Invariance of Approximative Semantics with Respect to Program Transformations

Robert Giegerich†
Institut für Informatik
Technische Universität
Arcisstr. 21
8000 München 2

Ulrich Möncke‡
Reinhard Wilhelm
FB 14 - Informatik
Universität des Saarlandes
6600 Saarbrücken

1 Introduction

Program transformations, as used for optimization and sometimes for program development, utilize information about the program which is derived by the method of abstract interpretation within various data flow analysis frameworks. Each transformation, as it changes the program, may render the flow information derived for the original program suboptimal or even invalid with respect to the new program. Although the outdated flow information may serve well as a starting point for rederivation, it appears profitable to look for conditions where passes of many transformations may be performed without recalculating flow information each time a transformation rule is applied.

We define the notion of invariance of an approximative semantics with respect to a given set of transformation rules. In the case of invariance, the original flow information remains valid and even optimal across transformations. We demonstrate for several traditional frameworks that they do not have this property, and show how they can be revised to yield invariant frameworks.

These revised, invariant approximate semantics are sometimes more expensive to calculate. Therefore, we also show that when an approximate semantics S is invariant with respect to a set T of transformation rules, T may also be used with an approximate semantics S' without intervening rederivation, if S' is itself an approximation of S. S' need not be invariant with respect to T, and T may safely use the original information derived with S', even if it becomes invalid in the traditional sense during transformations.

†Supported by SFB 49 - Programmiertechnik -, TU München

‡Supported by DFG-Projekt "Manipulation attributierter Bäume"

2 DFA frameworks and transformations: definitions

For reasons of space, we will have to be very terse with our definitions.

2.1 Programs

Syntactic objects:

V**:** set of *values*

X**:** set of program *variables*

***EXP*:** set of (syntactic) *expressions* over X and V, written $e(x_1, \ldots, x_n)$ and not specified further

***COM*:** set of *commands*, disjoined union of

***ASS*:** set of *assignment statements* $x_0 := e(x_1, \ldots, x_n), x_i \in X$

***TESTS*:** set of *tests* $P(x_1, \ldots, x_n) \in EXP$, and

{skip} the empty command

A *program graph* is a directed graph $G = (N, E, n_{in}, n_{out})$,

N**:** set of *nodes*

$E \subseteq N \times N$**:** set of directed *edges*, written $e : n \to n'$

$n_{in}, n_{out} \in N$**:** "*entry*"- and "*exit*"-node

A program P is a pair $(G, prog)$, where G is a program graph and $prog : E \to COM$ assigns a command (assignment, test or {skip}) to each edge in the intuitively obvious way.

2.2 Program Semantics

Semantics are defined by giving some semantic domain S and an interpretation *Int* which associates a mapping $S \to S$ with each program. Correctness of program transformations will have to be defined relative to the semantics of programs. We will need two different semantics, "static semantics" and "traces semantics" for the optimizations dealt with in this paper. Common to both are the following definitions:

S**:** $\cup$-lattice of semantic objects,

Int : ***COM*** $\rightarrow (S \rightarrow S)$**:** is the (concrete) semantic interpretation of commands.

Example 1 (static semantics [CC77]): Let $S := (X \rightarrow V)$ set of states, $Z := 2^S$ powerset of states, also called "assertions". $Int_e : EXP \rightarrow (V^n \rightarrow V)$ associates a function f_e or f_p with each e or $p \in EXP$ $Int_{SS} : COM \rightarrow (Z \rightarrow Z)$ defines commands as state (set) transformers according to

$$\begin{array}{lcl} Int_{SS}(x_0 := e(x_1 \ldots, x_n)) & := & \lambda z.\{s \in S | \exists v : s[x_0 \leftarrow v] \in z \wedge \\ & & s(x_0) = f_e(\ldots, (s[x_0 \leftarrow v])(x_i), \ldots)\} \\ Int_{SS}(p(x_1, \ldots, x_n)) & := & \lambda z.\{s \in z | f_p(\ldots, s(x_i), \ldots)\} \\ Int_{SS}(\{\text{skip}\}) & := & \lambda z.z \end{array}$$

prog is extended to directed paths in G, yielding sequences of commands. Int_{SS} is extended to sequences of commands by composition. The static semantics of $(G, prog)$ is then given by $ss : N \rightarrow Z$ according to $ss(n) := \bigcup_p Int_{SS} \circ prog(p)(s_{in})$, where p is a path from n_{in} to n, and s_{in} is the set of possible entry states.

$\diamond$

2.3 Data flow analysis frameworks as approximate semantics

When discussing various DFAs, the duality between $\cup$-lattices, with "$\leq$" and $\cap$-lattices with "$\geq$" gives rise to confusion. Note that in this paper, our concrete semantics are defined via $\cup$-lattices, while approximate semantics use $\cap$-lattices, which leads to simpler intuitive arguments in each case. However, monotonicity of a mapping $m : J \rightarrow I$ from a $\cup$-lattice J to a $\cap$- lattice I must be interpreted as $\forall i, j \in J : i \leq_J j \Rightarrow m(j) \leq_I m(i)$. A data flow analysis framework (DFA) is a pair (M, L), where L is a complete $\cap$-lattice, $M \subseteq (L \rightarrow L)$ is a monoid of monotone functions.
A DFA is called an approximate semantics, if there are monotone mappings α, γ with the following properties:

1. $\alpha : S \rightarrow L$ is the *abstraction function*, $\gamma : L \rightarrow S$ is the *concretization function*, with $\gamma \circ \alpha$ *extensive*: $\forall s \in S : s \leq \gamma(\alpha(S))$;

2. There is another mapping (also denoted by α): $\alpha : (S \rightarrow S) \rightarrow M$, such that $\forall c \in COM : \forall s \in S : Int(c)(s) \leq \gamma(\alpha(Int\ c)(\alpha(s)))$

Using (M, L) instead of (Int, S) therefore may mean some loss of information, but never derive incorrect information.
A *data flow problem* DFP is a tuple $(P, Int, S, (M, L), \alpha, \gamma)$, as defined above; we will write (P, Int, α) for short, assuming S to be implied by Int, and M, L, γ to be implied by α.
An *assignment* for a DFP (P, Int, α) is a mapping $\ell_N : N \to L$. Sometimes we will omit the index.
A DFP defines a set of *data flow equations* according to

$$\ell(n) = \bigcap_{e:n' \to n} \alpha \circ Int \circ prog(e)(\ell(n'))$$

This system corresponds to a so-called "forward"-DFP. For a "backward"-DFP, just think of all the edges inverted.
An assignment ℓ_N is a *fixpoint*, if it is a solution to the above system of equations (there is always a unique maximal fixpoint $\hat{\ell}_N$).
An assignment ℓ_N is conservative, if $\forall n : \ell_N(n) \leq \hat{\ell}_N(n)$.

Example 2 (constant propagation):
The DFP constant propagation CONST is defined by choosing:

$$\begin{array}{ll}
Int = Int_{SS} & \text{static semantics} \\
S = X \to V, Z = 2^S & \\
L = 2^{X \times V} & \\
\alpha : Z \to L, \;\; \alpha(z) = \{(x, v) | \forall s \in z,\; s(x) = v\} & \\
\gamma : L \to S, \;\; \gamma(\ell) = \{s \in [X \to V] | (x, v) \in \ell \Rightarrow s(x) = v\} & \\
\alpha(Int(x_0 := e(x_1, \ldots, x_n))) = \lambda \ell.\; \ell \backslash KILL(x_0 := e(x_1, \ldots, x_k))(\ell) & \\
\qquad \cup \text{ if } (x_1, v_1) \ldots (x_k, v_k) \in \ell & \\
\qquad\quad \text{then } \{(x_0, (Int_e(e))(v_1, \ldots, v_k)\} & \\
\qquad\quad \text{else } \emptyset & \\
\qquad\quad \text{fi} & \\
KILL(x_0 := e(x_1, \ldots, x_k)) = \lambda \ell. \{(x_0, y) \in \ell\} & \\
\alpha(Int(P(x_1, \ldots, x_k))) = \lambda \ell. \ell & \\
\alpha(Int(\{\text{skip}\})) = \lambda \ell. \ell &
\end{array}$$

$\diamond$

2.4 Program Transformations

A (program) transformation rule r based on an approximate semantics $(Int, S, (M, L), \alpha, \gamma)$ is a triplet $(c, c', B) \subseteq COM \times COM \times (L \to BOOL)$, where B is the "applicability condition" and satisfies

$$B(\ell) \Rightarrow \forall s \in \gamma(\ell) : Int(c)(s) \doteq Int(c')(s),$$

for some appropriately defined relation $\doteq$ of semantic equivalence. Thus, $B(\ell)$ implies preservation of program correctness when a program $P = (G, prog)$ is transformed by r, which is defined as follows:
Let ℓ_N be a conservative assignment, $e : n \to n'$ for forward, $e : n' \to n$ for backward problems, $prog(e) = c$ and $B(\ell(n)) =$ true.
The transformed program $P' = (G, prog')$ is defined by $prog' := prog[e \leftarrow c']$.
Of course, the modified program P' gives rise to a modified DFP with a set of equations differing from the original one with respect to $\alpha \circ Int \circ prog(e)$.
As an example, the rule (actually a rule schema) for constant folding is

$$\begin{aligned} rc : (\ & x := e(x_1, \ldots, x_k), x := v, \lambda\ell.(x_1, v_1) \in \ell \wedge \ldots \wedge (x_k, v_k) \in \ell \wedge \\ & (Int(e))(v_1, \ldots, v_k) = v) \end{aligned}$$

3 Safeness and invariance of frameworks with respect to transformations

3.1 Enabling and affected frameworks

As transformations modify programs and therefore the associated data flow equations, they may outdate the flow information calculated for the original program. In general, there are two frameworks to be considered: The "enabling" framework, which the transformation is based on, and the "affected" framework - the one one is interested in for the next transformation step, whose flow information has also been calculated already. When we ask if a transformation modifies its "own" flow information, enabling and affected framework are the same. In this case, or in the case where both frameworks use the same lattice L of abstract information, we will be able to make stronger assertions about possible effects than in the case of totally unrelated frameworks.
For the rest of this section, let

$A = (P, \boldsymbol{Int}_A, S_A, (M_A, L_A), \alpha_A, \gamma_A)$ be the enabling,

$O = (P, \boldsymbol{Int}_O, S_O, (M_O, L_O), \alpha_O, \gamma_O)$ be the affected DFP, and let

$r = (c, c', B)$ be a transformation rule based on A, i.e. $B \in L_A \to BOOL$.

The transformed DFPs are referred to as A' and O'.

3.2 Safeness

The least we can ask for is that the information calculated for the affected framework remains valid (but not necessarily optimal). This means, a conservative assignment for O must also be one for O'.

Definition 3: Let $r = (c, c', B : L_A \to BOOL)$ as before, and B monotone, i.e. $\ell \leq \ell' \Rightarrow (B(\ell) \Rightarrow B(\ell'))$. O is safe with respect to r , if

1. for $L_A = L_O$, $\gamma_A = \gamma_O$: $\forall \ell \in L_A : B(\ell) \Rightarrow ((\alpha_O \circ Int_O)(c)(\ell) \leq (\alpha_O \circ Int_O)(c')(\ell))$
2. for $L_A \neq L_O$ or $\gamma_A \neq \gamma_O$: $\forall \ell \in L_A : B(\ell) \Rightarrow ((\alpha_O \circ Int_O)(c) \leq (\alpha_O \circ Int_O)(c'))$

□

Theorem 4: If ℓ_A is a conservative assignment to A, and if ℓ_N is a conservative assignment to O, and O is safe with respect to r, then ℓ'_N is a conservative assignment for O'.

Proof: Obviously, the system of equations for O' differs from the one for O only with respect to the edge $e : m \to n$, where r was applied. Let $\hat{\ell}_N$ be the maximal fixpoint of O, $\hat{\ell}'_N$ that of O'. We have to show that $\ell_N \leq \hat{\ell}_N \leq \hat{\ell}'_N$ is implied by $B(\ell_A(m))$. In case 2., $\alpha_O \circ Int_O(c) \leq \alpha_O \circ Int_O(c')$ implies $\hat{\ell}_N \leq \hat{\ell}'_N$. In case 1., the monotonicity of B implies that $\alpha_O \circ Int_O(c)(\ell) \leq \alpha_O \circ Int_O(c')(\ell)$ for all $\ell \geq \hat{\ell}_N(m)$, ℓ in the Kleene sequence calculating $\hat{\ell}'_N$. Therefore we also have $\hat{\ell}_N \leq \hat{\ell}'_N$ in this case.

□

Example 5 (DEADVARS, COPIES): Let DEADVARS be the framework that determines at each node of a program graph the set of variables whose current value will not be used later on, either since no use is made of the variable until the end of the program, or since any use of the variable is preceded by a modification of the variable.

Let COPIES be the framework that determines at each node the following relation *copy* between variables: x *copy* y iff x and y hold the same value, every time execution of the program reaches this node, cf. [Hec77], [AU77]. Both frameworks can be used to eliminate "useless" assignments. Let r_e be the elimination of useless assignments, r_i the introduction of an assignment to a dead variable.

1. DEADVARS is not safe with respect to r_e and COPIES,

2. COPIES is not safe with respect to r_e and DEADVARS.
   transformed using COPIES original program transformed using DEADVARS

$$
\begin{array}{ccccccccc}
\text{COPIES} & & \text{DEADVARS} & & \text{COPIES} & & \text{DEADVARS} & & \text{COPIES} & & \text{DEADVARS} \\
\emptyset & & \{x\} & & \emptyset & & \{x\} & & \emptyset & & \{x\} \\
& \Big\downarrow \text{x:=y} & & & & \Big\downarrow \text{x:=y} & & & & \Big\downarrow \{\text{skip}\} & \\
\{(x,y)\} & & \{x\}^{*} & & \{(x,y)\} & & \{x\} & & \{(x,y)\}^{*} & & \{x\} \\
& \Big\downarrow \{\text{skip}\} & & \xleftarrow[(x,y)\,\in\text{COPIES}]{r_e} & & \Big\downarrow \text{x:=y} & & \xrightarrow[x\,\in\text{DEADVARS}]{r_e} & & \Big\downarrow \text{x:=y} & \\
& & \emptyset & & & & \emptyset & & & & \emptyset
\end{array}
$$

3. DEADVARS is safe with respect to r_e and DEADVARS. Clear, since $(\alpha \circ \mathit{Int})(\{\text{skip}\})$ eliminates no dead variables, and $(\alpha \circ \mathit{Int})(x_0 := e(x_1, \ldots, x_n))$ introduces no *new* dead variables when x_0 is dead.
4. DEADVARS is not safe with respect to r_i and DEADVARS.

$$
\begin{array}{ccccc}
\{x,y\} & & & \{x,y\}^{*} & \\
& \Big\downarrow \{\text{skip}\} & \xrightarrow[x\,\in\text{DEADVARS}]{r_i} & & \Big\downarrow \text{x:=y} \\
\{x,y\} & & & \{x,y\} &
\end{array}
$$

$\diamond$

3.3 Invariance

If we require that maximality of fixpoints be retained, or if we require safeness for a symmetric (with respect to c and c') set of transformation rules, we arrive at the notion of invariance:

Definition 6: Let $r = (c, c', B)$ as before, B monotone. O is *invariant* with respect to r, if

1. for $L_A = L_O, \gamma_A = \gamma_O$: $\forall \ell \in L_A : B(\ell) \Rightarrow (\alpha \circ \mathit{Int})(c)(\ell) = (\alpha \circ \mathit{Int})(c')(\ell)$
2. for $L_A \neq L_O$ or $\gamma_A \neq \gamma_O$: $\forall \ell \in L_A : B(\ell) \Rightarrow (\alpha \circ \mathit{Int})(c) = (\alpha \circ \mathit{Int})(c')$.

*Invalid Information!

□

Theorem 7: If O is invariant with respect to r and A, and $\hat{\ell}_N$ is the maximal fixpoint of O, then $\hat{\ell}_N$ is the maximal fixpoint of O'.

Proof. In case 2., the system of equations of O and O' and hence their maximal fixpoints are identical. In case 1., the monotonicity of B implies that the Kleene sequences calculating the maximal fixpoints, and hence the fixpoints themselves are identical, q.e.d.

□

Example 8 (DEADVARS, CONST): DEADVARS, as used before, is apparently not invariant with respect to r_e and itself. CONST [Hec77] is invariant with respect to constant folding (r_c) and itself. DEADVARS (CONST) is not invariant with respect to r_c and CONST (r_e and DEADVARS). DEADVARS is safe with respect to r_e and CONST is not safe with respect to r_e and DEADVARS.

◇

From these examples, it becomes clear how the properties of invariance and safeness help the compiler designer with the structuring of optimization phases:
Invariance allows a phase of many transformations, all based on optimal information. Safeness allows to proceed with transformations before rederiving flow information, at the price of missing some chances this time. Otherwise, invalidated information must be corrected before optimization can go on, cf. [Wil79].
Apparently, it is highly desirable that a DFA be invariant with respect to itself and the transformations it is used for. In the next section we show how to turn DEADVARS into such a framework.

4 Dead variables, faint variables

4.1 Classical dead variables

It was shown in section 3 that the DEADVARS-framework is not invariant under the elimination (and insertion) of assignments to dead variables. Now, we construct a framework that is stronger than DEADVARS and invariant under the elimination (and insertion) of useless assignments. (Insertion is an important rule on the machine code level of optimization [Gie81]). Let us first give an exact description of DEADVARS.

DEADVARS is an abstraction of a traces semantics [CC79]. We will only give the $\alpha \circ \mathit{Int}$ definition of the functions of the framework.

$$(\alpha \circ \mathit{Int})(c)(a) = (a \cup \mathit{KILL}(c)) \setminus \mathit{USE}(c),$$

where $\mathit{KILL}(c)$ is the set of variables, whose value may be changed by the execution of command c, $\mathit{USE}(c)$ is the set of variables, whose value may be used when command c is executed.

DEADVARS is a backwards, meet framework. The system of equations for a given program P is usually written

$$\mathit{DEADVARS}(m) = \bigcap_{e:m \to n} ((\mathit{DEADVARS}(n) \cup \mathit{KILL}(\mathit{prog}(e)) \setminus \mathit{USE}(\mathit{prog}(e))$$

4.2 Construction of an invariant framework for dead variables

Let the framework we look for be (M_a, L_a). The regarded transformation is

$$(c, c', B : L_a \to \mathit{BOOL})$$

where $c \equiv x_0 := e(x_1, \ldots, x_n)$
$c' \equiv \{\text{skip}\}$
$B \equiv x_0 \in \ell$

The condition for invariance according to Def. 3 (1) is:

$$\forall \ell \in L_a : B(\ell) \Rightarrow ((\alpha_a \circ \mathit{Int}_a)(c)(\ell) = ((\alpha_a \circ \mathit{Int}_a)(c')(\ell))$$

Here, this reads:

$$\forall \ell \in L_a : x_o \in \ell \Rightarrow ((\ell \cup \{x_0\}) \setminus \{x_1, \ldots, x_n\} = \ell)$$

The new framework, let us call it FAINTVARS, anticipates the elimination of assignments for faint variables by the following definition of the abstract semantics of an assignment statement:

$$\alpha \circ \mathit{Int}(x_0 := e(x_1, \ldots, x_n))(\ell) = \text{ if } x_0 \in \ell \text{ then } \ell \text{ else } (\ell \cup \{x\}) \setminus \{x_1, \ldots, x_n\}$$

The system of equations can then be written as

$$\begin{aligned}\mathit{FAINTVARS}(m) = \bigcap_{e:m \to n} \; & \text{if } \mathit{KILL}(\mathit{prog}(e)) \subseteq \mathit{FAINTVARS}(n) \text{ then} \\ & \mathit{FAINTVARS}(n) \text{ else} \\ & (\mathit{FAINTVARS}(n) \cup \mathit{KILL}(\mathit{prog}(e)) \setminus \mathit{USE}(\mathit{prog}(e))\end{aligned}$$

FAINTVARS has the following nice properties:

- it is invariant under elimination and insertion of assignments to faint variables (can be seen from the construction),
- it is stronger than DEADVARS, that is, any dead variable is also faint (can be seen by comparing the two systems of equations),
- it is exhaustive in the sense that, if a maximal fixpoint for FAINTVARS is computed, and all assignments to variables found to be faint are eliminated, then a new computation of a fixpoint of FAINTVARS would reveal no faint variables having assignments (follows from the invariance),
- it may be more expensive to compute than DEADVARS but is still cheap (for every assignment we have an additional membership test),
- it ensures the preservation of semantic equivalence by the transformation regarded, since it is an implementation of an axiom of the traces semantics: When an assignment $x_0 := e(x_1, \ldots, x_n)$ is redundant, then the values of $x_1, \ldots, x_n$ are irrelevant at this point; therefore, the redundant assignment need not be considered as using the values of $x_1, \ldots, x_n$.

5 Working with approximations of invariant frameworks

In this section, we show by an example, which is very easily generalized, that one can benefit from the mere existence of invariant frameworks, while actually using "cheaper" approximations.

DEADVARS is itself an approximation of FAINTVARS. Assume DEADVARS is the one actually used for optimization. By application of r_e and r_i, DEADVARS-information may become invalid. But it must still be valid with respect to FAINTVARS, which we know to be invariant and hence safe wrt both r_e and r_i, and itself. Thus, in the "higher" sense, the outdated DEADVARS-Information is still valid and may safely be used for further transformations. Note that this would not be the case if it had been outdated by a rule with respect to which FAINTVARS is not safe, say r_e based on CONST.

6 Conclusion

An optimizing compiler performs a number of code improving program transformations. Most of these transformations are based on global data flow information. In this paper the properties of safeness and invariance of data flow analysis frameworks under transformations are defined. Several traditional DFA-frameworks are shown to be unsafe, or not invariant under some transformations: For the well known framework DEADVARS a stronger and invariant framework has been constructed. It is argued that the notions of safeness and invariance are valuable aids in devising the optimization phase of a compiler.

References

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison Wesley, 1977.

[CC77] Patrik Cousot and Radhia Cousot. A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In: *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[Gie81] Robert Giegerich. *Automatische Erzeugung von Maschinencode-Optimierern.* PhD thesis, TU München, 1981.

[Hec77] M.S. Hecht. *Flow analysis of computer programs.* North Holland, 1977.

[Wil79] Reinhard Wilhelm. Computation and use of data flow information in optimizing compilers. *Acta Informatica*, 12, 1979.