1, ..., t\}$ then also $t_{\pi_1}(y_j)[x_1/in_1, ..., x_s/in_s] = t_{\pi_2}(y_j)[x_1/in_1, ..., x_s/in_s]$ for all $j \in \{1, ..., t\}$ due to the same input values for both programs.

The application of the semantics function $\mathfrak{A}[\![\cdot]\!]$ to two identical terms thus yields the same result (because the semantics are deterministic). □

For weak equivalence the above theorem is not valid:

Example 2.17. Let $\pi_i := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_i),\ i \in \{1, 2\}$ with $\vec{v}_{in} = (x, y)$, $\vec{v}_{out} = (z)$, $\beta_1 = z \leftarrow x + y$, and $\beta_2 = z \leftarrow y + x$.

We see that the term representations of π_1 and π_2 w.r.t. $z \in V_{out}$ are not identical, but that assuming arithmetic on $\mathbb{Z}$ yields the same semantics due to the commutativity[9] of $+_{\mathbb{Z}}$.

[8] According to Def. 2.9 the input/output vectors of the two programs must have equal length. Thus by renaming variables one can obtain the same naming.

[9] Exploiting such rules is part of algebraic optimizations that can be done during *peephole optimization* [McK65,TvSS82,DF84].

Therefore one cannot conclude that two programs with different term representations are not equivalent. In fact for arbitrary interpretations weak equivalence of SLC–programs is undecidable:

Theorem 2.18 (Undecidability of Equivalence). *Given two SLC–programs π_1 and π_2 over Σ and an interpretation $\mathfrak{A}$ of Σ it is generally undecidable whether $\pi_1 \sim_{\mathfrak{A}} \pi_2$.*

Proof. For $\pi := (\Sigma, \varepsilon, (y), \beta) \in \mathcal{SLC}$ with $\Sigma := (\{+, /\}, \{1\})$ where / denotes integer division one can show that the zero equivalence problem, i.e., the question whether $\mathfrak{A}[\![\pi]\!] = 0$ holds is undecidable.

This is done by reducing the problem to Hilbert's Tenth Problem which is undecidable [DMR76]. The detailed proof can be found in [IL80]. □

In special cases however, like the arithmetic on $\mathbb{Z}$ without division, one can obtain a positive result:

Theorem 2.19. *Let $\Sigma := (F, \mathbb{Z})$ with $F := \{+^{(2)}, -^{(2)}, *^{(2)}\}$, $\mathfrak{A} := (\varphi, \mathbb{Z})$, $\varphi(\circ) := \circ_{\mathbb{Z}}$ for $\circ \in F$ and $\varphi(z) = z$ for all $z \in \mathbb{Z}$. Additionally let $\pi_i := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_i) \in \mathcal{SLC}$ for $i \in \{1, 2\}$ with $\vec{v}_{in} = (x_1, ..., x_s)$, $\vec{v}_{out} = (y_1, ..., y_t)$ and $V_{\pi_1} = V_{\pi_2}$. Then one can decide whether $\pi_1 \sim_{\mathfrak{A}} \pi_2$ holds.*

Proof. Both programs are transformed in a sequence of polynomials $P_j^{(i)} := P_j^{(i)}(x_1, ..., x_s)$ for $j \in \{1, ..., t\}$. By converting these to a normal form and comparing the coefficients one can determine if the programs are equivalent.

The algorithm works in the following steps:

1. For $i \in \{1, 2\}$ and $j \in \{1, ..., t\}$ compute the term representation $t_{\pi_i}(y_j)$ of π_i w.r.t. y_j.
2. Convert $t_{\pi_i}(y_j)$ for $i \in \{1, 2\}$ and $j \in \{1, ..., t\}$ into polynomials. For a term $t \in T_\Sigma(X)$ we will denote the corresponding polynomial by $P[t]$.
3. Transform $P[t_{\pi_i}(y_j)]$ for all $i \in \{1, 2\},\ j \in \{1, ..., t\}$ into the normal form
$$P_j^{(i)}(x_1, ..., x_s) := \sum_{k_1=0}^{n} \sum_{k_2=0}^{n} ... \sum_{k_s=0}^{n} c_{k_1 k_2 ... k_s}^{(i,j)} \cdot x_1^{k_1} \cdot x_2^{k_2} \cdot ... \cdot x_s^{k_s}$$
by multiplicative expansion. The length[10] of this representation is $O(n^s)$, i.e., it is exponential in the number of input variables. Therefore also the transformation has at least this complexity.
4. Test if $c_{k_1 ... k_s}^{(1,j)} = c_{k_1 ... k_s}^{(2,j)}$ for all $j \in \{1, ..., t\},\ k_l \in \{1, ..., n\},\ l \in \{1, ..., s\}$.
If the above holds the programs are equivalent, otherwise not. □

Because of its high complexity the algorithm is practically unusable. As per [IM83], however, the problem is probabilistically solvable in polynomial time (this is also the case for the arithmetic over $\mathbb{R}$).

[10] Number of summands

2.5 Cost and Optimality

We need certain criteria for comparing programs and for assessing the quality of an optimization.

Definition 2.20 (Cost Function). A function $c : \mathcal{SLC} \rightarrow \mathbb{R}_0^+$ is called *cost function* if for two SLC–programs $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; ...; \alpha_n)$ and $\pi' := (\Sigma, \vec{v}'_{in}, \vec{v}'_{out}, \alpha_{i_1}; ...; \alpha_{i_k})$ with $1 \leq i_j \leq n$ and $i_j < i_{j+1}$ for $j \in \{1, ..., k-1\}$ the following holds:

$$c(\pi') \leq c(\pi) \quad \text{(monotonicity)}$$

As standard cost functions we will use c_l for the number of instructions of a program and c_{op} for the number of operations (where copy instructions are not counted). The monotonicity ensures that removing instructions from a program will never increase its cost value.

Determining the cost value of an SLC–program is quite simple because SLC–programs do not contain loops or branchings. In contrast, in iterative programs it is generally not decidable at compile time how often a loop will be traversed and thus the cost computation can be difficult or even impossible (depending on the particular cost function).

When deciding whether a program π is optimal we only consider programs that are equivalent to π:

Definition 2.21 (Optimality). $\pi \in \mathcal{SLC}$ is *optimal* w.r.t. a cost function c and an interpretation $\mathfrak{A}$ if all $\pi' \sim_{\mathfrak{A}} \pi$ have a higher cost, i.e., $c(\pi) \leq c(\pi')$. It is called *(strongly) c–optimal* if $c(\pi) \leq c(\pi')$ for every $\pi' \sim \pi$.

Thus for deciding program optimality it is necessary to decide program equivalence and hence the optimality of SLC–programs is undecidable. But for the special case of arithmetic on $\mathbb{Z}$ (without division) one can decide optimality.

The idea is to enumerate all SLC–programs in the order of their cost and then to check the equivalence. For this we need another property of cost functions: $|c^{-1}(r)| < \infty, \ \forall r \in \mathbb{R}$ (for each cost value only finitely many programs may exist[11]). The program length c_l fulfills this property only if the number of constant symbols $|C|$ is finite. The cost function c_{op} never fulfills it (one could add infinitely many copy instructions).

It is clear that even for this special case an optimality test is practically infeasible. Therefore we will concentrate from now on on transformations that *improve* programs instead of really *optimizing* them. In the following we will nevertheless call these transformations optimizations for simplicity.

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 run in two phases:

[11] Up to the renaming of variables.

Analysis: Collection of information that are necessary for the transformation of the program. This is accomplished by defining a starting information that is modified for each instruction. The program remains unchanged during this step.

Transformation: Execution of the actual optimization; the program is modified using the analysis information of the first phase. This transformation must preserve equivalence.

Before introducing the algorithms we define some important properties that an optimizing transformation should satisfy:

Definition 3.1 (Program Transformation). A function $T : \mathcal{SLC} \to \mathcal{SLC}$ 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.*

Note that every ($\mathfrak{A}$–)program transformation is required to be a function and is deterministic therefore.

3.1 Dead Code Elimination

Dead Code Elimination removes instructions that are dispensable because they do not influence program semantics.[12] 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 $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta = \alpha_1; ...; \alpha_n$. For an instruction $\alpha = x \leftarrow e$ we define the transfer function $t_\alpha : \mathfrak{P}(V_\pi) \to \mathfrak{P}(V_\pi)$ as follows:

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

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

$$\begin{aligned} NV_n &:= V_{out} \\ NV_{i-1} &:= t_{\alpha_i}(NV_i) \text{ for } i \in \{n, ..., 2\} \end{aligned}$$

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

[12] The term *dead code* is also used to denote unreachable code in iterative programs.

Definition 3.3 (Dead Code Elimination). For $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta = \alpha_1; ...; \alpha_n$, the program transformation $T_{DC} : \mathcal{SLC} \to \mathcal{SLC}$ for eliminating dead code is given by:

$$T_{DC}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta') \text{ with } \beta' := t_{DC}(\alpha_1); ...; t_{DC}(\alpha_n)$$
$$t_{DC}(\alpha_i = x \leftarrow e) := \begin{cases} \alpha_i & \text{if } x \in NV_i \\ \varepsilon & \text{else} \end{cases}$$

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 (because the length of the input vector would be reduced).

Example 3.4. The application of the Dead Code Elimination to the example program from Fig. 1 yields:

i	α_i	NV_i	dead code?
1	$u \leftarrow 3;$	NV_2	No
2	$v \leftarrow x - y;$	$NV_3 \setminus \{w\} \cup \{u\} = \{u, x, y\}$	Yes
3	$w \leftarrow u + 1;$	$NV_4 \setminus \{x\} \cup \{x, y\} = \{w, x, y\}$	No
4	$x \leftarrow x - y;$	$NV_5 \setminus \{v\} \cup \{w\} = \{w, x, y\}$	No
5	$v \leftarrow w - 1;$	$NV_6 \setminus \{u\} \cup \{x, y\} = \{v, x, y\}$	No
6	$u \leftarrow x - y;$	NV_7	No
7	$z \leftarrow u * w;$	$NV_7 \setminus \{u\} \cup \{u\} = \{u, v\}$	Yes
8	$u \leftarrow 2 * u;$	$V_{out} = \{u, v\}$	No

Thus the instructions α_2 and α_6 are removed by T_{DC}.

The Dead Code Elimination is independent of program semantics, i.e., the interpretation has no effect on the result of the transformation.

Theorem 3.5 (Correctness). *For every $\pi \in \mathcal{SLC}$, $\pi \sim T_{DC}(\pi)$.*

Proof. We have to show that $\mathfrak{A}[\![T_{DC}(\pi)]\!] = \mathfrak{A}[\![\pi]\!]$ holds for every interpretation $\mathfrak{A}$ (alternatively a proof using Thm. 2.16 would be possible). Induction on the number of instructions $n \in \mathbb{N}$:

$n = 0$**:**
$\beta = \varepsilon$ so trivially $\mathfrak{A}[\![T_{DC}(\pi)]\!] = \mathfrak{A}[\![\pi]\!]$ holds and therefore also $\vec{v}_{out} = \varepsilon$ (otherwise the completeness of π would be violated).

$n \to n + 1$**:**
Induction Hypothesis: $\mathfrak{A}[\![T_{DC}(\pi)]\!] = \mathfrak{A}[\![\pi]\!]$ for all $\pi \in \mathcal{SLC}$ with $\beta = \alpha_1; ...; \alpha_n$.
Define $\pi_n := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_n)$ with $\beta_n := \alpha_1; ...; \alpha_n$. Additionally let $\pi_{n+1} := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_{n+1}) \in \mathcal{SLC}$ with $\beta_{n+1} = \alpha_1; ...; \alpha_n; \alpha_{n+1}$ and $\alpha_{n+1} = x \leftarrow e$.
Now we have two cases:

1. $x \notin V_{out}$:
   α_{n+1} is dead code and does not influence the semantics of π_{n+1}. Thus we have for π_n:

$$\mathfrak{A}[\![\pi_{n+1}]\!] = \mathfrak{A}[\![\pi_n]\!] \underbrace{=}_{\text{I.H.}} \mathfrak{A}[\![T_{DC}(\pi_n)]\!] \underbrace{=}_{\text{def.}} \mathfrak{A}[\![T_{DC}(\pi_{n+1})]\!]$$

2. $x \in V_{out}$:
According to the induction hypothesis for π_n it holds $\mathfrak{A}[\![T_{DC}(\pi_n)]\!] = \mathfrak{A}[\![\pi_n]\!]$. Hence it follows:

$$(\mathfrak{A}[\![\beta_{n+1}]\!]\sigma_0)(y) = (\mathfrak{A}[\![\beta_n]\!]\sigma_0)(y) \text{ for } y \in V_{out} \setminus \{x\}$$
$$(\mathfrak{A}[\![\beta_{n+1}]\!]\sigma_0)(x) = (\mathfrak{A}[\![e]\!](\mathfrak{A}[\![\beta_n]\!]\sigma_0))(x)$$

The reduction of $\mathfrak{A}[\![T_{DC}(\pi_{n+1})]\!]$ to $\mathfrak{A}[\![T_{DC}(\pi_n)]\!]$ works similarly:
It holds: $V_{out} = NV_{n+1} \;\Rightarrow\; NV_n = V_{out} \setminus \{x\} \cup V_e$.
In π_{n+1} and π_n therefore the same instructions are removed. We get:

$$(\mathfrak{A}[\![T_{DC}(\beta_{n+1})]\!]\sigma_0)(y) = (\mathfrak{A}[\![T_{DC}(\beta_n)]\!]\sigma_0)(y) \text{ for } y \in V_{out} \setminus \{x\}$$
$$(\mathfrak{A}[\![T_{DC}(\beta_{n+1})]\!]\sigma_0)(x) = (\mathfrak{A}[\![e]\!](\mathfrak{A}[\![T_{DC}(\beta_n)]\!]\sigma_0))(x)$$

Thus altogether $\mathfrak{A}[\![\pi_{n+1}]\!] = \mathfrak{A}[\![T_{DC}(\pi_{n+1})]\!]$. □

The second defining property of a program transformation is its idempotency. There also exists a non–idempotent variant of Dead Code Elimination which is based on a so–called "Live–Variable Analysis", and which has to be applied repeatedly to obtain optimal results. For details see e.g. [ASU86,NNH99].

Theorem 3.6 (Idempotency). *For every $\pi \in \mathcal{SLC}$, $T_{DC}(T_{DC}(\pi)) = T_{DC}(\pi)$.*

Proof. It suffices to show that T_{DC} does not modify the NV–sets of the instructions remaining after the first T_{DC}–application. For simplicity we will only examine the elimination of a single instruction. An inductive extension to arbitrarily many is possible.

Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; ...; \alpha_n)$ and $\pi' := T_{DC}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; ...; \alpha_{i-1}; \alpha_{i+1}; ...; \alpha_n)$ and $\alpha_i = x \leftarrow e$ with $x \notin NV_i$. The sets $NV_{i+1}, ..., NV_n$ remain unchanged because of the backward analysis. Hence for the case $i = 1$ nothing is left to show.

For $i > 1$ in the initial program π:

$$\begin{aligned} NV_{i-1} &= t_{\alpha_i}(NV_i) \\ &= \begin{cases} NV_i \setminus \{x\} \cup V_e & \text{if } x \in NV_i \\ NV_i & \text{else} \end{cases} \\ &= NV_i \quad (\text{since } x \notin NV_i) \\ &= \begin{cases} V_{out} & \text{if } i = n \\ t_{\alpha_{i+1}}(NV_{i+1}) & \text{else} \end{cases} \end{aligned}$$

For π' we obtain the same result:

$$NV_{i-1} = \begin{cases} V_{out} & \text{if } i-1 = n-1 \\ t_{\alpha_{i+1}}(NV_{i+1}) & \text{else} \end{cases}$$

Assumption: T_{DC} also eliminates $\alpha_j = x' \leftarrow e'$, $i \neq j$ from π'.

It follows $x' \notin NV_j$ in π' and $x' \notin NV_j$ in π. But then α_j would already have been removed from π after the first application of T_{DC}. □

Because of the idempotency a single application of the Dead Code Elimination suffices to obtain a program that is optimal[13] w.r.t. T_{DC}. Later we will see that by using other optimizations on a T_{DC}–optimal program it can be possible that new dead code arises.

Thus we have proven that T_{DC} is a program transformation. Moreover it is clear that it improves a program with respect to both the number of instructions and operations: $c(T_{DC}(\pi)) \leq c(\pi)$ for $c \in \{c_l, c_{op}\}$.

3.2 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 3.7 (Available Expressions Analysis). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta = \alpha_1; ...; \alpha_n$ and $\alpha_i = x_i \leftarrow e_i$ for every $i \in \{1, ..., n\}$. 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 transfer functions $t_{\alpha_i} : \mathfrak{P}(\{1, ..., n\}) \to \mathfrak{P}(\{1, ..., n\})$ are given by $t_{\alpha_i}(M) := kill_{\alpha_i} \circ gen_{\alpha_i}$ where $gen_{\alpha_i}, kill_{\alpha_i} : \mathfrak{P}(\{1, ..., n\}) \to \mathfrak{P}(\{1, ..., n\})$ are defined by

$$gen_{\alpha_i}(M) := \begin{cases} M \cup \{i\} & \text{if } e_i = f(u_1, ..., u_r) \text{ and } \forall j \in M : e_j \neq e_i \\ M & \text{else} \end{cases}$$
$$kill_{\alpha_i}(M) := M \setminus \{j \in M \mid x_i \in V_{e_j}\}$$

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

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

By computing the AE–sets the available expressions for each instruction can be found. For the subsequent program transformation, however, we need the inverted view: given an instruction we have to determine where the corresponding expression is repeated and available.

Definition 3.8 (Common Subexpression Elimination). For each instruction, the function $vr : \{1, ..., n\} \to \mathfrak{P}(\{1, ..., n\})$ yields the *valid recurrences* of the corresponding expression: $vr(i) := \{j \in \{i + 1, ...n\} \mid i \in AE_j, e_i = e_j\}$.

The program transformation $T_{CS} : \mathcal{SLC} \to \mathcal{SLC}$ works as follows: for every $i \in \{1, ..., n\}$ 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. replace the instructions $\alpha_j = y \leftarrow e$ by $y \leftarrow t_i$ for all $j \in vr(i)$.

Example 3.9. Also the Common Subexpression Elimination will be illustrated with an exemplary computation for the program from Fig. 1:

[13] $\pi \in \mathcal{SLC}$ is called *optimal* w.r.t. $T : \mathcal{SLC} \to \mathcal{SLC}$ if $T(\pi) = \pi$.

i	α_i	AE_i	$vr(i)$	new instruction(s)
1	$u \leftarrow 3;$	$\emptyset$	$\emptyset$	
2	$v \leftarrow x - y;$	$\emptyset$	$\{4\}$	$t_2 \leftarrow x - y;$ $v \leftarrow t_2;$
3	$w \leftarrow u + 1;$	$AE_2 \cup \{2\} = \{2\}$	$\emptyset$	
4	$x \leftarrow x - y;$	$AE_3 \cup \{3\} = \{2,3\}$	$\emptyset$	$x \leftarrow tv_2;$
5	$v \leftarrow w - 1;$	$(AE_4 \cup \{4\}) \setminus \{2,4\} = \{3\}$	$\emptyset$	
6	$u \leftarrow x - y;$	$AE_5 \cup \{5\} = \{3,5\}$	$\emptyset$	
7	$z \leftarrow u * w;$	$(AE_6 \cup \{6\}) \setminus \{3\} = \{5,6\}$	$\emptyset$	
8	$u \leftarrow 2 * u;$	$AE_7 \cup \{7\}) = \{5,6,7\}$	$\emptyset$	

The output program is longer than the input program. But we were able to avoid a second evaluation of the expression $x - y$.

Now we have to show that the Common Subexpression Elimination is a program transformation.

Theorem 3.10 (Correctness). *For every $\pi \in \mathcal{SLC}$, $\pi \sim T_{CS}(\pi)$.*

Proof. According to Thm. 2.16 for strong equivalence it suffices to show that the term representation of π w.r.t. the output variables does not change, i.e., that $t_\pi(v) = t_{T_{CS}(\pi)}(v)$ for all $v \in V_{out}$.

Similarly to the proof of Thm. 3.6 we consider only the simplified case that exactly one expression with one valid recurrence is optimized by T_{CS}.

Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; ...; \alpha_n)$ and $\pi' := T_{CS}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ with

$$\beta' := \alpha'_1; ...; \alpha'_{n+1} := \alpha_1; ...; \alpha_{i-1}; \alpha'_i; \alpha''_i; \alpha_{i+1}; ...; \alpha'_j; ...; \alpha_n.$$

Thus $\alpha_i = x \leftarrow e$ and $\alpha_j = y \leftarrow e$ with $i \in AE_j$. Then $\alpha'_i = t_i \leftarrow e$, $\alpha''_i = x \leftarrow t_i$ and $\alpha'_j = y \leftarrow t_i$.

Now we analyze the substitution term $s \in T_\Sigma(V_\pi)$ for y at the position i in both of the programs. In π we have $s = e$ since e is not changed anymore due to $i \in AE_j$. In π' we get $s = t_i[t_i/e] = e$ because t_i is modified nowhere, except at position i.

If y occurs in a term during the computation of the term representation of π for $v \in V_{out}$ we obtain the same substitutions for both of the programs and therefore $t_\pi(v) = t_{T_{CS}(\pi)}(v)$ for all $v \in V_{out}$. □

Like Dead Code Elimination also Common Subexpression Elimination is idempotent and therefore a single application is sufficient to obtain a T_{CS}–optimal program:

Corollary 3.11 (Idempotency). *For every $\pi \in \mathcal{SLC}$, $T_{CS}(T_{CS}(\pi)) = T_{CS}(\pi)$.*

Proof. Since already during the first analysis all available expressions are found and their valid recurrences are replaced accordingly but no additional operation expressions are added. Therefore a second application of T_{CS} cannot effectuate any changes. □

3.3 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. 2.9) is extended to allow the "evaluation" of expressions with unknown variable values.

Definition 3.12 (Extended Instruction Semantics). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with signature $\Sigma := (F, C)$ and $\beta := \alpha_1; ...; \alpha_n$. In addition let $\mathfrak{A} = (A, \varphi)$ be an interpretation of Σ. Define the new state space as $\bar{S} := \{\sigma \mid \sigma : V_\pi \to A \cup \{\perp\}\}$ where $\perp$ denotes an unknown variable value. The extended semantics $\bar{\mathfrak{A}}[\![\alpha]\!] : \bar{S} \to \bar{S}$ of an instruction $\alpha = x \leftarrow e$ for $\sigma \in \bar{S}$ are then:

$$\bar{\mathfrak{A}}[\![\alpha]\!]\sigma := \begin{cases} \mathfrak{A}[\![\alpha]\!]\sigma & \text{if } \forall v \in V_e : \sigma(v) \neq \perp \\ \sigma[x/\perp] & \text{else} \end{cases}$$

In other words: if at least one argument of a function is an unknown variable also the evaluation result is unknown. Special properties of operations, e.g. $\forall x \in \mathbb{R} : 0 \cdot x = 0$, are ignored.

Definition 3.13 (Reaching Definition Analysis). Let $\pi, \mathfrak{A}, \bar{S}$ be given as in Def. 3.12. The transfer function $t_\alpha : \bar{S} \to \bar{S}$ for an instruction α is defined by

$$t_\alpha(\sigma) := \bar{\mathfrak{A}}[\![\alpha]\!]\sigma.$$

Now we obtain the variable assignment $RD_i \in \bar{S}$ for α_i by successive transformation:

$$\begin{aligned} RD_1 &:= \sigma_\perp \text{ with } \forall v \in V_\pi : \sigma_\perp(v) := \perp \\ RD_{i+1} &:= t_{\alpha_i}(RD_i) \text{ for } i \in \{1, ..., n\} \end{aligned}$$

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

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

where for $\sigma \in \bar{S}$ (and $c \in C, y \in V, u_i \in V \cup C, f \in F^{(r)}$):

$$\begin{aligned} \bar{\sigma}(c) &:= \varphi(c) \\ \bar{\sigma}(y) &:= \begin{cases} \sigma(y) & \text{if } \sigma(y) \neq \perp \\ y & \text{else} \end{cases} \\ \bar{\sigma}(f(u_1, ..., u_r)) &:= \begin{cases} f(\bar{\sigma}(u_1), ..., \bar{\sigma}(u_r)) & \text{if } \exists i \in \{1, ..., r\} \text{ such that } \sigma(u_i) = \perp \\ \varphi(f)(\bar{\sigma}(u_1), ..., \bar{\sigma}(u_r)) & \text{else} \end{cases} \end{aligned}$$

Example 3.15. When applying T_{CF} to the program from Fig. 1 we get the following computation:

i	α_i	RD_i	new instruction α'_i
1	$u \leftarrow 3;$	$\sigma_\perp$	$u \leftarrow 3;$
2	$v \leftarrow x - y;$	$\sigma_\perp[u/3] =: \sigma_1$	$v \leftarrow x - y;$
3	$w \leftarrow u + 1;$	$\sigma_1[v/\perp] = \sigma_1$	$w \leftarrow 4;$
4	$x \leftarrow x - y;$	$\sigma_1[w/4] =: \sigma_2$	$x \leftarrow x - y;$
5	$v \leftarrow w - 1;$	$\sigma_2[x/\perp] = \sigma_2$	$v \leftarrow 3;$
6	$u \leftarrow x - y;$	$\sigma_2[v/3] =: \sigma_3$	$u \leftarrow x - y;$
7	$z \leftarrow u * w;$	$\sigma_2[u/\perp] =: \sigma_4$	$z \leftarrow u * 4;$
8	$u \leftarrow 2 * u;$	$\sigma_4[z/\perp] = \sigma_4$	$u \leftarrow 2 * u;$

Constant Folding is "producing" additional Dead Code as we see in the example: the first instruction is dispensable since u is not used anymore until its next assignment in the 5th instruction. T_{CF} has replaced all occurrences of u between the first and the fifth instruction with the scalar 3.

For the correctness of Constant Folding the particular interpretation has to be taken into account. Thus strong equivalence is generally not preserved.

Theorem 3.16 (Correctness). *For every $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ and every Σ–algebra $\mathfrak{A}$, $\pi \sim_{\mathfrak{A}} T_{CF}(\pi)$.*

Proof. Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\Sigma := (F, C)$, $\beta := \alpha_1; ...; \alpha_n$, $\alpha_i := x_i \leftarrow e_i$ and $\mathfrak{A} := (A, \varphi)$ an interpretation of Σ. It suffices to show that $\mathfrak{A}[\![e_i]\!] = \mathfrak{A}[\![e'_i]\!]$ holds for all $i \in \{1, .., n\}$ where $e'_i = \overline{RD}_i(e)$.

Depending on the type of e_i we get different cases:

1. $e_i = c \in C$.
   Then $e'_i := \varphi(c)$ and for a valuation $\sigma \in S$:
$$\mathfrak{A}[\![e_i]\!]\sigma = \varphi(c) \overset{\text{def. of } \varphi}{=} \varphi(\varphi(c)) = \mathfrak{A}[\![e'_i]\!]\sigma$$
2. $e_i = y \in V$. Two subcases:
   (a) $RD_i(y) = \perp \;\Rightarrow\; e'_i = e_i$, i.e., the semantics are trivially identical.
   (b) $RD_i(y) = a \in A \;\Rightarrow\; e'_i = a$. In this case follows for $\sigma \in S$:
$$\mathfrak{A}[\![e_i]\!]\sigma = \underbrace{\sigma(y)}_{\in A} \overset{\text{def. of } \varphi}{=} \varphi(\sigma(y)) = \mathfrak{A}[\![e'_i]\!]\sigma$$
3. $e_i = f(u_1, ..., u_r)$. Again two cases:
   (a) $\exists j \in \{1, ..., r\}$ such that $RD_i(u_j) = \perp$:
   $e'_i = f(\overline{RD}_i(u_1), ..., \overline{RD}_i(u_n))$ and we have to show:
$$\begin{aligned}\mathfrak{A}[\![e_i]\!]\sigma &= \varphi(f)(\mathfrak{A}[\![u_1]\!]\sigma, ..., \mathfrak{A}[\![u_r]\!]\sigma)\\ &= \varphi(f)(\mathfrak{A}[\![\overline{RD}_i(u_1)]\!]\sigma, ..., \mathfrak{A}[\![\overline{RD}_i(u_r)]\!]\sigma) = \mathfrak{A}[\![e'_i]\!]\sigma\end{aligned}$$
   Thus we have to prove that $\mathfrak{A}[\![u_j]\!]\sigma = \mathfrak{A}[\![\overline{RD}_i(u_j)]\!]\sigma$ for $j \in \{1, ..., r\}$. Since the u_j are neither variables nor constants the statement follows already from case 1 or 2.

(b) $\forall j \in \{1, ..., r\}$ we have either $RD_i(u_j) = a \in A$ or $u_j \in C$. $e'_i = \varphi(f)(\overline{RD}_i(u_1), ..., \overline{RD}_i(u_n))$. The only difference to subcase (a) is the additional application of $\varphi(f)$. But since this needs to be done nevertheless when computing the semantics we get the same problem as in (a).

The above conclusions can be applied for all e_i, $i \in \{1, ..., n\}$, and hence the theorem is proved. $\square$

Corollary 3.17 (Idempotency). *For every* $\pi \in \mathcal{SLC}$, $T_{CF}(T_{CF}(\pi)) = T_{CF}(\pi)$.

The idempotency of Constant Folding is obvious because already the first application substitutes all variables that have constant values with constants and evaluates constant expressions. Therefore only "unknown" variables and expressions that contain at least one unknown operand remain.

Constant folding does not remove instructions, hence the program length does not change. A subsequent Dead Code Elimination, however, will shorten the program in many cases.

4 DAG Optimization

The DAG optimization is an optimization algorithm for SLC–programs based on the construction of a *directed acyclic graph* (DAG). A basic version of this optimization has been introduced in [ASU70]. We will present a modified version which, however, does not consider the register allocation procedure since we only focus on intermediate code.

Definition 4.1 (DAG). A DAG is a graph $G = (K, L, lab, suc)$ with the following components/properties:

- a set of nodes K,
- a set of labels L,
- a labeling function $lab : K \to L$,
- a partially defined[14] successor function $suc :\subseteq K \times \mathbb{N} \to K$ and
- $\nexists k \in K$ such that $\exists k_1, k_2, ..., k_n \in K$ with $k_1 = k$, $k_n = k$ and $suc(k_j, i_j) = k_{j+1}$ for $i_j \in \mathbb{N}$, $j \in \{1, ..., n-1\}$ (G is *acyclic*).

A DAG can represent the result and the operands of a functional expression by (different) nodes that are linked by the successor function.

4.1 DAG of an SLC–Program

The DAG of an SLC–program can be seen as a graphical version of the term representation with the difference that identical subterms are shared. Furthermore a partial evaluation of expressions (similar to Constant Folding) is performed (extending the algorithm in [ASU70]). Hence the signature of the target program has to be modified analogously to Constant Folding.

In addition to the DAG we need a valuation function

$$val :\subseteq V_\pi \times \mathbb{N} \to K$$

[14] $f :\subseteq A \to B$ denotes a partially defined function f from A to B.

with $val(x,i) = k$ iff the subgraph starting with k represents the value of the variable x after i computation steps.

Algorithm 4.2 (DAG Construction). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\Sigma := (F, C)$, $\beta := \alpha_1; ...; \alpha_n$ and $\mathfrak{A} := (A, \varphi)$ an interpretation for Σ. The *DAG* D_π *of* π consists of a DAG $G = (K, L, lab, suc)$, a valuation function val (as above), i.e., $D_\pi = (G, val)$. We set $L := F \cup V_{in} \cup A$ to label nodes that represent expressions with the corresponding function symbols (the others are labeled with themselves).

The graph G and val are inductively constructed as follows:

1. Select $K := V_{in} \cup \varphi(C_\pi)$ with $lab(k) = k,\ \forall k \in K$ as initial nodes[15] and set $val(x, 0) := x$ for all $x \in V_{in}$ where
$$\varphi(C_\pi) := \{\varphi(c) \mid c \in C_\pi\}$$
2. *Induction Hypothesis:* Assume G and val are already constructed for $\alpha_1; ...; \alpha_i$ (especially[16] $val(x, i) \in K$ for $x \in DV_i$).
Let w.l.o.g. $\alpha_{i+1} = x \leftarrow e$. We distinguish different cases depending on the type of the expression e:
(i) $e = y \in V$:
According to the induction hypothesis $val(y, i) \in K$ is already representing the current value of y. Thus G is not extended; set
$$\begin{aligned} val(x, i+1) &:= val(y, i) \\ val(x', i+1) &:= val(x', i) \text{ for } x' \neq x \end{aligned}$$
(ii) $e = c \in C$:
$\varphi(c) \in K$ already exists. Therefore G remains unchanged:
$$\begin{aligned} val(x, i+1) &:= \varphi(c) \\ val(x', i+1) &:= val(x', i) \text{ for } x' \neq x \end{aligned}$$
(iii) $e = f(u_1, ..., u_r),\ u_j \in V \cup C,\ f \in F^{(r)}$. We get further subcases:
(a) $u_j \in C$ or $u_j \in V$ and $val(u_j, i) \in A$ for all $j \in \{1, ..., r\}$.
Let $a := \varphi(f)(u'_1, ..., u'_r) \in A$ with
$$u'_j := \begin{cases} \varphi(u_j) & \text{if } u_j \in C \\ val(u_j, i) & \text{if } u_j \in V \text{ and } val(u_j, i) \in A \end{cases}$$
(a1) $a \in \varphi(C_\pi)$: No extension of G.
$$\begin{aligned} val(x, i+1) &:= a \\ val(x', i+1) &:= val(x', i) \text{ for } x' \neq x \end{aligned}$$
(a2) $a \notin \varphi(C_\pi)$: insertion of a new node a:
$$\begin{aligned} K &:= K \cup \{a\} \\ val(x, i+1) &:= a \\ val(x', i+1) &:= val(x', i) \text{ for } x' \neq x \end{aligned}$$

[15] One could also add the constants later "on demand".
[16] The DV_i sets have been introduced in Def. 2.5.

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

(b1) $\exists k \in K$ 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:

$$\begin{aligned} val(x, i+1) &:= k \\ val(x', i+1) &:= val(x', i) \text{ for } x' \neq x \end{aligned}$$

(b2) Otherwise ($\nexists k \in K$ as in (b1)): insert a node k_{i+1}:

$$\begin{aligned} K &:= K \cup \{k_{i+1}\} \\ lab(k_{i+1}) &:= f \\ suc(k_{i+1}, j) &:= \begin{cases} \varphi(u_j) & \text{if } u_j \in C \\ val(u_j, i) & \text{if } u_j \in V \end{cases} \\ val(x, i+1) &:= k_{i+1} \\ val(x', i+1) &:= val(x', i) \text{ for } x' \neq x \end{aligned}$$

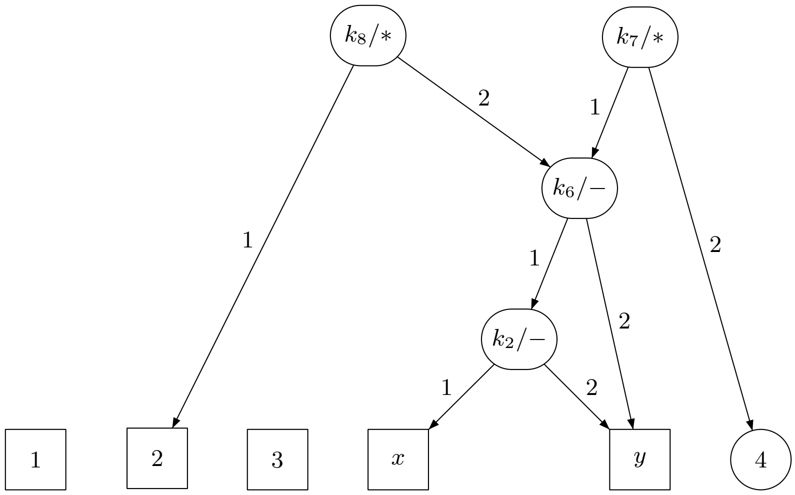
In the following we will refer to a node k as *operation node* if it has a label $lab(k) \in F$. Nodes that stand for variables or constants are called *variable* or *constant nodes*. Nodes that do not have any successors are called *leaves*.

Example 4.3. Figure 2 shows the DAG belonging to the program from Fig. 1. The square nodes are the nodes already present at the beginning of the construction. The round nodes were created later according to the above definition.

The edges represent the *suc*–function, the numbering is mandatory for non–commutative operations (like $-$). For better clarity the table of the *val*–function only shows those entries that represent changes.

Let us now examine the construction of the graph in detail. For first $V_{in} \cup \varphi(C_\pi)$ are inserted in K as initial nodes. In the example these are the nodes $\{1, 2, 3, x, y\}$. Furthermore $val(x, 0) = x$ and $val(y, 0) = y$ since $\{x, y\} = V_{in}$. The DAG D_π is now constructed as follows:

1. $\alpha_1 = u \leftarrow 3$: no extension of the graph is necessary since $\varphi(3) = 3 \in K$. The *val*–function has to be modified with $val(u, 1) := 3$ (all other values are transferred from $val(\star, 0)$).
2. $\alpha_2 = v \leftarrow x - y$: because $\{val(x, 1), val(y, 1)\} \cap A = \emptyset$ and no operation node is existing the node k_2 with label $lab(k_2) := -$ is inserted. Additionally we set $suc(k_2, 1) := val(x, 1) = x$, $suc(k_2, 2) := val(y, 1) = y$ and $val(v, 2) := k_2$ (the other *val*–values are transferred).
3. $\alpha_3 = w \leftarrow u + 1$ represents a function application to constant operands. $3 + \varphi(1) = 3 + 1 = 4$ holds. Since the constant 4 is not yet present in the graph a node 4 is inserted and *val* updated: $val(w, 3) := 4$.
4. $\alpha_4 = x \leftarrow x - y$: for $x - y$ the equivalent node k_2 exists already in the graph, the conditions of case (iii)(b1) of the algorithm are fulfilled and thus no new node is inserted. Update: $val(x, 4) := k_2$.



val-Function:

i	u	v	w	x	y	z
0				x	y	
1	3					
2		k_2				
3			4			
4				k_2		
5		3				
6	k_6					
7						k_7
8	k_8					

Fig. 2. DAG D_π for the program π from Fig. 1

5. $\alpha_5 = v \leftarrow w - 1$: $val(w, 4) = 4 \in A$ and $\varphi(1) = 1 \in A$. Therefore we evaluate $\varphi(-)(4, 1) = 3$ and set $val(v, 5) := 3$ because 3 is already a node in K.
6. $\alpha_6 = u \leftarrow x - y$: due to the change of x in instruction 4 a new node k_6 with $lab(k_6) = -$, $suc(k_6, 1) = k_2$ and $suc(k_6, 2) = y$ must be created (obviously the arguments of $-$ are not constant). Besides $val(u, 6) := k_6$ is set.
7. $\alpha_7 = z \leftarrow u * w$: since u is a non–constant operand of $*$ ($val(u, 6) = k_6 \notin A$) and no node with the successors k_6 and 4 and label $*$ exists yet we insert the node k_7 with $lab(k_7) = *$, $suc(k_7, 1) = k_6$ and $suc(k_7, 2) = 4$. Moreover val is modified by $val(z, 7) := k_7$.
8. $\alpha_8 = u \leftarrow 2 * u$: $val(u, 7) = k_6 \notin A$ and no matching node with the corresponding successors exists. Thus we create k_8 with $lab(k_8) = *$, $suc(k_8, 1) = 2$ and $suc(k_8, 2) = k_6$ and update $val(u, 8) := k_8$.

The DAG construction incorporates aspects of Common Subexpression Elimination and Constant Folding:

– For expressions that are already represented by a node in the DAG no additional operation node is created, instead a “pointer” to the equivalent node is used (node sharing).
– The DAG construction performs a partial evaluation of expressions based on constant information. Either we get for the whole expression one constant

value (for instance if all the operands are constants) or the variables occurring in the expression are replaced by constants as far as it is possible. Therefore no node represents a constant expression.

The elimination of dead code takes place during the code generation from the DAG and not during its construction.

4.2 Code Generation from a DAG

For defining the program transformation the input program is no longer needed, its DAG is sufficient; this procedure we call *code generation*. First it has to be determined which nodes in the graph are really necessary for generating an equivalent output program.

Definition 4.4 (Output–Relevant Nodes). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta := \alpha_1; ...; \alpha_n$ and $D_\pi = (G, val)$ with $G = (K, L, lab, suc)$ the DAG of π. A node $k \in K$ is called *output relevant* if

1. there exists a $y \in V_{out}$ such that $val(y, n) = k$, or
2. there exists an output relevant node $k' \in K$ and an $i \in \mathbb{N}$ with $suc(k', i) = k$.

Thus all nodes that are reachable from an "output node" are output relevant. The other nodes are dispensable and are not considered during code generation, thus implementing Dead Code Elimination.

For code generation the nodes of the graph have to be processed in a certain order. Before creating an instruction for a node k all the successors of k have to be processed first for the input values to be available.

A Simple Nondeterministic Algorithm

In this section we will present a simple algorithm for code generation on the basis of an example.

Example 4.5. Let π be the program from Fig. 1 and D_π its DAG which is depicted in Fig. 2. Then the code generation could be done in the following steps:

1. First we eliminate all nodes that are not output relevant, these are the nodes k_7, 4 and 1. The others are reachable from k_8 except for the constant 3, for which $val(v, 8) = 3$ holds. Thus the remaining nodes are $\{2, 3, x, y, k_2, k_6, k_8\}$.
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 nodes depend on it. For this node we generate the instruction
$$k_2 \leftarrow x - y;$$
The node label determines the operation and its successors the operands. As temporary variable name we simply use the name of the node.
3. The next node to process is k_6 because k_8 depends on it. For k_6 the instruction
$$k_6 \leftarrow k_2 - y;$$
is created.

4. For the last remaining operation node k_8 the generated code looks as follows:

$$u \leftarrow 2 * k_6;$$

Here we do not use a temporary 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 yet undefined. Therefore we have to add the instruction

$$v \leftarrow 3;$$

since $val(v, 8) = 3$.

Thus we get the following result:

$$\begin{aligned} \vec{v}_{in} :\ & (x, y) \\ \beta' :\ & k_2 \leftarrow x - y; \\ & k_6 \leftarrow k_2 - y; \\ & u \ \leftarrow 2 * k_6; \\ & v \ \leftarrow 3; \\ \vec{v}_{out} :\ & (u, v) \end{aligned}$$

This "naive" code generation technique has several disadvantages:

- The algorithm is nondeterministic, multiple choices for the next node to process are possible.[17] Hence the output depends on the node ordering strategy.
- The idempotency is violated because the node names will change in $D_{T_{DAG}(\pi)}$ due to the elimination of output irrelevant nodes.
- The assignment order is different from the input program, e.g. the variable v gets its final value before u in the input program but in the optimized program after u. This increases the difficulty of proofs.

Extended Algorithm

Since the simple algorithm for code generation does not fulfill the desired properties (e.g. it is not an $\mathfrak{A}$–program transformation) we will now introduce an extended version which, however, is more complicated.

Algorithm 4.6 ($T_{DAG} : \mathcal{SLC} \rightarrow \mathcal{SLC}$). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\Sigma := (F, C)$ and $\beta := \alpha_1; ...; \alpha_n$. Let $\mathfrak{A} := (A, \varphi)$ be an interpretation for Σ and $D_\pi = (G, val)$ the DAG of π with $G := (K, L, lab, suc)$ where K w.l.o.g. contains *only* output–relevant nodes. Finally we need a substitution function $\delta : K \rightarrow V_{out} \cup K$, a function $last : V_{out} \rightarrow \{1, ..., n\}$ that assigns every output variable the index of its final assignment[18] and a counter variable $i \in \mathbb{N}$ with initialization $i := 0$.

Generate the SLC–program $T_{DAG}(\pi) := ((F, A), \vec{v}_{in}, \vec{v}_{out}, \beta')$ as follows (annotations for the particular steps are given in a smaller font):

[17] In our example this is not the case.
[18] In the example program of Fig. 1 $last = \{u \mapsto 8, v \mapsto 5\}$.

1. Set $\beta' := \varepsilon$, $\delta := id$, $K_f := \{k \in K \mid k \text{ is a leaf}\}$ and $V_{out}^f := \emptyset$.
   We start with the empty instruction sequence and the identical substitution. K_f denotes the set of already processed nodes. Initially these are the leaves which are immediately available without generating code for them. V_{out}^f is the set of output variables that are already assigned a final value and not be used for new assignments.
2. Let $K' := \{k \in K \setminus K_f \mid \nexists i \in \mathbb{N} \text{ with } suc(k,i) \in K \setminus K_f\}$. Case distinction:
   (a) If $K' \neq \emptyset$: choose the node $k_l \in K'$ with $\forall k_j \in K' : \ l \leq j$ and increment $i := i + 1$.
   (b) Otherwise: continue with step 8.
   K' denotes the set of not yet processed nodes from K whose successors are already processed (these must be operation nodes). The node k_l has the smallest index of all nodes in K'. This preserves the computation order of the input program (since in Alg. 4.2 l is the index of the instruction that causes the generation of k_l). In addition the choice of the lowest–numbered node guarantees the determinism because there could be multiple choices that would result in different outputs.
3. Let $V_{out}^{<l} := \{y \in V_{out} \setminus V_{out}^f \mid last(y) < l\} =: \{y_{i_1}, ..., y_{i_q}\}$ with $last(y_{i_j}) < last(y_{i_k})$ for $j < k$. Set $V_{out}^f := V_{out}^f \cup V_{out}^{<l}$ as well as $i := i + q$ and create for $j = 1, ..., q$ respectively a new instruction:
   $$\beta' := \beta' \cdot y_{i_j} \leftarrow \delta(val(y_{i_j}, n))$$
   $V_{out}^{<l}$ is the set of output variables that do not yet have a final value in the target program, but whose final values in the input program are assigned in an instruction numbered lower than the current node. Before processing k_l these must defined to maintain the computation order of the input program.
   The order of the assignments to output variables in $V_{out}^{<l}$ is corresponding to the input program since the y_{i_j} are ordered by their *last*–value. The set of "used" output variables is extended by $V_{out}^{<l}$ after creating the new instructions. The counter i is incremented by q to represent the number of the instructions to be inserted in step 5.
4. Let $V_{out}^{k_l} := \{y \in V_{out} \mid val(y,n) = k_l\}$. If $V_{out}^{k_l} \neq \emptyset$ select $y \in V_{out}^{k_l}$ such that $\forall y' \in V_{out}^{k_l} : \ last(y) \leq last(y')$ and set:
   $$\begin{aligned} \delta &:= \delta[k_l/y] \\ V_{out}^f &:= V_{out}^f \cup \{y\} \end{aligned}$$
   $V_{out}^{k_l}$ are the output variables associated with k_l out of which one is chosen. If there is more than one we select the one with the smallest *last*–value (determinism, computation order of the input program). Then the substitution δ is updated to rename the current node k_l into the newly chosen output variable y and y is inserted in the set of used output variables V_{out}^f.
5. If $\delta(k_l) \notin V_{out}$ set $\delta := \delta[k_l/v_i]$.
   If no output variable is associated with k_l instead of the node name the variable v_i is used for the next assignment (change of the δ–function). This is necessary for assuring the idempotency of the algorithm.
6. Let w.l.o.g. $lab(k_l) = f \in F^{(r)}$. Extend β':
   $$\beta' := \beta' \cdot \delta(k_l) \leftarrow f(\delta(suc(k_l,1)), \delta(suc(k_l,2)), ..., \delta(suc(k_l,r)));$$
   Insertion of a new operation instruction for the current node k_l. Thereby k_l itself and all successor nodes that serve as arguments for f are renamed by δ.

7. Set $K_f := K_f \cup \{k_l\}$. Continue with step 2.
   The processing of node k_l is finished and therefore inserted in the set K_f .
8. Let $V_{out}^{<\infty} := V_{out} \setminus V_{out}^{f} =: \{y_{i_1}, ..., y_{i_q}\}$ with $last(y_{i_j}) < last(y_{i_k})$ for $j < k$.
   For $j = 1, ..., q$ set:
   $$\beta' := \beta' \cdot y_{i_j} \leftarrow \delta(val(y_{i_j}, n));$$
   Now we regard the yet unused output variables. Those have to be assigned their final values according to val.

The algorithm processes operation nodes in the order of their generation. Value assignments of output variables eventually have to be inserted "in between". Thus the computation order of the original program is maintained. In the resulting program for every assignment a new, previously undefined variable is used, meaning that it satisfies the following normal form:

Definition 4.7 (SSA Form). A program $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta := \alpha_1; ...; \alpha_n$ and $\alpha_i := x_i \leftarrow e_i$ is in *static single assignment (SSA) form* if for all $i \in \{1, ..., n\}$ the following holds:

$$x_i \notin V_{in} \cup \bigcup_{k=1}^{i-1} \{x_k\}$$

Due to the inductive nature of the definition and the completeness of SLC–programs it ensues that for every i also

$$x_i \notin V_{e_i} \cup \bigcup_{k=1}^{i-1} V_{\alpha_k}$$

Example 4.8. If we apply the extended algorithm for code generation to the example DAG of Fig. 2 we get the result:

$$\begin{aligned}
\vec{v}_{in} :\ & (x, y) \\
\beta' :\ & v_1 \leftarrow x - y; \\
& v \ \ \leftarrow 3; \\
& v_2 \leftarrow v_1 - y; \\
& u \ \ \leftarrow 2 * v_2; \\
\vec{v}_{out} :\ & (u, v)
\end{aligned}$$

As we see, now the output variable v gets its value before u like it is in the original program π and reapplying T_{DAG} to the result will yield the same program.[19] Thus the idempotency holds for the example.

Note that if the number of available registers is limited the optimal code generation from a DAG is NP–complete [AJU77]. The efficiency of the resulting program hereby varies depending on the node–ordering strategy. The code generation from expression *trees*, however, is also in this case efficient [AJ76].

[19] This can be easily tested with the web interface at [Rie05a].

4.3 Correctness of the DAG Algorithm

In this section we will prove the correctness of the DAG optimization. Because of the complex structure of the algorithm the proof will be done step–by–step and be based mainly on terms. Hence for a DAG we have to generate a characterizing term.

Definition 4.9 (Term Representation of a DAG). Let $D_\pi = (G, val)$ with $G = (K, L, lab, suc)$ be the DAG of the SLC–program $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ with $\Sigma := (F, C)$, $\beta := \alpha_1; ...; \alpha_n$ and let $\mathfrak{A} := (A, \varphi)$ be an interpretation of Σ. Then the term representation $t_{D_\pi}(y) \in T_\Sigma(V_{in})$ of D_π w.r.t. an output variable $y \in V_{out}$ is defined by:

$$t_{D_\pi}(y) := t_{val(y,n)}$$
$$\text{and for } k \in K: \ t_k := \begin{cases} k & \text{if } k \in A \cup V_{in} \\ f\left(t_{suc(k,1)}, ..., t_{suc(k,r)}\right) & \text{if } lab(k) = f \in F^{(r)} \end{cases}$$

The definition complies with the intuitive idea that all successors of a node are representing subterms of the node term. Here the optimization effect achieved by the sharing of nodes is lost which, however, is irrelevant for the correctness.

Example 4.10. The computation of the term representations for the DAG from Fig. 2 yields

$$t_{D_\pi}(u) = *(2, -(-(x, y), y))$$
$$t_{D_\pi}(v) = 3$$

The term representation of the DAG for the variable v is not equal to the term representation of π because during the DAG construction an evaluation of constant expressions has been performed.

For adapting the term representation of the input program we need to evaluate constant subterms.

Definition 4.11 (Partial Evaluation of Terms). Let $t \in T_\Sigma(X)$ with $\Sigma := (F, C)$ and $\mathfrak{A} := (A, \varphi)$ an interpretation of Σ. Then $\mathfrak{A}[t] \in T_{(F,A)}(X)$ denotes the term resulting from *partial evaluation* of t. For $c \in C,\ x \in X,\ f \in F^{(r)}$ and $t_1, ..., t_n \in T_{(F,A)}(X)$ it is defined as:

$$\mathfrak{A}[c] := \varphi(c)$$
$$\mathfrak{A}[x] := x$$
$$\mathfrak{A}[f(t_1, ..., t_r)] := \begin{cases} \varphi(f)(\mathfrak{A}[t_1], ..., \mathfrak{A}[t_r]) & \text{if } \forall i \in \{1, ..., r\} : \mathfrak{A}[t_i] \in A \\ f(\mathfrak{A}[t_1], ..., \mathfrak{A}[t_r]) & \text{else} \end{cases}$$

The correctness proof of the DAG optimization is based on two main steps: first we show the correctness of the DAG construction and then – based on the correctness of the DAG construction – that the code generation algorithm is correct. Then it only remains to show that the equality of the terms implies the semantic equality.

Lemma 4.12 (Correctness of the DAG Construction). *For $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$, its DAG D_π and an interpretation $\mathfrak{A}$ for Σ it holds*

$$\mathfrak{A}[t_\pi(y)] = t_{D_\pi}(y) \text{ for all } y \in V_{out}.$$

Proof. Let $D_\pi := (G, val)$ with $G := (K, L, lab, suc)$ and $\beta := \alpha_1; ...; \alpha_n$ with $\alpha_i := x_i \leftarrow e_i$. Moreover let $\Sigma := (F, C)$ and $\mathfrak{A} := (A, \varphi)$. Induction on the term structure (for $y \in V_{out}$):

1. $\mathfrak{A}[t_\pi(y)] = a \in A$. Inductive case distinction:
   (a) In π there exists an instruction $\alpha_k = y \leftarrow c$ with $c \in C$ and $y \notin \bigcup_{i=k+1}^{n}\{x_i\}$. Then

   $$val(y, n) = val(y, k) = \varphi(c)$$

   (b) An instruction $\alpha_l = y \leftarrow z$ exists in π such that $y \notin \bigcup_{i=l+1}^{n}\{x_i\}$ and for z again one of the cases (a), (b) or (c) (termination in case (a) or (c)(i)) holds with $n = l - 1$ and $y = z$. It follows:

   $$val(y, n) = val(y, l) = val(z, l - 1)$$

   (c) In π there is an instruction $\alpha_m = y \leftarrow f(u_1, ..., u_r)$ with $f \in F^{(r)}$ such that $y \notin \bigcup_{i=l+1}^{n}\{x_i\}$. Additional subcases:
   (i) $\forall j \in \{1, ..., r\} : \ u_j \in C$. Then we get:

   $$val(y, n) = val(y, m) = \varphi(f)(\varphi(u_1), ..., \varphi(u_r))$$

   (ii) $\exists j \in \{1, ..., r\} : \ u_j \in V$ and for u_j either (a), (b) or (c) holds with $n = m - 1$ and $y = u_j$. Result:

   $$val(y, n) = val(y, m) = \varphi(f)(u'_1, ..., u'_r) \text{ with}$$
   $$u'_j := \begin{cases} val(u_j, m-1) & \text{if } u_j \in V \\ \varphi(u_j) & \text{if } u_j \in C \end{cases}$$

   Because of the induction $val(u_j, m - 1) \in A$ must hold for $u_j \in V$.

   We obtain $val(y, n) = b \in A$ and thus $t_{val(y,n)} = b$ (since $n < \infty$ and a termination is only possible in the subcases (a) and (c)(i)). $b = a$ must hold because in the above case distinction the same subterms as in $\mathfrak{A}[t_\pi(y)]$ have been evaluated, with the only difference that evaluations took place immediately and not after the construction of an entire term. Thus:

   $$\mathfrak{A}[t_\pi(y)] = a = t_{D_\pi}(y)$$

2. $\mathfrak{A}[t_\pi(y)] = x \in V_{in}$. Similarly to the first case we inductively distinguish the possibilities:
   (a) It exists an $\alpha_k = y \leftarrow z$ in π with $y \notin \bigcup_{i=k+1}^{n}\{x_i\}$, $z \neq x$ and for z either (a) (induction: *copy chain*[20]) or (b) (termination) holds with $n = k - 1$ and $y = z$. Then:

   $$val(y, n) = val(y, k) = val(z, k - 1)$$

[20] A sequence of copy instructions that depend on each other is called copy chain.

(b) In π, $y = x$ and $y \notin \bigcup_{i=1}^{n}\{x_i\}$. Because of $V_{in} \cap V_{out} = \emptyset$ this case can only occur after case (a) occurred at least once. We get:

$$val(y,n) = val(y,0) = val(x,0)$$

Thus $val(y,n) = x$ and $t_{val(y,n)} = x$. It follows:

$$\mathfrak{A}[t_\pi(y)] = x = t_{D_\pi}(y)$$

3. $\mathfrak{A}[t_\pi(y)] = f(t_1, ..., t_r)$ with $t_j \in T_{(F,A)}(V_{in})$ and $f \in F^{(r)}$.
Induction Hypothesis: For all $j \in \{1, ..., r\}$ already holds

$$t_j = \begin{cases} t_{val(u_j,l-1)} & \text{for } u_j \in V \\ t_{\varphi(u_j)} & \text{for } u_j \in C \end{cases}$$

An instruction $\alpha_l = y \leftarrow f(u_1, ..., u_r)$ with $y \notin \bigcup_{i=l+1}^{n}\{x_i\}$ must exist in π (y unchanged after α_l).
Case distinction:
(a) $\forall j \in \{1, ...r\} : t_j \in T_{(F,A)}$
This case cannot occur since otherwise $\mathfrak{A}[f(t_1, ..., t_r)]$ would be a constant.
(b) $\exists j \in \{1, ...r\} : t_j \notin T_{(F,A)}$
A node $k \in K$ exists with

$$\begin{aligned} lab(k) &= f \\ \forall j \in \{1, ..., r\} : \ suc(k,j) &= \begin{cases} val(u_j, l-1) & \text{if } u_j \in V \\ \varphi(u_j) & \text{if } u_j \in C \end{cases} \\ val(y,n) &= k \end{aligned}$$

Then:

$$\begin{aligned} t_{D_\pi}(y) &= t_{val(y,n)} \\ &= t_k \\ &\overset{\text{Def}}{=} f\left(t_{suc(k,1)}, ..., t_{suc(k,r)}\right) \\ &\overset{\text{IH}}{=} f(t_1, ..., t_r) \\ &= \mathfrak{A}[t_\pi(y)] \end{aligned}$$

Hence for all $y \in V_{out}$ the claim holds. □

Now we have to show the correctness of Alg. 4.6. Also here the proof is based on term representations.

Lemma 4.13 (Correctness of the DAG Code Generation). *For $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ and its DAG D_π it holds*

$$t_{D_\pi}(y) = t_{T_{DAG}(\pi)}(y) \text{ for all } y \in V_{out}.$$

Proof. Let π, D_π, Σ and $\mathfrak{A}$ be defined as in the proof of La. 4.12. Moreover let $T_{DAG}(\pi) =: ((F,A), \vec{v}_{in}, \vec{v}_{out}, \beta')$ with $\beta' := \alpha'_1; ...; \alpha'_m$ and $\alpha'_i := x'_i \leftarrow e'_i$. Induction on the term structure ($y \in V_{out}$):

1. $t_{D_\pi}(y) = a \in A$:
   Hence we have $val(y,n) = a$ in D_π. Then according to step 3 or 8 of Alg. 4.6 an assignment $\alpha'_l = y \leftarrow a$ is created and afterwards y remains unchanged in $T_{DAG}(\pi)$, i.e., $y \notin \bigcup_{i=l+1}^{m} \{x'_i\}$ (this is guaranteed by the V_{out}^f set in which y is inserted after the creation of α'_l). For the term representation of $T_{DAG}(\pi)$ we obtain:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t^{(0)}_{T_{DAG}(\pi)}(y) \\
&= y[x'_m/e'_m]...[x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= y[x'_m/e'_m]...[y/a][x'_{l-1}/e'_{l-1}]...[x_1/e'_1] \\
&= y[y/a][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] && (x'_i \neq y \text{ for all } i > l) \\
&= a[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= a && (a \text{ is variable–free})
\end{aligned}$$

2. $t_{D_\pi}(y) = x \in V_{in}$:
   Then $val(y,n) = x$ in D_π. As in the first case an instruction $\alpha'_l = y \leftarrow x$ is added by step 3 or 8 of the algorithm and it holds $y \notin \bigcup_{i=l+1}^{m} \{x'_i\}$ and $x \notin \bigcup_{i=1}^{m} \{x'_i\}$ since $\bigcup_{i=1}^{m} \{x'_i\} \subseteq V_{out} \cup \{v_j \mid j \in \mathbb{N}\}$. It ensues:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t^{(0)}_{T_{DAG}(\pi)}(y) \\
&= y[x'_m/e'_m]...[x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= y[x'_m/e'_m]...[y/x][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= y[y/x][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] && (x'_i \neq y \text{ for all } i > l) \\
&= x[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= x && (x'_i \neq x \text{ for all } i < l)
\end{aligned}$$

   Hint: If $V_{in} \cap V_{out} = \emptyset$ (Def. 2.3) would not be required it could not be guaranteed that x is not assigned a value before α_l. A counterexample for the correctness of the DAG algorithm would then be the following program:

$$\pi := ((\emptyset, \{0\}), (x), (x, y), \beta) \notin \mathcal{SLC} \text{ with } \beta := \begin{array}{l} t \leftarrow x; \\ x \leftarrow 0; \\ y \leftarrow z; \end{array}$$

3. $t_{D_\pi}(y) = f(t_1, ..., t_r)$ with $t_i \in T_{(F,A)}(V_{in})$ and $f \in F^{(r)}$:
   Then an $i \in \{1, ...r\}$ exists such that $t_i \notin T_{(F,A)}$ (otherwise the term would have been substituted by a constant). Hence in D_π there must be a node $k \in K$ with $lab(k) = f$ and $val(y,n) = k$. Because of step 6 of Alg. 4.6 in β' it exists already an instruction

$$\alpha'_l = \delta(k) \leftarrow f(\delta(suc(k,1)), \delta(suc(k,2)), ..., \delta(suc(k,r))),$$

   where $\delta(k) \in V_{out} \cup \{v_j \mid j \in \mathbb{N}\}$. Let π_k be the program that is obtained if the algorithm is applied to the DAG that only contains all nodes reachable from k. Let the set of nodes restricted by k, K_k, be given by

$$\begin{aligned}
&k \in K_k \\
&k' \in K_k \text{ if } \exists i \in \mathbb{N},\ \exists k'' \in K_k \text{ such that } suc(k'', i) = k'
\end{aligned}$$

Induction Hypothesis: For $j \in \{1,...r\}$, $t_j = t_{\pi_{suc(k,i)}}(\delta(suc(k,j)))$ already holds.

$y \in V_{out}^k$ must hold, i.e., $val(y,n) = k$. Case distinction:

(a) For all $y' \in V_{out}^k$: $last(y) \leq last(y')$. Then according to the algorithm (step 4) $\delta(k) = y$ and $y \notin \bigcup_{i=l+1}^m \{x'_i\}$. It follows:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t^{(0)}_{T_{DAG}(\pi)}(y) \\
&= y[x'_m/e'_m]...[x'_{l+1}/e'_{l+1}][x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= y[x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= y[y/f(\delta(suc(k,1)),...,\delta(suc(k,r)))][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= f(\delta(suc(k,1)),...,\delta(suc(k,r)))[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= f(\underbrace{\delta(suc(k,1))[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1]}_{=t_{\pi_{suc(k,1)}}(\delta(suc(k,1)))},..., \\
&\qquad \underbrace{\delta(suc(k,r))[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1]}_{=t_{\pi_{suc(k,r)}}(\delta(suc(k,r)))})
\end{aligned}$$

With the induction hypothesis the claim holds for this subcase.

(b) $\exists z \in V_{out}^k$ with $z \neq y$ such that $\forall z' \in V_{out}^k$: $last(z) \leq last(z')$. Therefore $\delta(k) = z$ and there exists an instruction $\alpha'_p = y \leftarrow z$ in β with $p > l$ (step 3 or 8). Then $z \notin \bigcup_{i=l+1}^m \{x'_i\}$ as well as $y \notin \bigcup_{i=p+1}^m \{x'_i\}$ and it yields:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t^{(0)}_{T_{DAG}(\pi)}(y) \\
&= y[x'_m/e'_m]...[x'_{p+1}/e'_{p+1}][y/z][x'_{p-1}/e'_{p-1}]...[x'_l/e'_l]...[x'_1/e'_1] \\
&= y[y/z][x'_{p-1}/e'_{p-1}]...[x'_{l+1}/e'_{l+1}][x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= y[y/z][x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= z[x'_l/e'_l][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= z[z/f(\delta(suc(k,1)),...,\delta(suc(k,r)))][x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= f(\delta(suc(k,1)),...,\delta(suc(k,r)))[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1] \\
&= f(\delta(suc(k,1))[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1],..., \\
&\qquad \delta(suc(k,r))[x'_{l-1}/e'_{l-1}]...[x'_1/e'_1])
\end{aligned}$$

The result is identical to case (a).

Thus the claim is proven. □

With the aid of the above lemmata the correctness of the DAG algorithm is almost clear.

Theorem 4.14 (Correctness of the DAG Algorithm). *Let* $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ *and* $\mathfrak{A}$ *be an interpretation of* Σ*. Then* $\pi \sim_{\mathfrak{A}} T_{DAG}(\pi)$*.*

Proof. It suffices to show that if $\mathfrak{A}[t_\pi(y)] = t_{T_{DAG}(\pi)}(y)$ for all $y \in V_{out}$ also $\mathfrak{A}[\![\pi]\!] = \mathfrak{A}[\![T_{DAG}(\pi)]\!]$ holds. Then by La. 4.12 and 4.13 π and $T_{DAG}(\pi)$ are $\mathfrak{A}$–equivalent.

It is obvious that $\mathfrak{A}[\![t]\!] = \mathfrak{A}[\![\mathfrak{A}[t]]\!]$ for $t \in T_\Sigma$. Hence according to La. 2.14:

$$\begin{aligned}
\mathfrak{A}[\![\pi]\!](\vec{in}) &= (\mathfrak{A}[\![t_\pi(y_1)[x_1/in_1, ..., x_s/in_s]]\!], ..., \mathfrak{A}[\![t_\pi(y_t)[x_1/in_1, ..., x_s/in_s]]\!]) \\
&= (\mathfrak{A}[\![\mathfrak{A}[t_\pi(y_1)][x_1/in_1, ..., x_s/in_s]]\!], ..., \mathfrak{A}[\![\mathfrak{A}[t_\pi(y_t)][x_1/in_1, ..., x_s/in_s]]\!]) \\
&= (\mathfrak{A}[\![t_{T_{DAG}(\pi)}(y_1)[x_1/in_1, ..., x_s/in_s]]\!], ..., \\
&\quad\ \mathfrak{A}[\![t_{T_{DAG}(\pi)}(y_t)[x_1/in_1, ..., x_s/in_s]]\!]) \\
&= \mathfrak{A}[\![T_{DAG}(\pi)]\!](\vec{in})
\end{aligned}$$

□

The idempotency of the extended algorithm is intuitively clear because of its construction, for a detailed proof please refer to [Rie05b]. Thus the DAG algorithm is a program transformation.

5 A DAG–Equivalent Composition of Simple Transformations

Now after the formal introduction of the different optimizing program transformations we will analyze the relations between them. We call two optimizations equivalent if they is optimal w.r.t. each other:

Definition 5.1. Let $T_1, T_2 : \mathcal{SLC} \to \mathcal{SLC}$ be two program transformations.

- T_2 is called T_1*–optimal* ($T_1 \leq T_2$) if $\forall \pi \in \mathcal{SLC} : T_1(T_2(\pi)) = T_2(\pi)$.
- T_1 and T_2 are called *equivalent* if $T_1 \leq T_2$ and $T_2 \leq T_1$.

The following lemma states an important property of the $\leq$ relation on program transformations:

Lemma 5.2. *Let $T_1, T_2, U : \mathcal{SLC} \to \mathcal{SLC}$ be three program transformations. Then*

$$T_1 \leq U \wedge T_2 \leq U \ \Rightarrow\ T_1 \circ T_2 \leq U$$

Proof.

$$\begin{aligned}
&T_1 \leq U \wedge T_2 \leq U \\
&\Rightarrow \forall \pi : T_1(U(\pi)) = U(\pi) \wedge T_2(U(\pi)) = U(\pi) \quad (*) \\
&\Rightarrow \forall \pi : (T_1 \circ T_2)(U(\pi)) = T_1(T_2(U(\pi))) \\
&\qquad\qquad\qquad\qquad\quad \overset{(*)}{=} T_1(U(\pi)) \\
&\qquad\qquad\qquad\qquad\quad \overset{(*)}{=} U(\pi) \\
&\Rightarrow T_1 \circ T_2 \leq U
\end{aligned}$$

□

In the following section we will see that the DAG optimization is optimal w.r.t. the classical transformations. Later we will examine how the classical transformations can influence each other when executed in a certain order.

5.1 Optimality of the DAG Algorithm

In previous examples we have seen that the DAG algorithm had a higher "optimizing potential" than the classical transformations. Now will verify this observation on a generalized basis.

Theorem 5.3. *It holds $T_{DC} \leq T_{DAG}$ (T_{DAG} is T_{DC}–optimal).*

Proof. Let $\pi \in \mathcal{SLC}$ with n instructions and $\pi' := T_{DAG}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ with $\beta = \alpha_1; ...; \alpha_m$ and $\alpha_i := x_i \leftarrow e_i$. Furthermore let $D_\pi = (G, val)$ with $G = (K, L, lab, suc)$ be the DAG of π. Assume that a $j \in \{1, .., m\}$ exists such that $x_j \notin NV_j$ (in π').

We have to distinguish two cases:

1. x_j is not used anymore after α_j ($x_j \notin V_{e_i},\ i > j$).
   Then in D_π a node $\delta^{-1}(x_j)$ exists[21] to which no other edges point, i.e.,
   $$\forall i \in \mathbb{N} \text{ and } \forall k \in K : suc(k, i) \neq \delta^{-1}(x_j)$$
   Moreover x_j is no output variable because otherwise α'_j would not be dead code. Thus no $y \in V_{out}$ exists such that $val(y, n) = \delta^{-1}(x_j)$. Then $\delta^{-1}(x_j)$ is not output relevant by Def. 4.4 and the instruction α_j would have never been created.
2. x_j is redefined in an instruction α_l with $l > j$ ($x_l = x_j$) but $x_j \notin V_{e_i},\ k < i \leq l$.
   Algorithm 4.6 generates output programs in which no variable occurs more than once on the left–hand side of an instruction (SSA form). Therefore a contradiction follows. □

For the Common Subexpression Elimination we get a similar result:

Theorem 5.4. *$T_{CS} \leq T_{DAG}$.*

Proof. Let π, π' and D_π be given as in the proof of Thm. 5.3. Similarly we will prove the claim by contradiction:

Assumption: $\exists i, j \in \{1, ..., m\}$ with $e_i = e_j = f(u_1, ..., u_r)$ such that $j \in vr(i)$, i.e., e_j is a valid recurrence of e_i. T_{CS} would eliminate it using a temporary variable.

Since Alg. 4.6 creates for every operation node in the DAG exactly one instruction, two nodes $k, k' \in K$ must exist where $suc(k, j) = suc(k', j)$ for all $j \in \{1, ..., r\}$ and $lab(k) = lab(k') = f \in F^{(r)}$.

According to the DAG construction (Alg. 4.2, subcase 3(b)(b1)) however this is impossible because after k has been created no k' with the above properties would have been inserted in the DAG. Instead only the val–function would have been updated. Thus we have a contradiction. □

Finally we will verify the optimality of the DAG algorithm w.r.t. Constant Folding.

Theorem 5.5. *$T_{CF} \leq T_{DAG}$.*

[21] Let δ be the substitution that one obtains after complete execution of Alg. 4.6.

Proof. It suffices to show that for all $i \in \{1, ..., n\}$ and all $y \in V_\pi$:

$$RD_i(y) \in A \Rightarrow val(y, i-1) \in A$$

Then the Constant Folding will not provoke any further change when applied to π' (since T_{DAG} already utilized the full "replacement potential"). Complete induction on $i \in \{1, ..., n\}$:

$i = 1 :\ RD_1(y) = \sigma_\perp(y) = \perp$
$i \to i+1 :$ Case distinction for the variable y:

1. If $y = x_i$:

$$\begin{aligned} RD_{i+1}(y) &= RD_i[x_i / \bar{\mathfrak{A}}[\![e_i]\!] RD_i](y) \\ &= \bar{\mathfrak{A}}[\![e_i]\!] RD_i \in A \end{aligned}$$

And we get:

$$\begin{aligned} &\forall z \in V_{e_i} :\ RD_i(z) \in A \\ \overset{\text{IH}}{\Rightarrow}\ &\forall z \in V_{e_i} :\ val(z, i-1) \in A \\ \Rightarrow\ &val(y, i) \in A \text{ (according to the DAG construction)} \end{aligned}$$

2. Otherwise ($y \neq x_i$):

$$\begin{aligned} RD_{i+1}(y) &= RD_i[x_i / \bar{\mathfrak{A}}[\![e_i]\!] RD_i](y) \\ &= RD_i(y) \in A \\ &\overset{\text{IH}}{\Rightarrow} val(y, i-1) \in A \\ &\Rightarrow val(y, i) \in A \text{ (since } y \neq x_i) \end{aligned}$$

Due to the induction principle the claim holds. □

Thus the DAG algorithm is optimal w.r.t. the classical transformations. 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.

5.2 Copy Propagation

The first result is that the classical transformations do not suffice for "simulating" the DAG optimization.

Theorem 5.6. *There exists a $\pi \in \mathcal{SLC}$ such that π is T–optimal for every $T \in \{T_{DC}, T_{CS}, T_{CF}\}$, but π is not T_{DAG}–optimal.*

Proof. Consider the program depicted in Fig. 3. For this program none of the classical optimizations will provoke any change:

- In π exists no dead code, all instructions are needed for computing the value of the output variable y.
- All expressions in π are distinct. Hence T_{CS} does not influence the result.
- The absence of constants implies that Constant Folding would have no effect.

Thus π is optimal w.r.t. all three classical transformations. An application of T_{DAG} yields however:

$$\begin{aligned} v_1 &\leftarrow x + y; \\ u &\leftarrow x * v_1; \\ v &\leftarrow u + x; \end{aligned}$$

□

$$\begin{array}{rl} \vec{v}_{in}: & (x, y) \\ \beta: & u \ \leftarrow x; \\ & y \ \leftarrow u + y; \\ & v \ \leftarrow u; \\ & u \ \leftarrow v * y; \\ & v \ \leftarrow u + v; \\ \vec{v}_{out}: & (u, v) \end{array}$$

Fig. 3. $\pi = (\Sigma_{\text{arithm}}, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with copy instructions

In the example Dead Code Elimination would be applicable (in the sense of eliminating instructions) 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 does not improve the input program directly but is a *preprocessing* 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.

Definition 5.7 (Valid Copies). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta := \alpha_1; ...; \alpha_n$. For an instruction $\alpha = x \leftarrow e$ we define the transfer function $t_\alpha : V_\pi^2 \times \{1, ..., n\} \to V_\pi^2 \times \{1, ..., n\}$ by

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

Here $trans : V^2 \times \mathbb{N} \to V^2 \times \mathbb{N}$ computes the *transitive closure* of the first two arguments for an $M \subseteq V^2 \times \mathbb{N}$:

$$\begin{aligned} trans(M) := \{(x, y, d) \mid \exists k \in \mathbb{N} : \exists z_1, ..., z_k \in V,\ \exists d_0, ..., d_k \in \mathbb{N} \text{ such that} \\ \{(x, z_1, d_0), (z_1, z_2, d_1), ..., (z_{k-1}, z_k, d_{k-1}), (z_k, y, d_k)\} \subseteq M\} \\ \text{where } d := \sum_{i=0}^{k} d_i \end{aligned}$$

In particular all $(x, y, d) \in M$ are included in $trans(M)$ for $k = 0$. Now the analysis sets $CP_i \subseteq V_\pi^2 \times \{1, ..., n\}$ can be computed inductively:

$$\begin{aligned} CP_1 &:= \emptyset \\ CP_{i+1} &:= t_{\alpha_i}(CP_i) \text{ for } i \in \{1, ..., n-1\} \end{aligned}$$

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

Definition 5.8 (Copy Propagation). The transformation $T_{CP} : \mathcal{SLC} \to \mathcal{SLC}$ for $\pi = (\Sigma, V_{in}, V_{out}, \beta) \in \mathcal{SLC}$ with $\beta = x_1 \leftarrow e_1; ...; x_n \leftarrow e_n$ and $\Sigma = (F, C)$ is given by

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

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

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

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. This is necessary since, due to the computation of the transitive closure, for a variable x several tuples of the form (x, y, d) with different y and d might exist.

Example 5.9. Applying the 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	$u \leftarrow v * y;$	$\{(u, x, 1), (v, u, 1), \boldsymbol{(v, x, 2)}\}$	$u \leftarrow x * y;$
5	$v \leftarrow u + v$	$\{\boldsymbol{(v, x, 2)}\}$	$v \leftarrow u + x;$

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. It is clear that on the above result Dead Code Elimination would remove the first and the third instructions.

5.3 Execution Order

Now that we have introduced Copy Propagation we will analyze the relations between the simple algorithms[22]. Particularly interesting is if, given a program which is optimal w.r.t. an optimization T_1, the application of an algorithm T_2 will yield additional optimization potential for T_1 afterwards.

[22] With "simple" algorithms we refer to the classical transformations and Copy Propagation.

Definition 5.10. Let $T_i : \mathcal{SLC} \to \mathcal{SLC},\ i \in \{1,2\}$ be two program transformations. If there exists a $\pi \in \mathcal{SLC}$ such that

$$T_1(\pi) = \pi \ \wedge \ T_1(T_2(\pi)) \neq T_2(\pi)$$

T_2 is called T_1–enabling ($T_2 \to T_1$).

The enabling–relations that hold for our transformations are given in Fig. 4. At this point we will not give a detailed proof for every combination but a short explanation (for a detailed analysis refer to [Rie05b]):

- Dead Code Elimination does not enable any of the other transformations because it eliminates instructions and therefore does not create any available expressions, copy instructions or constant variables.
- Common Subexpression Elimination 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. 3.15 that Constant Folding "produces" additional Dead Code. One can easily construct examples in which the substitution of variables with constants creates common subexpressions.
- From Ex. 5.9 it is clear that T_{CP} is T_{DC}–enabling. Similarly to Constant Folding the substitution of variables by others can create common subexpressions.

	T_{DC}	T_{CS}	T_{CF}	T_{CP}
T_{DC}		-	-	-
T_{CS}	-		-	$\to$
T_{CF}	$\to$	$\to$		-
T_{CP}	$\to$	$\to$	-	

Fig. 4. Influence Relations between the Simple Transformations

Between Copy Propagation and Common Subexpression Elimination there is a mutual dependence. Thus a repeated application of both is unavoidable. The question is if a constant amount of iterations suffices.

Theorem 5.11. *There exists a sequence of SLC–programs $(\pi_n)_{n\in\mathbb{N}\setminus\{0\}}$ 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. Let $\pi_n := (\Sigma, (x), (y,z), \beta_n)$ with $\Sigma = (F, \emptyset)$ and $F = \{f^{(2)}\}$ where β_n is given by:

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

It is clear that $T^{n+1}(\pi_n) = T^n(\pi_n)$ but $\nexists i < n : T^{i+1}(\pi_n) = T^i(\pi_n)$. $\square$

Thus there is no fixed constant (independent from program length) that could restrict the number of required iterations.

Theorem 5.12. *Let* $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ *with* $\beta := \alpha_1; ...; \alpha_n$ *and* $\alpha_i := x_i \leftarrow e_i$ *as well as*

$$m := \left| \{i \in \{1, ..., n\} \mid e_i = f(u_1, ..., u_r) \text{ for an } f \in F^{(r)} \text{ and } u_j \in V \cup C\} \right|.$$

Then for $T := T_{CS} \circ T_{CP}$*:*

$$T^{m+1}(\pi) = T(T^{m+1}(\pi))$$

Proof. m represents the number of operation expressions in the program π. After every application of Common Subexpression Elimination with $T_{CS}(\pi) \neq \pi$ the program contains at least one operation expression less.

Since T_{CP} is idempotent it follows that when T_{CS} does not provoke any further change also two subsequent applications of T_{CP} will not create any additional optimization potential w.r.t. T_{CS} (this holds analogously for the symmetric case that we obtain a T_{CP}–optimal program first).

Thus after a maximum of $m+1$ iterations a fixed point of T is reached because the first application of T_{CS} remains possibly ineffective. □

The above bound eventually can be further refined. Since T_{CS} is optimizing at least two operation expressions with every "successful" application and the expressions assigned to the temporary variables should not have additional valid recurrences it is presumable that $\lfloor \frac{m}{2} \rfloor + 1$ applications of T are sufficient.

In practice it certainly is not advisable to use "blindly" worst–case iteration number but a demand–driven method.

Definition 5.13. For $\pi \in \mathcal{SLC}$ and $T := T_{CP} \circ T_{CS}$ the transformation $T_{CPCS} : \mathcal{SLC} \to \mathcal{SLC}$ is be defined by:

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

T_{CS} and T_{CP} are applied in alternation until a fixed point – that exists by Thm. 5.12 – is reached.

From the previous observations, represented in Fig. 4, we now can 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})

A different order would be unfavorable, e.g. executing Dead Code Elimination at the beginning would cause the dead code created by the other transformations to remain. It though seems reasonable to apply Dead Code Elimination immediately after Constant Folding (before T_{CPCS}) since due to the removal of instructions the fixed point of T_{CPCS} could be eventually reached faster. This would however increase the complexity especially regarding proofs.

Definition 5.14 (Compositional Optimization). The compositional transformation incorporating the classical optimizations and Copy Propagation is given by $T_{COPT} := T_{DC} \circ T_{CPCS} \circ T_{CF}$.

Theorem 5.15. *T_{COPT} is an $\mathfrak{A}$–program transformation.*

Proof. According to Def. 3.1 we have to show three properties:

1. T_{COPT} is correct.
   Since T_{COPT} is constructed from correct subtransformations the compositional transformation is also correct.
2. T_{COPT} is idempotent.
   T_{CF} cannot be enabled from any other subtransformation according to Fig. 4. Therefore
   $$T_{CF}(T_{COPT}(\pi)) = T_{COPT}(\pi) \quad (*)$$
   The following application of T_{CPCS} cannot provoke further changes because of $(*)$, the idempotency of T_{CPCS} (by definition) and since neither $T_{DC} \to T_{CS}$ nor $T_{DC} \to T_{CP}$ holds (Fig. 4). Hence we obtain:
   $$T_{CPCS}(T_{CF}(T_{COPT}(\pi))) = T_{COPT}(\pi) \quad (**)$$
   Dead Code Elimination does neither change the program (due to $(**)$ and the idempotency of T_{DC}). Thus T_{COPT} is idempotent.

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

Fig. 5. Application of T_{COPT} to π from Fig. 1

Example 5.16. Figure 5 shows an exemplary computation of T_{COPT} for our example program from Fig. 1 starting from the already T_{CF}–optimized program from Ex. 3.15. 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 Ex. 4.8 except for the variable naming. Also for the example from Fig. 3 we would get the same results "modulo" variable names.[23]

[23] For testing these results see [Rie05a].

5.4 Simulation of the DAG Transformation

For achieving a T_{DAG}–optimal transformation we obviously need to rename the variables in the output program. Otherwise the DAG algorithm would change the variable names and thus the program would not be T_{DAG}–optimal according to Def. 5.1.

Obtaining Static Single Assignment Form

We will use a transformation that transforms any SLC–program in static single assignment form and uses a variable naming according to the DAG code generation (Alg. 4.6).

Definition 5.17 (SSA Transformation). Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta := \alpha_1; ...; \alpha_n$ and $\alpha_i := x_i \leftarrow e_i$. Define for $i \in \{0, ..., n\}$ substitution functions $\rho_i : V \to V$:

$$\rho_0 = id$$

and for $i \in \{1, ..., n\}$:

$$\rho_i = \begin{cases} \rho_{i-1}[x_i/x_i] & \text{if } x_i \in V_{out} \text{ and } x_i \neq x_j \text{ for } j > i \\ \rho_{i-1}[x_i/v_i] & \text{else} \end{cases}$$

Here the v_i are mutually distinct variables that do not occur in π. Then the transformation $T_{SSA} : \mathcal{SLC} \to \mathcal{SLC}$ is computed as follows:

$$\begin{aligned} T_{SSA}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta') \text{ with } \beta' := \rho_1(x_1) &\leftarrow \bar{\rho}_0(e_1); \\ &\vdots \\ \rho_k(x_k) &\leftarrow \bar{\rho}_{k-1}(e_k); \\ &\vdots \\ \rho_n(x_n) &\leftarrow \bar{\rho}_{n-1}(e_n); \end{aligned}$$

$\bar{\rho}_i$ denotes the application of the substitution ρ_i to expressions (as expected).

T_{SSA} renames all left–hand side variables – except output variables in their final assignment – in new variables subscripted with the index of the instruction of their first appearance (analogously to the DAG code generation). Obviously the resulting program is in SSA form.

Example 5.18. Let us demonstrate the definition by an example. For this we use the program $T_{COPT}(\pi)$ from Fig. 5. For the ρ_i we will only represent those arguments that are not projected on themselves (remind: $\vec{v}_{out} = (u, v)$):

i	$T_{COPT}(\pi)$	ρ_i	$T_{SSA}(T_{COPT}(\pi))$
1	$t_2 \leftarrow x - y;$	$\{t_2 \mapsto v_1\}$	$v_1 \leftarrow x - y;$
2	$v \leftarrow 3;$	$\{t_2 \mapsto v_1\}$	$v \leftarrow 3;$
3	$u \leftarrow t_2 - y;$	$\{t_2 \mapsto v_1, u \mapsto v_3\}$	$v_3 \leftarrow v_1 - y;$
4	$u \leftarrow 2 * u;$	$\{t_2 \mapsto v_1\}$	$u \leftarrow 2 * v_3;$

Now the result is identical to $T_{DAG}(\pi)$ (Ex. 4.8). The idempotency of the DAG optimization implies $T_{DAG}(T_{SSA}(T_{COPT}(\pi))) = T_{SSA}(T_{COPT}(\pi))$ for this example program.

We will not analyze the properties of the SSA transformation in detail. It is clear that T_{SSA} is a program transformation. Since the transformation has been developed for obtaining "DAG–compatible" variable names T_{DAG} is optimal w.r.t. T_{SSA}.

Also for iterative programs an SSA form is computable [AH00,CFR$^+$91] which is helpful for a variety of optimizations.

Weakness of Copy Propagation

Now, that we have achieved the "output compatibility" with the DAG transformation one could assume that $T_{SSA} \circ T_{COPT}$ is equivalent to T_{DAG}.

Theorem 5.19. *$T_{DAG} \not\leq T_{SSA} \circ T_{COPT}$.*

Proof. Consider the following SLC–program:

$$\pi := (\Sigma_{\text{arithm}}, (x), (y), \beta) \text{ with } \beta := \begin{array}{l} t \leftarrow x; \\ x \leftarrow -x; \\ y \leftarrow x + t; \end{array}$$

Applying $T_{SSA} \circ T_{COPT}$ on π and T_{DAG} to the result yields:

$T_{SSA}(T_{COPT}(\pi))$	$T_{DAG}(T_{SSA}(T_{COPT}(\pi)))$
$v_1 \leftarrow x;$	
$v_2 \leftarrow -x;$	$v_1 \leftarrow -x;$
$y \leftarrow v_2 + v_1;$	$y \leftarrow v_1 + x;$

Thus the claim is proven. □

The cause of this problem is that Copy Propagation prevents further optimization. t is obviously a copy of x, but because x is overwritten in the second instruction the corresponding analysis tuple $(t, x, 1)$ is not anymore contained in CP_2. If one would want to substitute the occurrence of t in the last instruction a renaming of the left–hand x in the second instruction and its later occurrences would be necessary.

This weakness of Copy Propagation can prevent further optimizations like Dead Code Elimination which is the case in the above proof. The DAG optimization avoids this problem by generating the output program from the DAG anew using a new variable for every assignment.

Transforming the input program in SSA form before processing it with T_{COPT} solves this problem. Since neither T_{DC} nor T_{CS} affects the SSA property T_{CP} receives an input–program in SSA form. If T_{CP} is applied to a program in SSA form tuples are never deleted from the CP–sets since no variable is redefined.

Thus we now have the following new execution order:

$$T_{COPT'} := T_{SSA} \circ T_{COPT} \circ T_{SSA} = T_{SSA} \circ T_{DC} \circ T_{CPCS} \circ T_{CF} \circ T_{SSA}$$

Unremovable Copy Instructions

Even the new $T_{COPT'}$ composition is insufficient for simulating the DAG optimization.

Theorem 5.20. $T_{DAG} \not\lesssim T_{COPT'}$.

Proof. Consider the program π_2 from the proof of Thm. 5.11. Applying $T_{COPT'}$ to π_2 and then T_{DAG} to the result yields:

$T_{COPT'}(\pi_2)$	$T_{DAG}(T_{COPT'}(\pi_2))$
$v_1 \leftarrow f(x,x);$	$v_1 \leftarrow f(x,x);$
$v_2 \leftarrow f(v_1,x);$	$y \leftarrow f(v_1,x);$
$y \leftarrow v_2;$	
$z \leftarrow v_2;$	$z \leftarrow y;$

The problem is that Common Subexpression Elimination can insert copy instructions involving output variables. These cannot be eliminated using the conventional algorithms. The DAG optimization is avoiding this problem by directly renaming output nodes with output variables. □

Unremovable copy instructions can occur at maximum once for every output variable and can be eliminated using a specialized algorithm. The analysis is operating backwards:

Definition 5.21. Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$, $\Sigma := (F, C)$, $\beta := \alpha_1; ...; \alpha_n$ and $\alpha_i := x_i \leftarrow e_i$. For an instruction α_i and $I := \mathfrak{P}((V_\pi^2 \times \{1, ..., n\}) \cup V_\pi)$ define the transfer function $t_{\alpha_i} : I \to I$ by:

$$t_{\alpha_i} := gen_{\alpha_i} \circ kill_{\alpha_i} \text{ with } kill, gen : I \to I \text{ and}$$

$$kill_{\alpha_i}(M) := M \setminus \{(y,z,j) \in M \mid x_i = y \text{ or } z \in V_{\alpha_i},\ j \in \{1, ..., n\}\}$$
$$gen_{\alpha_i}(M) := M \cup \{(y, x_i, i) \mid e_i = y \in V \setminus (M \cup V_{out}) \text{ and } x_i \in V_{out} \setminus M\} \cup \{x_i\}$$

The $t_{\alpha_n}, ..., t_{\alpha_1}$ determine – starting from the initial set $\emptyset$ – the analysis sets:

$$RC_n := \emptyset$$
$$RC_i := t_{\alpha_{i+1}}(RC_{i+1}) \text{ for } i \in \{n-1, ..., 1\}$$

Thus the analysis set RC_i of an instruction α_i contains the tuple (y, x, j) it exists an instruction α_j, $j > i$ that is an assignment of the form $x \leftarrow y$ with $x \in V_{out}$ and $y \notin V_{out}$, y is not redefined after α_i, x neither appears on the left–hand side of an instruction after α_i except in α_j, nor it is used between α_i and α_j. The third entry j is employed to guarantee the determinism of the algorithm.

The additional “collection” of defined variables is used to simply check if a variable is redefined later (after α_j).

Algorithm 5.22 (Reverse Copy Propagation). Let π be given as in Def. 5.21 and let $\delta : V_\pi \to V_\pi$ be a variable substitution function. Define $T_{RC} : \mathcal{SLC} \to \mathcal{SLC}$ by $T_{RC}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ where β' is computed as follows:

1. Set $\delta := id$ and $\beta' := \varepsilon$.
2. For $i = 1, ..., n$ (case distinction by the type of e_i):
   (a) $e_i = f(u_1, ..., u_r)$ for $f \in F^{(r)}$:
   If a $y \in V_{out}$ exists such that $(x_i, y, j) \in RC_i$ (if several such y exist choose[24] the tuple with the lowest j) and set

$$\beta' := \beta' \cdot y \leftarrow \bar{\delta}(e_i)$$
$$\delta := \delta[x_i/y]$$

[24] Necessary for achieving determinism.

(b) Otherwise: further cases
   (i) $e_i = y \in V$ and $\delta(y) = \delta(x_i)$: no change of β'.
   (ii) Else: set $\beta' := \beta' \cdot \bar{\delta}(\alpha_i)$.

Unlike Copy Propagation, Reverse Copy Propagation does not trace transitive relations. For eliminating the problem seen in the proof of Thm. 5.20 this is not needed.

Example 5.23. To clarify the functionality of the algorithm we will demonstrate it using as example the T_{COPT}–optimized version of the program π_2 from the proof of Thm. ThmNxCPCS, i.e., $\pi := T_{COPT}(\pi_2) := ((\{f^{(2)}\}, \emptyset), (x), (y, z), \beta)$ with

$$\begin{aligned} \beta := t_1 &\leftarrow f(x, x); \\ t_4 &\leftarrow f(t_1, x); \\ y &\leftarrow t_4; \\ z &\leftarrow t_4; \end{aligned}$$

Applying T_{RC} we get the following computation (let $\pi' := T_{RC}(\pi) := ((\{f^{(2)}\}, \emptyset), (x), (y, z), \beta'))$:

i	β	RC_i	δ	β'
1	$t_1 \leftarrow f(x, x);$	$\{t_4, y, z\}$	id	$t_1 \leftarrow f(x, x);$
2	$t_4 \leftarrow f(t_1, x);$	$\{(t_4, y, 3), (t_4, z, 4), y, z\}$	$id[t_4/y]$	$y \leftarrow f(t_1, x);$
3	$y \leftarrow t_4;$	$\{(t_4, z, 4), z\}$	$id[t_4/y]$	
4	$z \leftarrow t_4;$	$\emptyset$	$id[t_4/y]$	$z \leftarrow y;$

The application of $T_{SSA} \circ T_{DAG}$ to π' yields again the same program. Thus for this example we achieved the desired result introducing Reverse Copy Propagation.

Reverse Copy Propagation only substitutes variables that are not output variables. This is indispensable for the completeness of the program because otherwise the removal of an assignment to an output variable would cause an undefined output variable.

As all the other optimizations regarded in this work also T_{RC} satisfies the requirements of a program transformation and the DAG algorithm is as expected T_{RC}–optimal [Rie05b].

Extended Compositional Composition

Now the prerequisites are met to define a new transformation that hopefully is DAG–equivalent.

Definition 5.24 (Extended Compositional Transformation). For $\pi \in \mathcal{SLC}$ the Extended Compositional Transformation $T_{XOPT} : \mathcal{SLC} \to \mathcal{SLC}$ is defined by:

$$T_{XOPT} := T_{COPT'} \circ T_{SSA} = T_{SSA} \circ T_{RC} \circ T_{DC} \circ \underbrace{(T_{CP} \circ T_{CS})^\star}_{T_{CPCS}} \circ T_{CF} \circ T_{SSA}$$

(The $\star$–notation indicates the fixed point iteration)

At this point we will not show that T_{XOPT} fulfills the requirements of a program transformation. The reader may find the details in [Rie05b].

5.5 Equivalence of the DAG Algorithm and the Extended Compositional Transformation

Now we have to verify that the changes of T_{COPT} do indeed suffice to make the Extended Compositional Transformation T_{XOPT} equivalent to the DAG optimization for which we first have to show some auxiliary propositions.

Lemma 5.25. *Let π be a T_{DC}–optimal SLC–program and $D_\pi = (G, val)$ with $G = (K, L, lab, suc)$ its DAG. Then all $k \in K$ are output relevant.*

Proof. Let $\pi =: (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ with $\beta = \alpha_1; ...; \alpha_n$ and $\alpha_i = x_i \leftarrow e_i$. $x_i \in NV_i$ by the assumption that π is optimal w.r.t. T_{DC}. For $x \in V_\pi$ and $i \in \{1, ..., n\}$ it suffices to show:

$$x \in NV_i \Rightarrow val(x, i) \text{ is output relevant}$$

Backward induction on $i \in \{n, ..., 1\}$:

$i = n:$
$x \in NV_n = V_{out} \Rightarrow val(x, n)$ is output relevant according to Def. 4.4 (case 1).
$i \rightarrow i - 1:$
$x \in NV_{i-1} = NV_i \backslash \{x_i\} \cup V_{e_i}$ since $x_i \in NV_i$ by assumption. Case distinction:

1. $x \in V_{e_i}$ and $e_i = f(u_1, ..., u_r)$. Then $\exists j \in \{1, ...r\}$ and $suc(val(x_i, i), j) = val(x, i - 1)$. Because of the induction hypothesis and Def. 4.4 (case 2) follows: $val(x, i - 1)$ is output relevant.
2. $x \in V_{e_i}$ and $e_i = x$. This is inadmissible for SLC–programs (Def. 2.3).
3. $x \in NV_i \backslash \{x_i\}$. With the induction hypothesis follows the output relevance of $val(x, i)$. Due to $x_i \neq x$, $val(x, i - 1)$ is also output relevant according to Alg. 4.2. □

We will now show that the analysis information of Constant Folding is at least as precise as the one of T_{DAG}. For that we will prove the contrary to the claim of Thm. 5.5.

Lemma 5.26. *Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ with $\beta := \alpha_1; ...; \alpha_n$ and $D_\pi = (G, val)$ where $G = (K, L, lab, suc)$. Then for all $i \in \{1, ..., n\}$ and all $y \in V_\pi$:*

$$val(y, i - 1) \in A \Rightarrow RD_i(y) \in A$$

Proof. Induction on $i \in \{1, ..., n\}$:

$i = 1: \ RD_1(y) = \sigma_\perp(y) = \perp$
$i \rightarrow i + 1:$ Case distinction for the expression e_i and the variable y:

1. If $e_i = f(u_1, ..., u_r)$ and $y = x_i$:

$$val(y, i) = \varphi(f)(u'_1, ..., u'_r) \in A$$
$$\text{where } u'_j := \begin{cases} val(u_j, i - 1) & \text{if } u_j \in V \\ u_j & \text{if } u_j \in A \end{cases}$$

Then it follows:

$$\begin{aligned} & \forall j \in \{1, ..., r\} : \ u'_j \in A \\ \overset{\text{IH}}{\Rightarrow} & \forall j \in \{1, ..., r\} : \ RD_i(u_j) \in A \text{ if } u_j \in V \text{ or } u_j \in A \\ \Rightarrow & RD_{i+1}(y) = RD_i[x_i / \bar{\mathfrak{A}}[\![e_i]\!] RD_i](y) = \bar{\mathfrak{A}}[\![e_i]\!] RD_i \in A \end{aligned}$$

2. If $e_i = z \in V$ and $y = x_i$:

$$\begin{aligned} & val(y,i) = val(z,i-1) \in A \\ \overset{\text{IH}}{\Rightarrow}\ & RD_i(z) \in A \\ \Rightarrow\ & RD_{i+1}(y) = RD_i[x_i/\bar{\mathfrak{A}}[\![e_i]\!]RD_i](y) = \bar{\mathfrak{A}}[\![e_i]\!]RD_i = RD_i(z) \in A \end{aligned}$$

3. If $e_i = a \in A$ and $y = x_i$:

$$\begin{aligned} val(y,i) &= a \\ &= RD_i[x_i/\bar{\mathfrak{A}}[\![e_i]\!]RD_i](y) \\ &= RD_{i+1}(y) \end{aligned}$$

4. Otherwise ($y \neq x_i$):

$$\begin{aligned} & val(y,i) = val(y,i-1) \in A \\ \overset{\text{IA}}{\Rightarrow}\ & RD_i(y) \in A \\ \Rightarrow\ & RD_{i+1}(y) = RD_i[x_i/\bar{\mathfrak{A}}[\![e_i]\!]RD_i](y) = RD_i(y) \in A \end{aligned}$$

Thus the claim holds. □

We also have to check if T_{CPCS} is working optimally for an input program in SSA form in the sense that the DAG construction for a $T_{CPCS} \circ T_{CF}$–optimal SLC–program causes the creation of a new node for every operation expression. If this property would not hold, neither $T_{XOPT} \sim T_{DAG}$ would. Constant Folding has to be executed in advance since also the DAG algorithm is performing a partial evaluation of expressions using knowledge about variables with constant values.

Lemma 5.27. *Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ be in SSA form with $\Sigma := (F, C)$, $\beta := \alpha_1; ...; \alpha_n$, $\alpha_i := x_i \leftarrow e_i$, $T_{CPCS}(T_{CF}(\pi)) = \pi$ and let $D_\pi = (G, val)$ the DAG of π with $G := (K, L, lab, suc)$ and $\mathfrak{A} := (A, \varphi)$ an interpretation of Σ. Then for every $e_i \notin V \cup A$ it exists a $k_i \in K$.*

Proof. Suppose for an $l \in \{1, ..., n\}$ with $e_l = f(u_1, ..., u_r)$, $f \in F^{(r)}$ no $k_l \in K$ exists. Then by the DAG construction (Alg. 4.2) an $l' < l$ exists such that $k_{l'} \in K$ with

$$suc(k_{l'}, j) = \begin{cases} val(u_j, l) & \text{if } u_j \in V \\ u_j & \text{if } u_j \in A \end{cases} \qquad (*)$$

After La. 5.26 $val(u_j, l) \in A$ cannot hold for all $j \in \{1, ..., r\}$ with $u_j \in V$ since π is optimal w.r.t. Constant Folding. Let $\alpha_{l'}$ be of the form $x_{l'} \leftarrow f(u'_1, ..., u'_r)$.

Case distinction (inductively):

1. $\forall j \in \{1, ..., r\} :\ u'_j = u_j \in A \cup V$.
   Then it follows $l' \in AE_l$ and $l \in vr(l')$. But then $T_{CS}(\pi) = \pi$ would not hold and neither would $T_{CPCS}(T_{CF}(\pi)) = \pi$. Contradiction.
2. $\exists j \in \{1, ..., r\}$ such that $u_j \neq u'_j$ and $u_j, u'_j \in V$ according to La. 5.26. Then we have different cases:
   (a) It exists a $k < l$ with $\alpha_k = u_j \leftarrow z$, $z \in V$ and $val(z, k-1) = val(u_j, l)$ or $val(z', k'-1) = val(u'_j, l')$ holds for a $k' < l'$ with $\alpha_{k'} = u'_j \leftarrow z'$. Neither is possible since Copy Propagation would have substituted u_j and u'_j by z and z' because of the SSA form of the input program.

(b) $\exists k < l$ and $\exists k' < l'$ with $\alpha_k = u_j \leftarrow g(s_1, ..., s_p)$ and $\alpha_{k'} = u'_j \leftarrow g(s'_1, ..., s'_p)$ and it holds again $(*)$ with $u_j = s_j$, $u'_j = s'_j$, $f = g$, $l = k$, $l' = k'$ and $r = p$.

Because $n < \infty$ the induction must terminate; this can only occur in the cases 1 or 2(a). Hence we get a contradiction. □

A T_{XOPT}–optimal program cannot contain arbitrary copy instructions; they must follow a certain form.

Lemma 5.28. *Let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{SLC}$ be a T_{XOPT}–optimal program with $\beta := \alpha_1; ...; \alpha_n$ and $\alpha_i := x_i \leftarrow e_i$. Then for all $i \in \{1, ..., n\}$:*

$$\alpha_i = x \leftarrow y \quad \Rightarrow \quad x \in V_{out} \wedge y \in V_{in} \cup V_{out}$$

Proof.

1. x must be an output variable since otherwise all occurrences of x in α_j, $j > i$ would have been substituted by y due to the SSA property of π and α_i therefore would have become Dead Code. This is a contradiction to the T_{XOPT}–optimality.
2. Let $y \notin V_{in} \cup V_{out}$. Then an operation instruction $\alpha_{i'} = y \leftarrow f(u_1, ..., u_r)$ must exist for $i' < i$. A constant assignment is impossible because of the use of T_{CF} and a copy instruction is impossible according to case 1.
   The SSA property of π ensures that neither x nor y are ever assigned a value except in α_i and $\alpha_{i'}$. Furthermore y does not occur above α_i in the program. Then T_{RC} would substitute y by x and the instruction α_i would be removed. That is a contradiction to the optimality w.r.t. T_{XOPT}. □

Now all prerequisites for the proof of the T_{DAG}–optimality of the Extended Compositional Transformation are fulfilled.

Theorem 5.29 (Optimality of T_{XOPT} w.r.t. T_{DAG}). $T_{DAG} \leq T_{XOPT}$.

Proof. For $\pi_0 \in \mathcal{SLC}$ and $\Sigma := (F, C)$ let $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) := T_{XOPT}(\pi_0)$ with $\beta := \alpha_1; ...; \alpha_n$. By $\mathfrak{A} := (A, \varphi)$ an interpretation of Σ is given. In addition let $D_\pi := (G, val)$ with $G := (K, L, lab, suc)$ be the DAG of π and let $T_{DAG}(\pi) =: \pi' =: (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ with $\beta' := \alpha'_1; ...; \alpha'_n$ be the DAG–optimized version of π.

Because of the final application of T_{SSA} the program π is in SSA form. Therefore for all $i \in \{1, ..., n\}$ and all $v \in V_\pi$: if $val(v, i) \in K$ (function value is defined) $val(v, i) = val(v, n)$ holds. After La. 5.25 K does only contain output–relevant nodes, thus Alg. 4.6 is directly applicable (without eliminating "dead" nodes).

We show that $\alpha_i = \alpha'_i$ must hold for all $i \in \{1, ..., n\}$. Case distinction for $\alpha_i = x \leftarrow e$:

1. $e = a \in A$: this case is only possible if $x \in V_{out}$ since otherwise all later occurrences[25] of x would have been replaced by a and α_i therefore would be Dead Code.
   According to Alg. 4.2 then $a \in K$, $val(x, i) = val(x, n) = a$ and $last(x) = i$ since later no redefinition of x is possible.[1] Hence by Alg. 4.6 while choosing the next node k_l, $l > i$ before the code generation for k_l (step 3) first an assignment to y must be created, i.e., $\alpha'_i = x \leftarrow a = \alpha_i$. If no operation node as above exists we obtain the same result with step 8.

[25] Because of the SSA form of π, x is never redefined.

2. $e = y \in V$: according to La. 5.28 $x \in V_{out}$ and we have get two cases:
   (a) $y \in V_{in}$: $val(x,i) = y$ and $last(x) = i$. Analogously to the first case according to step 3 or 8 of the algorithm for the DAG code generation we obtain $\alpha'_i = x \leftarrow y$.
   (b) $y \in V_{out}$: then there is an instruction $\alpha_l = y \leftarrow f(u_1,...,u_r)$ with $l < i$ in π (since constant assignments to y and further copy instructions are obviously not possible) and by La. 5.27 a node k_l representing α_l has been created. Thus it holds $val(x,i) = val(y,i-1) = k_l$, $last(y) = l$ and $last(x) = i$. In π' an operation instruction for k_l with y as assignment variable already exists (this is α_l). Hence, according to step 3 or 8 analogously to the first case $\alpha'_i := x \leftarrow y$ is set.
3. $e = f(u_1,...,u_r)$ for an $f \in F^{(r)}$: by La. 5.27 a node $k_i \in K$ exists with

$$\begin{aligned} lab(k_i) &= f \\ suc(k_i,j) &= u'_j \text{ with } u'_j = \begin{cases} val(u_j,i-1) & \text{if } u_j \in V \\ u_j & \text{else} \end{cases} \\ val(x,i) &= k_i \end{aligned}$$

   Further case distinction by the type of y:
   (a1) $x \in V_{out}$: $last(x) = i$ because of the SSA property of π. Hence x is the entry with the lowest $last$–value of all $V_{out}^{k_i}$ in step 4 (since the other $z \in V_{out}^{k_i}$ cannot obtain a value before x). Thus $\delta := \delta[k_i/x]$ is set and the following code is generated by step 6:

$$\begin{aligned} \alpha_i &= \delta(k_i) \leftarrow f(\delta(suc(k_i,1)),...,\delta(suc(k_i,r))) \\ &= x \leftarrow f(\delta(suc(k_i,1)),...,\delta(suc(k_i,r))) \end{aligned}$$

   (b1) $x \notin V_{out}$: thus $x = v_i$ (because of the final application of T_{SSA}). Now $V_{out}^{k_i} = \emptyset$ because for an output variable $z \neq x$ with $val(z,n) = k_i$ it holds:
      – A value assignment to z via a copy chain is impossible since due to $v_i \notin V_{in} \cup V_{out}$ La. 5.28 would be violated.
      – An assignment of an operation expression equivalent to α_i to z is neither possible since π is optimal w.r.t. T_{CPCS}.

      Because of $\delta(k_i) \notin V_{out}$ in step 5 $\delta := \delta[k_i/v_i]$ is set. Then:

$$\begin{aligned} \alpha_i &= \delta(k_i) \leftarrow f(\delta(suc(k_i,1)),...,\delta(suc(k_i,r))) \\ &= v_i \leftarrow f(\delta(suc(k_i,1)),...,\delta(suc(k_i,r))) \end{aligned}$$

   Now only the validity of $\delta(suc(k_i,j)) = u_j$ for $j \in \{1,...,r\}$ is left to show. Case distinction:
   (a2) $u_j \in A$: the claim holds immediately according to the DAG construction.
   (b2) $u_j \in V$: an $l < i$ exists such that $\alpha_l = u_j \leftarrow e_l$ in π. Because of the SSA form of π an assignment to u_j is only possible in α_l. Thus $val(u_j,i) = val(u_j,l)$.

      If e_l is an operation expression by La. 5.27 a node $k_l \in K$ exists. Depending on the type of u_j we get different cases:
      (i) $u_j \notin V_{out}$: it holds

$$\delta(suc(k_i,j)) = \delta(val(u_j,i)) = \delta(val(u_j,l)) = \delta(k_l) = v_l = u_j,$$

      since similarly to case 3(b1) $V_{out}^{k_l} = \emptyset$.

(ii) $u_j \in V_{out}$: $\delta(suc(k_i, j)) = \delta(k_l) = u_j$ (because the *last*–value of u_j is automatically the lowest of all variables from $V_{out}^{k_l}$).

If $e_l \in V_{out}$ by La. 5.28 $e_l \in V_{in} \cup V_{out}$ must hold. Additional cases:

(i) $e_l \in V_{in}$: then $u_j = x$ since π is optimal w.r.t. T_{CP} and before the application of T_{CP} the SSA property was in effect. Given also that $val(u_j, i-1) = val(u_j, l) = x$ it follows $suc(k_i, j) = val(u_j, i-1) = x$ and $\forall v \in V_{in}: \ \delta(v) = v: \ \delta(suc(k_i, j)) = u_j = x$.

(ii) $e_l \in V_{out}$: then an $l' < l$ exists with $\alpha_{l'} = e_l \leftarrow g(s_1, ..., s_p)$ since due to T_{CP} an additional copy instruction $e_l \leftarrow z$ with $z \in V_{in} \cup V_{out}$ is out of question (then e_l would have been replaced by z).
Thus $val(u_j, i-1) = val(u_j, l) = val(e_l, l-1) = val(e_l, l')$ and $suc(k_i, j) = k_{l'}$. Then we have $u_j = e_l$, $\delta(k_{l'}) = e_l$ (according to the DAG code generation, step 4) and therefore $\delta(suc(k_i, j)) = u_j = e_{l'}$.

Thus overall T_{XOPT} is T_{DAG}–optimal. □

For the equivalence of the Extended Compositional Transformation and the DAG optimization we have to show the reverse too.

Main Theorem 5.30 (Equivalence of T_{DAG} and T_{XOPT}). *The two transformations T_{XOPT} and T_{DAG} are equivalent ($T_{XOPT} \sim T_{DAG}$).*

Proof. According to Def. 5.1 it remains to show:

$$T_{XOPT} \leq T_{DAG}$$

The reverse direction holds by Thm. 5.29.

In Sec. 4 we have shown that T_{DAG} is optimal w.r.t. the classical optimizations. The same holds according to [Rie05b] for the additionally introduced specialized algorithms. Thus the application of one of the algorithms composing T_{XOPT} does not modify a T_{DAG}–optimal program and neither a chaining of these algorithms will provoke any change. Hence the claim holds. □

References

[AH00] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In *CC*, pages 110–124, 2000.

[AJ76] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, 1976.

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977.

[ASU70] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. A formal approach to code optimization. *ACM SIGPLAN Notices*, 5(7):86–100, 1970.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. K Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical report, Providence, RI, USA, 1991.

[DF84] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 111–116, New York, NY, USA, 1984. ACM Press.

[DMR76] M. Davis, Y. Matijasevic, and J. Robinson. Hilbert's tenth problem. diophantine equations: Positive aspects of a negative solution. In *Mathematical developments arising from Hilbert problems*, pages 323–378. Amer. Math. Soc., Providence, 1976.

[IL80] Oscar H. Ibarra and Brian S. Leininger. The complexity of the equivalence problem for straight–line programs. In *Proc. of the 12th Annual ACM Symp. on Theory of Computing (STOC '80)*, pages 273–280, New York, NY, USA, 1980. ACM Press.

[IM83] Oscar H. Ibarra and Shlomo Moran. Probabilistic algorithms for deciding equivalence of straight–line programs. *J. ACM*, 30(1):217–228, January 1983.

[McK65] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, 1965.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer–Verlag, 1999.

[Rie05a] S. Rieger. *SLC Optimizer - Web Interface*, 2005. http://slc.srieger.com.

[Rie05b] Stefan Rieger. Analyse und optimierung linearen codes. Diplomarbeit, RWTH Aachen University, Dept. of Computer Science, Germany, August 2005. http://srieger.com/da/Diplomarbeit.pdf.

[TvSS82] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst.*, 4(1):21–36, 1982.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

1987-01 *	Fachgruppe Informatik: Jahresbericht 1986
1987-02 *	David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 *	Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 *	Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 *	Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 *	Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 *	Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 *	Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 *	Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 *	Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 *	Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12	J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 *	Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik
1988-02 *	Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 *	Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 *	Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 *	Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 *	Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 *	Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 *	Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 *	W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 *	Kai Jakobs: Towards User-Friendly Networking
1988-11 *	Kai Jakobs: The Directory - Evolution of a Standard
1988-12 *	Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 *	Martine Schümmer: RS-511, a Protocol for the Plant Floor

1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing

1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation

1988-16 * Fachgruppe Informatik: Jahresbericht 1987

1988-17 * Wolfgang Thomas: Automata on Infinite Objects

1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs

1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers

1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen

1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment

1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers

1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies

1988-24 * Uschi Heuter: Generalized Definite Tree Languages

1989-01 * Fachgruppe Informatik: Jahresbericht 1988

1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik

1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions

1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)

1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?

1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems

1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control

1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial

1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation

1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science

1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?

1989-13 * Martine Schümmer: CNC/DNC Communication with MAP

1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30 T. Margaria: First-Order theories for the verification of complex FSMs
1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991
1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes

1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems

1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line

1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme

1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management

1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines

1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking

1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD

1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars

1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)

1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik

1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual

1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems

1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages

1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)

1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine

1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus

1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions

1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code

1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine

1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)

1992-19-08 D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus

1992-19-09 D. Howe, G. Burn: Experiments with strict STG code

1992-19-10 J. Glauert: Parallel Implementation of Functional Languages Using Small Processes

1992-19-11 M. Joy, T. Axford: A Parallel Graph Reduction Machine

1992-19-12 A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)

1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer

1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine

1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell

1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation

1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages

1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)

1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment

1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)

1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers

1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)

1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)

1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting

1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)

1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models

1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures

1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)

1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language

1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language

1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing

1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free

1992-24 K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications

1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification

1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety

1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design

1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems

1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems

1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars

1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems

1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition

1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents

1994-16 P. Klein: Designing Software with Modula-3

1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems
1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming
1996-01 * Jahresbericht 1995
1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees
1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability
1996-05 Klaus Pohl: Requirements Engineering: An Overview
1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs
1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming
1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents
1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines
1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation
1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell
1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems
1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming
1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management
1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement
1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools
1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases
1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation

1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations

1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

1998-01 * Fachgruppe Informatik: Jahresbericht 1997

1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments

1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future
1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases
1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML
1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
1999-01 * Jahresbericht 1998
1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager
1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference
1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie
1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL
1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-03 D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools
2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 Mareike Schoop: Cooperative Document Management
2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 * Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08 Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 * Jahresbericht 2000
2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05 Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07 Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08 Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09 Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11 Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 * Jahresbericht 2001
2002-02 Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03 Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04 Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05 Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09 Markus Mohnen: Interfaces with Default Implementations in Java
2002-10 Martin Leucker: Logics for Mazurkiewicz traces
2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 * Jahresbericht 2002
2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises “Features”

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.