A Flow-Insensitive-Complete Program Representation

Solène Mirliaz, David Pichardie

A Flow-Insensitive-Complete Program Representation

Solène Mirliaz[1] and David Pichardie[2]

[1] Univ Rennes, Inria, CNRS, IRISA
[2] Facebook

Abstract. When designing a static analysis, choosing between a flow-insensitive or a flow-sensitive analysis often amounts to favor scalability over precision. It is well known than specific program representations can help to reconcile the two objectives at the same time. For example the SSA representation is used in modern compilers to perform a constant propagation analysis flow-insensitively without any loss of precision.
This paper proposes a provably correct program transformation that reconciles them for any analysis. We formalize the notion of Flow-Insensitive-Completeness with two collecting semantics and provide a program transformation that permits to analyze a program in a flow insensitive manner without sacrificing the precision we could obtain with a flow sensitive approach.

1 Introduction

Static analysis designers must face two main challenges. The first one is scalability because the analysis should compute a sound approximation within a reasonable amount of time. The second one is precision because the approximation should be accurate enough to prove the target properties on as many programs as possible.

Abstract interpretation provides a rich methodology to guide the static analysis design but precision and scalability are often difficult to optimize at the same time. At one side of the spectrum stand relational abstract interpreters [4, 13, 12] that compute expressive symbolic relations on program variables at each program point (flow sensitivity). At an other side of the spectrum, flow-insensitive analyses [15] (such as Andersen's pointer analysis [2]) compute one global invariant for the whole program, sparing time and memory.

Flow sensitivity allows to compute local invariants at each program point, without polluting the inferred properties with too many infeasible paths. But this technique generally requires to remember an invariant at several program points of the program. This may have bad impact on performance, in particular memory usage.

On very specific programs, flow-insensitive and flow-sensitive analyses have the same precision. Figure 1 shows two examples. On the left, the global invariant $\mathtt{x} = \mathtt{y} = 0$ is invalid after the last assignment $\mathtt{x} := 1$. However, after a simple renaming we obtain the program on the right where $\mathtt{x0} = \mathtt{y0} = 0$ is a valid global

```
x := 0;                    x0 := 0;
y := 0;                    y0 := 0;
x := 1;                    x1 := 1;
```

Fig. 1: Comparing flow insensitive analysis precision losses on two programs.

invariant. This renaming is a very simple case of *Static Single Assignment* transformation (SSA) [5] where each variable is given a unique definition point. The SSA intermediate representation is very popular in compiler frameworks because many flow sensitive program optimizations can be performed with a flow insensitive approach on a SSA representation without loss of precision. This has been observed for constant propagation analysis [10] in an analysis named *Sparse conditional constant propagation* [18].

But SSA transformation is not always enough. For example, a popular compiler optimization, *Global Value Numbering* [1] is performed flow insensitively on SSA form in order to detect equivalence between program sub-expressions and perform common sub-expressions elimination. But Gulwani and Necula show [6] it is not precise enough and provide a provably more precise flow sensitive alternative version.

An alternative program representation to SSA, the *Single Single Information* (SSI) form [17], extends the SSA form with extra-properties. In [17], Pereira and Rastello consider *non-relational* analyses which bind information to i) each program variable, and ii) each program point where the variable is live. They design the SSI form in order to ensure that each variable will respect the same invariant at any point where it is alive. Their work shows that for non-relational analyses, the SSI transformation allows to compute, with a flow insensitive analysis of the SSI program, the same amount of information than with a standard flow sensitive analysis of the original program. But they also conclude with the remark that this property does not hold for *relational* analyses that compute relations between program variables. Part of this limitation is removed with [14] for what is called semi-relational analyses.

This paper is the first to explore the problem without restrictions on the relational nature of the analysis. We take a *semantic* approach and do not bind our work to a specific numerical analysis or abstract domain. We make the following contributions:

- We propose a new program transformation technique that inserts enough move instructions (called σ copies in the SSI vocabulary and simply copies in this paper) to turn a SSA program into an equivalent *Flow Insensitive Complete* (FIC) program. The obtained program can be analyzed with a flow insensitive approach without loss of precision compare to a flow sensitive manner.
- We formalize the notion of *Flow Insensitive Completeness* with two collecting semantics. The flow-sensitive collecting semantics characterizes the set of reachable states in term of program paths while the flow-insensitive collecting

semantics characterizes another set of states with respect to any permutation of blocks of instructions.
- We prove that the two collecting semantics detect the same set of assert failures for all Flow Insensitive Complete programs.
- We implement the transformation for Java bytecode in SSA form and observe that the total number of variables remains reasonable compared to the size of flow-sensitive analyses invariants.

2 Motivating example

We present in Figure 2 an example that explains why the SSI form does not introduce enough variables to allow relational reasoning, and how our approach handles the problem.

Figure 2 contains both the source program and its SSA form in a graph representation. We iterate the loop 10 times (using the loop counter i). Since j is initialized at 0, and is incremented by one or two at each iteration, it is expected to be in the range $[10, 20]$ at the end of the loop.

Note that, in our SSA representation, ϕ instructions are performed before each junction points, rather than at the entrance. This inoffensive convention makes our proof easier to expose.

We present in Figure 3a a SSI form of this program. According to the standard SSI transformation, copy instructions (σ copies of the form $x \overset{\sigma}{\leftarrow} y$) have been added to all branching points, for all the variables used in the corresponding branching test (i_1 for the loop test and x_0 for the conditional test), and to blocks containing assumes, for all the variables used in them (j_1 in b_5).

As expected, on this SSI program, a non-relational flow insensitive analysis like an interval analysis will be as precise as a flow sensitive version. But such a non-relational analysis will conclude that $i_4 = 10$ and $j_1 \in [0, +\infty]$ and it will fail to verify the assertion because it fails to discover the relational invariant between i and j.

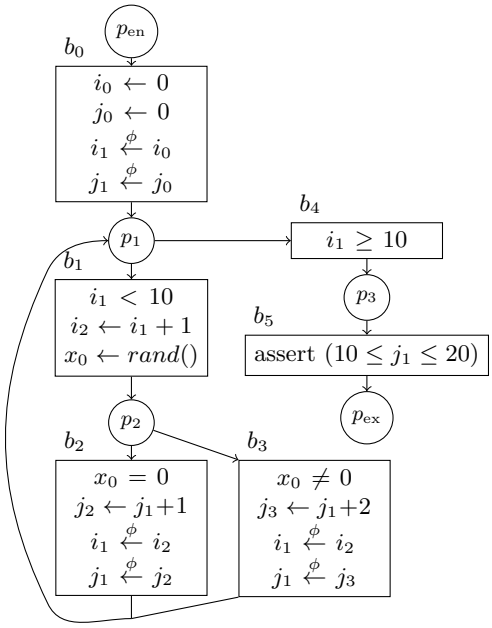
A relational abstract domain, like the polyhedral one will not solve the precision problem either, if it is performed in a flow insensitive style. Indeed the global polyhedral fixpoint should be closed by operations $[\![i_1 \leftarrow 0 | j_1 \leftarrow 0]\!]$ (parallel assignment of i_1 and j_1) and $[\![j_4 \overset{\sigma}{\leftarrow} j_1]\!]$ so assertion at block b_5 will raise an alarm because j_4 seems to be nullable.

The current paper proposes a FIC form displayed in Figure 3b to fix this imprecision. It is build from the SSA form, by adding copies in strategic blocks. In this new form, the assertion block b_5 now uses j_4, not j_1, so it can only be applied on a state where j_4 has been defined by b_4. This time, the previous problem does not hold because the global polyhedral fixpoint should be closed by the operation $[\![j_4 \leftarrow j_1]\!] \circ [\![i_1 \geq 10]\!]$ which *prevents* the case $i_1 = 0; j_1 = 0$ to be spuriously propagated into j_4.

Notice that we do not introduce copies for i_1 in b_4 before the assume, unlike the SSI form. The FIC form only ensures completeness w.r.t the assertions, not to any point of the program. Such consideration avoid the insertion of copies

```
i = 0;
j = 0;
while (i < 10):
  i += 1;
  x = rand();
  if (x = 0):
    j += 1
  else:
    j += 2
(...)
assert(10 <= j <= 20)
```

(a) Source program



(b) Program in SSA form

Fig. 2: A program and its SSA form with relational information to infer

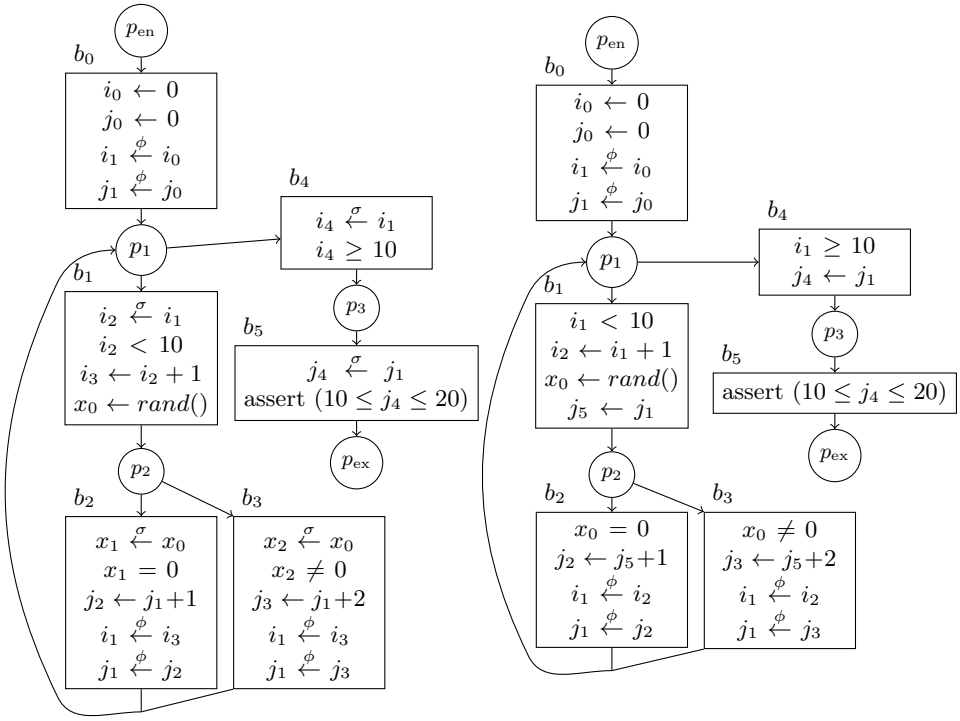
for every original variable (i, j and k) at each block. The number of variables would be overwhelming for most abstract domains and one will lose the benefits of flow-insensitivity on memory saving. Generally speaking, if the number of variables in the FIC form is greater or equal to the number of blocks times the number of variables in the original program, then a flow-insensitive analysis on the FIC form is not an improvement compared to a flow-sensitive analysis on the source program.

3 Background definitions

This section introduces the definition of programs used in this paper. The section ends with the definition of both the flow-sensitive and the flow-insensitive semantics.

3.1 Program

A program P is defined as a graph connecting program points, and whose edges are labeled with basic blocks. The program as a unique entry point p_{en} and a unique exit point p_{ex}. A basic block b is a tuple $\langle body, c, \phi \rangle$. The body is composed



(a) Program in SSI form

(b) Program in flow-insensitive complete form

Fig. 3: Comparison of the SSI form and the FIC form of the program from Figure 2

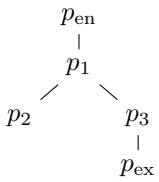
of a sequence of atomic instructions which can be assignments, assumes or assertions. The second element, c, is a set of (parallel) copies e.g. $[\![x_1 \leftarrow x_0 | y_1 \leftarrow y_0]\!]$ assigns x_1 and y_1 in parallel. Similarly the last element ϕ is a set of (parallel) ϕ-definitions e.g. $[\![x_1 \overset{\phi}{\leftarrow} x_0 |\ y_1 \overset{\phi}{\leftarrow} y_0]\!]$. A more precise definition of their semantics is developed in Section 3.3. A basic block labels an edge between two program points and thus entry(b) and exit(b) respectively define the unique program points from and to which the edge goes. For instance in Figure 2, b_1 and b_4 have the same entry point p_1 and b_0, b_2 and b_3 share the same exit point p_1. All edges should be labeled with a non-empty block. We note $p \overset{b}{\rightarrow} p'$ the fact that block b labels an edge from p to p'.

For each program point p we define its set of predecessors blocks pred(p) such that $b \in \text{pred}(p) \iff \text{exit}(b) = p$.

Definition 1 (Program point path). *A path from program point p to program point p' is a sequence of program points $p, p_1, \dots p_n, p'$ such that $p \xrightarrow{b_0} p_1 \xrightarrow{b_1} \dots \xrightarrow{b_{n-1}} p_n \xrightarrow{b_n} p'$.*

Definition 2 (Dominance). *p dominates p' if all paths from p_{en} to p' must go through p.*

The dominance is *strict* if $p \neq p'$. The dominance relation is transitive, and it is possible to organize all points in a dominance tree where the parents of a node dominate it. For instance the dominance tree of Figure 2 is



The *direct dominator* of a program point is its parent in the dominance tree. We extend the notion of dominance to blocks.

Definition 3 (Block dominance). *A program point p dominates a block b iff it dominates its entry point.*

3.2 Static information

Let $\mathbb{V}$ be the set of variables in the program p. We can define for each block the set of variables it uses and defines: uses(b) and defs(b). These sets do not include temporary variables, meaning that the set uses(b) does not include variables that are defined before their usage in block b, and the set defs(b) does not include variables that are not used outside of b. For instance in Figure 3b, the initial block b_0 is not considered to be defining nor using i_0 and j_0 because these variables are defined by this block but never used outside of it. The initial block defines i_1 and j_1 and uses no variables. The block b_2 uses x_0, j_5 and i_2 and defines i_1 and j_1.

Unlike textbook SSA form, our ϕ-definitions are parts of the predecessors of the junction point. Because of this convention, a variable is not necessarily defined in a unique block. Despite the unconventional choice, our notion of program still enjoys the foundational property of definition dominance of SSA programs.

Invariant 1 (SSA dominance) *Let x be a variable, and B be the set of blocks defining them. Then the set of exit points of B is a singleton $\{p\}$ and p dominates all blocks b' such that $x \in \text{uses}(b')$.*

In textbook SSA form, all variables have a unique definition points. In our representation, we split the ϕ-function $x_3 \xleftarrow{\phi} (x_1 : b_1, x_2 : b_2)$ attached to a junction point p, so that there is a ϕ-definition $x_3 \xleftarrow{\phi} x_1$ in block b_1 and $x_3 \xleftarrow{\phi} x_2$ in b_2.

This definition is in the ϕ component of the block. All other definitions from the textbook SSA form are found in the *body* component of the blocks. So, in the property, B is not a singleton iff x is defined by ϕ-definitions.

Definition 4 (Program points definitions). *The definitions of a program point p is the set of variables defined by all its predecessors:*

$$\text{defs}(p) = \bigcap_{\forall b \in \text{pred}(p)} \text{defs}(b)$$

3.3 Block local semantics

States We note $s \in \mathbb{S} = \mathbb{V} \rightharpoonup \mathbb{Z}$ a state of the variables. It is a partial function from the variables to values and its domain $\text{dom}(s) \subseteq \mathbb{V}$ is the set of variables for which it is defined. Partial functions are useful in a flow-insensitive analysis to account for the variables never assigned. The initial state $s_\emptyset$ has an empty domain reflecting the fact that no variable is initially assigned.

Definition 5 (State equivalence). *Two states s and s' are said* equivalent *on a set of variables V, noted $s \approx_V s'$, iff they both include this set in their domains and if they are equal on these variables.*

$$s \approx_V s' \iff (V \subseteq \text{dom}(s) \wedge V \subseteq \text{dom}(s') \wedge \forall v \in V, s(v) = s'(v))$$

The symbol Ω denotes an halting state obtained when an assert failed. As a convention its domain is empty. $\mathbb{S}^\Omega = \mathbb{S} \cup \{\Omega\}$ denotes the complete set of states.

Semantics We use the notation $[\![a]\!] : \mathbb{S} \to \mathcal{P}\left(\mathbb{S}^\Omega\right)$ for the concrete semantics of a list of instructions a. The output is a set of states since our semantics is non-deterministic (for instance with the call to `rand()`). We extend the semantics to any set of states $S \subseteq \mathbb{S}^\Omega$, $[\![a]\!](S) = \bigcup_{s \in S} [\![a]\!](s)$ with $[\![a]\!](\Omega) = \emptyset$.

The semantics of a block is the composition of its parts: $[\![\phi]\!] \circ [\![c]\!] \circ [\![body]\!]$.

We only consider programs which manipulate variables, not memory. Assumes are supposed to block the execution for states not satisfying its condition, while an assertion will result in a halting state Ω.

$$[\![\texttt{assume}(false)]\!] = \emptyset \qquad [\![\texttt{assert}(false)]\!] = \Omega$$

The exact definition of the semantics of blocks $[\![b]\!]$ is not important for the proofs as long as it respects the following two characterization.

Invariant 2 (Semantic characterization of uses) *The semantics of a block only depends on the variables it uses.*

$$\forall b, \forall s_1, s_2, s_1' \in \mathbb{S}^\Omega, \left(s_1 \approx_{\text{uses}(b)} s_2 \wedge s_1' \in [\![b]\!](s_1)\right) \implies \exists s_2' \in [\![b]\!](s_2), s_1' \approx_{\text{defs}(b)} s_2'$$

With the special case for Ω:

$$\forall b, \forall s_1, s_2 \in \mathbb{S}^\Omega, s_1 \approx_{\text{uses}(b)} s_2 \implies (\Omega \in [\![b]\!](s_1) \iff \Omega \in [\![b]\!](s_2))$$

The non-determinism prevents the conclusion that any state out of $[\![b]\!](s_2)$ is equivalent to s'_1.

Invariant 3 (Semantic characterization of definitions) *The semantics of a block only modifies the variables it defines.*

$$\forall b, \forall s \in \mathbb{S}^{\Omega}, \forall s' \in [\![b]\!](s), s' \approx_{\mathbb{V}\setminus \mathrm{defs}(b)} s$$

This two characterization consider that temporary variables of a block b are not in the domain of a state $s' \in [\![b]\!](s)$ (for any s). They can be ignored or remove from the domain of s'.

3.4 Flow-sensitive collecting semantics

The flow-sensitive collecting semantics of a program associates to each program point a set of reachable states $\textsc{Local}(p)$. The function is defined as the least fixpoint of the following equations.

$$\forall p, \textsc{Local}(p) = \begin{cases} \{s_\emptyset\} & \text{if } p = p_{\text{en}} \\ \bigcup\limits_{p' \xrightarrow{b} p} [\![b]\!] \circ \textsc{Local}(p') & \text{otherwise} \end{cases}$$

Lemma 1. *For all program points p,*

$$\textsc{Local}(p) = \bigcup_{p_{en} \xrightarrow{b_1} \dots \xrightarrow{b_n} p \ a\ path} [\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset)$$

The proof of this lemma is classical for least fixpoints and is available in Appendix A.

3.5 Flow-insensitive collecting semantics

For the flow-insensitive collecting semantics, the information is not associated to program points but to the whole program. States are collected from anywhere in the program : $\textsc{Global} \in \mathcal{P}\left(\mathbb{S}^{\Omega}\right)$. The flow-insensitive semantics is the least fixpoint satisfying the following equation.

$$\textsc{Global} = \{s_\emptyset\} \cup \bigcup_b [\![b]\!](\textsc{Global})$$

In other word it is the smallest set of states containing the initial state $s_\emptyset$ closed by $[\![b]\!]$ for any block b.

$$s_\emptyset \in \textsc{Global} \qquad \text{and} \qquad \forall b, [\![b]\!](\textsc{Global}) \subseteq \textsc{Global}$$

In this settings, a block b can be applied to any state, any partial function. In case the state s does not have a domain containing all variables used by b, then the semantics of the block is an empty set: $\mathrm{uses}(b) \not\subseteq \mathrm{dom}(s) \implies [\![b]\!](s) = \emptyset$.

Lemma 2. *Any elements of* Global *is actually the result of the application of a sequence of blocks on the state $s_\emptyset$. There is no restriction on the order of these blocks.*

$$\textsc{Global} = \bigcup_{(b_1,\ldots,b_n)} [\![b_n]\!] \circ \cdots \circ [\![b_1]\!](s_\emptyset)$$

Thanks to this lemma, it is easy to see that the flow-insensitive semantics always contains each local invariants.

Corollary 1. *For all program points p,*

$$\textsc{Local}(p) \subseteq \textsc{Global}$$

4 Flow-Insensitive Complete (FIC) programs

The example in Section 2 illustrates the need of a different representation of program to ensures the equivalence to a flow-insensitive semantics. This section presents the intrinsic properties expected of the FIC representation. We rely on these properties to ensure the main theorem of precision in Section 5. Section 6.2 presents a transformation from an SSA program to a program in FIC form.

Incoherence from disjoint definition points in SSA form A first issue of the SSA form to establish flow-insensitive invariants on variables is the potentially different definition points of the variables used by a block. The flow-insensitive semantics can collect states where it applies these definitions in any order, and any number of times. In Figure 3a for instance, in block b_3, both the variables i_3 and j_1 are used but they are defined in different blocks. i_3 is defined in b_1 and j_1 in b_0, b_2 and b_3. Let us consider a state $s \in [\![b_3]\!] \circ [\![b_3]\!] \circ [\![b_1]\!] \circ [\![b_0]\!](s_\emptyset) \subseteq \textsc{Global}$. This state has the following evaluations:

$$s(i_1) = 0 \quad s(i_3) = 1 \quad s(j_3) = 4 = s(j_1) \quad \text{(since we applied } b_3 \text{ twice)}$$

But with this state we already lost the invariant linking i_3 and j_3 at the end of $b_3 : i_3 \leq j_3 \leq 2 \times i_3$. To prevent this, if variables are used in a block b', then any block b defining some of them must actually redefine all of them, to ensure coherence.

Intrinsic FIC property 1 (Comprehensive definition coverage) *For any blocks b and b', if b defines some variables used by b', then it defines all variables used by b'.* $\text{defs}(b) \cap \text{uses}(b') \neq \emptyset \implies \text{uses}(b') \subseteq \text{defs}(b)$

On the example in FIC form, the version j_5 introduced in b_1 ensures the coherence between i and j.

Invisible path from definition to use In the introduction we observed that the assertion was violated because we could apply the block b_0 first, defining j_1, and then the block b_5, which uses j_1 for the assertion, without taking into account the assume in block b_4. This block b_4 dominates b_5 and restricts its reachable states. To account for this control, a new version of j is introduced in b_4. This new version will be defined only in states where the condition $i_1 \geq 10$ holds.

The minimal property we expect is that for any state reaching the exit of a definition block b', there exists a path from $\text{exit}(b') = p$ to $\text{entry}(b)$ which is *non-altering* for the variables used by b.

Definition 6 (Non-altering path). *Let p be a program point, b a block, and $s \in \textsc{Local}(p)$, a* non-altering path *for s from p to b is a path $p \xrightarrow{b_1} \ldots \xrightarrow{b_n} \text{entry}(b)$ such that*

$$\exists s' \in [\![b_n]\!] \circ \cdots \circ [\![b_1]\!](s), s' \approx_{\text{uses}(b)} s$$

Intrinsic FIC property 2 (Non-altering def-use path) *If $\forall p, \Omega \notin \textsc{Local}(p)$ then for any block b and any program point p,*

$$\text{defs}(p) \cap \text{uses}(b) \neq \emptyset \implies \begin{pmatrix} \forall s \in \textsc{Local}(p), \\ \exists \text{ a non-altering path from } p \text{ to } b \text{ for } s \end{pmatrix}$$

We also add a special case for any block which uses no variable. In that case we only require the existence of some state s' reaching the block.

$$\text{uses}(b) = \emptyset \implies \exists s' \in \textsc{Local}(\text{entry}(b))$$

As it is a strong property on the semantics, we define in the Section 6.1 syntactical conditions to ensure this property. However we use this property in the proof of our central Theorem 2, in order to be as general as possible on the shape of the program graph.

Definition 7 (FIC form). *A Flow-Insensitive Complete program is a SSA program that respects properties 1 and 2.*

5 Main Theorem: Flow Insensitive Completeness

The completeness of the flow-insensitive semantics w.r.t the flow-sensitive one is evaluated through the violation of assertions. The flow-insensitive semantics must found an assertion violation ($\Omega \in \textsc{Global}$) if and only if there exists a block b which also violates an assertion in the flow-sensitive semantics ($\exists p, \Omega \in \textsc{Local}(p)$).

Theorem 1 (Semantics completeness). *For any program p in FIC form,*

$$(\exists p, \Omega \in \textsc{Local}(p)) \iff \Omega \in \textsc{Global}$$

The implication $\exists p, \Omega \in \textsc{Local}(p) \Rightarrow \Omega \in \textsc{Global}$ trivially holds according to Corollary 1.

The other implication is more challenging because Global contains more states than the flow-sensitive semantics. The Theorem 2 below provides an equivalence which is needed between these states and the states in the flow-sensitive semantics. With this equivalence theorem we can prove Theorem 1. If there is a violation of an assert in the flow-insensitive semantics, then it is raised by some state s at block b, and there must be a flow-sensitive state at entry(b) which is equivalent to s and will thus also lead to a violation.

Theorem 2 (Equivalence preservation). *If $\forall p, \Omega \notin \textsc{Local}(p)$, then any state s of* Global *respects the following property $P(s)$.*

$$P(s): \qquad \forall b, \text{uses}(b) \subseteq \text{dom}(s) \implies (\exists s' \in \textsc{Local}(\text{entry}(b)), s \approx_{\text{uses}(b)} s')$$

Proof. We suppose that $\Omega \notin \textsc{Local}(p)$ for any p. Any state s of Global is the result of the application of a sequence $b_1, \ldots, b_n$ of blocks on $s_\emptyset$ as stated by lemma 2. The proof is made by strong induction on the size n of the sequence.

($n = 0$) No block is applied and $s = s_\emptyset$. For any block b such that $\text{uses}(b) = \emptyset$, property 2 requires that b is reachable and that there exists a state $s' \in \textsc{Local}(\text{entry}(b))$. Since the set of variables used by b is empty, $s \approx_{\text{uses}(b)} s'$.

($n+1$) We suppose that we have $s_1 \in [\![b_n]\!] \circ \cdots \circ [\![b_1]\!](s_\emptyset)$ and that $P(s)$ holds for any intermediate state s of this sequence. Let us take $s_2 \in [\![b_{n+1}]\!](s_1)$, we want to prove $P(s_2)$.

$$s_\emptyset \xrightarrow{[\![b_1]\!]} \cdots \xrightarrow{[\![b_n]\!]} s_1 \xrightarrow{[\![b_{n+1}]\!]} s_2$$

Let b such that $\text{uses}(b) \subseteq \text{dom}(s_2)$. We do a case study on $\text{defs}(b_{n+1}) \cap \text{uses}(b) = \emptyset$.

$\star$ Case $\text{defs}(b_{n+1}) \cap \text{uses}(b) = \emptyset$, the block b_{n+1} does not define variables used by b. It implies that all variables used by b are already in $\text{dom}(s_1)$ since $\text{uses}(b) \subseteq \text{dom}(s_2) = \text{dom}(s_1) \cup \text{defs}(b_{n+1})$. By $P(s_1)$, there exists a state $s'_1 \in \textsc{Local}(\text{entry}(b))$ such that $s'_1 \approx_{\text{uses}(b)} s_1 \approx_{\text{uses}(b)} s_2$ since the application of b_{n+1} on s_1 cannot change the valuation of $\text{uses}(b)$. We found s'_1 as a candidate for $P(s_2)$.

$\star$ Case $\text{defs}(b_{n+1}) \cap \text{uses}(b) \neq \emptyset$, b_{n+1} defines some variables used by b. By intrinsic FIC property 1 it defines all of them. The existence of $s_2 \in [\![b_{n+1}]\!](s_1)$ implies that $\text{uses}(b_{n+1}) \subseteq \text{dom}(s_1)$. By induction $P(s_1)$ holds so there exists $s'_1 \in \textsc{Local}(\text{entry}(b_{n+1}))$ such that $s'_1 \approx_{\text{uses}(b_{n+1})} s_1$.

Let us note p the entry of block b_{n+1}, p' its exit. Proving the existence of the intermediate state s'_3 in the figure below will help find the state s'_2 associated to s_2 in $P(s_2)$.

$$\begin{array}{ccc} s_1 & \xrightarrow{\quad [\![b_{n+1}]\!] \quad} & s_2 \\ \approx_{\text{uses}(b_{n+1})} \| & & \approx_{\text{uses}(b)} \| \\ s'_1 \in \textsc{Local}(p) & \xrightarrow{\quad [\![b_{n+1}]\!] \quad} & s'_3 \in \textsc{Local}(p') \end{array} \qquad \begin{array}{c} [\![d_k]\!] \circ \cdots \circ [\![d_1]\!] \\ \xrightarrow{\qquad\qquad} \\ \overset{=}{\approx_{\text{uses}(b)}} \end{array} \quad s'_2 \in \textsc{Local}(\text{entry}(b))$$

The semantic characterization of definitions (Invariant 2) ensures that there is a state $s'_3 \in [\![b_{n+1}]\!](s'_1) \subseteq \textsc{Local}(p')$ such that $s'_3 \approx_{\text{defs}(b_{n+1})} s_2 \in [\![b_{n+1}]\!](s_1)$. Since $\text{uses}(b) \subseteq \text{defs}(b_{n+1})$ by hypothesis, we can restrict the equivalence: $s'_3 \approx_{\text{uses}(b)} s_2$.

Since b_{n+1} defines the variables used by b, and since $\forall p, \Omega \notin \textsc{Local}(p)$, the intrinsic FIC property 2 implies the existence of a non-altering path $p' \xrightarrow{d_1} \ldots \xrightarrow{d_k} \text{entry}(b)$ associated to s'_3. The property ensures the existence of $s'_2 \in [\![d_k]\!] \circ \ldots [\![d_1]\!](s'_3)$ such that $s'_2 \approx_{\text{uses}(b)} s'_3$. Also, $s'_2 \in \textsc{Local}(\text{entry}(b))$ because $\text{exit}(d_k) = \text{entry}(b)$. By transitivity $s'_2 \approx_{\text{uses}(b)} s_2$ and we found s'_2 with the good properties so that $P(s_2)$ holds.

By induction, $P(s)$ holds for any s resulting from a sequence of blocks and thus it holds for any state of the flow-insensitive collecting semantics.

We can now make the complete proof of our central Theorem 1.

Proof. ($\Rightarrow$) Trivially holds by Corollary 1.

($\Leftarrow$) Let us suppose that there is no program point p such that $\Omega \in \textsc{Local}(p)$ but that $\Omega \in \textsc{Global}$. Then there exists a (potentially infinite) sequence of blocks $b_1, \ldots, b_n$ such that $\Omega \in [\![b_n]\!] \circ \cdots \circ [\![b_1]\!](s_\emptyset)$. Let us consider the state $s \neq \Omega$ such that $s \in [\![b_{n-1}]\!] \circ \cdots \circ [\![b_1]\!]$ and $\Omega \in [\![b_n]\!](s)$. To have such output state from applying b_n, we necessarily have that $\text{uses}(b_n) \subseteq \text{dom}(s)$. Since $P(s)$ by Theorem 2, and since $\forall p, \Omega \notin \textsc{Local}(p)$, there exists a flow-sensitive state $s' \in \textsc{Local}(\text{entry}(b_n))$ such that $s' \approx_{\text{uses}(b_n)} s$. Since the behavior of a block can only depend on its used variables by property 2, if there is an assert violated by s in b_n it is also violated by s'. So $\Omega \in [\![b_n]\!](s) \subseteq \textsc{Local}(\text{exit}(b_n))$ and we found a contradiction. The hypothesis that $\forall p, \Omega \notin \textsc{Local}(p)$ is false and we proved that $\Omega \in \textsc{Global} \implies \exists p, \Omega \in \textsc{Local}(p)$.

6 Transformation to Flow Insensitive Complete form

This section of the paper makes the simplifying assumptions that the program is *well-structured* and *terminating* and presents an algorithm to transform an SSA program into a FIC one.

A *well-structured* program comes from a structured language such as a While language. A more precise definition is available in Appendix.

A *terminating* program is either one that has a failed assertion, or one where for all reachable states s in p we can find a non-blocking path from p to the exit p_{ex}.

Definition 8 (Terminating program). *A program is* terminating *iff*

$$\Big(\forall p, \forall s \in \textsc{Local}(p), \exists p \xrightarrow{b_1} \ldots \xrightarrow{b_n} p_{ex}, [\![b_n]\!] \ldots [\![b_1]\!](s) \neq \emptyset\Big) \vee \exists p, \Omega \in \textsc{Local}(p)$$

```
b = false;
if (true) {
  while (true) {}
}
assert(b)
```

```
b = false;
if (true) {
  while (true) {};
  end = true
}
assert(end) ; assert(b)
```

Fig. 4: Infinite loops need a variable to assess their termination

Issues with infinite executions Infinite executions are problematic to ensure the existence of a path from dominator to dominated. For instance consider the program on the left of Figure 4.

Let us consider the block containing the assignment `b = false`. Any state out of this block will go into the infinite loop and cannot reach the assertion. Thus this program does not satisfy FIC property 2. To satisfy the property, we would need to artificially introduce a variable that can only be assigned after the loop and we would need to add a use of such a variable in the block of the assertion, as we did on the right program of Figure 4.

6.1 Sufficient conditions for FIC form

The intrinsic property 2 we expect from the FIC form is difficult to ensure in the general case as it relies on the semantics of paths. The main idea of our algorithm is to look at the paths in the dominance tree from definitions to uses and ensure that they are *constant*. This is simpler than checking the existence of a non-altering path. If the program is well-structured and terminating, constanteness in the dominance tree ensures the existence of a non-altering path.

Constant def-use path A definition point p of a variable x always dominates its usage in a block b: it dominates $\text{entry}(b)$. We must ensure that the path from p to $\text{entry}(b)$ is *constant* for the set of variables $\text{uses}(b)$.

Definition 9 (Constant path). *Let V be a set of variables and let p and p' be two program points such that p dominates p' and such that $p \to p_1 \to \cdots \to p_n \to p'$ is the path in the dominance tree from p to p'. The path is* constant *for V if for all the points p_i in $\{p_1, \ldots, p_n, p'\}$, p_i is either a joining point or its unique predecessor block b does not contain an assume nor definitions of V.*

For instance p_0 dominates p_3 but the path $p_0 \to p_1 \to p_3$ is not constant for any set V because p_3 has exactly one predecessor block, b_4 and it contains an assume.

Definition 10 (Constant paths completeness). *A program is* constant paths complete *if and only if for any blocks b and b', if $\text{defs}(b) \cap \text{uses}(b') \neq \emptyset$, then there is a constant path from $\text{exit}(b)$ to $\text{entry}(b')$ for $\text{uses}(b')$.*

Such property on the program is both easy to ensure and to check since we only have to look at the dominance tree and add copies to split the def-use path of a variable into two constant paths. The transformation of next section 6.2 directly enforces this property.

Theorem 3. *A well-structured, terminating and constant paths complete program satisfies the intrinsic FIC property 2.*

To prove this theorem we rely on a lemma: the constant paths imply the existence of a non-blocking path if the program is well-structured and terminating.

Lemma 3 (Existence of a non-blocking path). *In a well-structured terminating program, either there exists a point p such that $\Omega \in \textsc{Local}(p)$ or for any points p and p', if there is a constant path from p to p' then for any state s reaching p, there exists a non-blocking path from p to p' such that p only appears as the first point of the path.*

Proof. The proof is available in Appendix C. It proceeds by recurrence on the length of the constant path, and for each pair p_i, p_{i+1} it reasons by induction on the syntax of the program. Most cases of pairs where p_i dominates a point p_{i+1} in the program show an obvious path for any $s \in \textsc{Local}(p)$, or the direct domination is not a constant path. One case is to consider with care: the conditional. Indeed the entry of the conditional dominates its exit, a joining point and the path between the two is constant. However, to ensure that a state reaching the entry will reach the exit requires the termination of the program. Otherwise, the state may start an infinite loop in a branch, never to leave it to reach the exit, as shown on Figure 4.

The proof of the Theorem 3 is the following.

Proof. If there exists p such that $\Omega \in \textsc{Local}(p)$ then the intrinsic FIC property 2 trivially holds. Let us suppose that it is not the case. Let us take b and b' such that $\text{defs}(b) \cap \text{uses}(b') \neq \emptyset$. Then by constant paths completeness there exists a constant path from $\text{exit}(b)$ to $\text{entry}(b')$ for $\text{uses}(b')$. Let us take $s \in \textsc{Local}(\text{exit}(b))$. By lemma 3, and since the program is terminating, there is a non-blocking path $\text{exit}(b) \xrightarrow{b_1} \dots \xrightarrow{b_n} \text{entry}(b')$ such that $[\![b_n]\!] \circ \cdots \circ [\![b_1]\!](s) \neq \emptyset$. We only need to show that this path is non-altering for the variables of $\text{uses}(b')$. All definitions of $\text{uses}(b')$ must dominate their use in b'. Thus if some b_i modifies $\text{uses}(b')$ then $\text{exit}(b_i)$ is a dominator of b'. It can strictly dominate or be dominated by $\text{exit}(b)$. If $\text{exit}(b_i)$ is strictly dominated by $\text{exit}(b)$ we found a program point in the dominance path from $\text{exit}(b)$ to $\text{entry}(b')$ which violates the constanteness. This case is thus impossible. In the other case, $\text{exit}(b_i)$ strictly dominates $\text{exit}(b)$ but defines some variables used by b' and thus we are violating constant paths completeness since $\text{exit}(b)$ is in the way of the constant path from definitions in b_i to use in b'. So b_i cannot exist, no definition of $\text{uses}(b')$ can be encountered on the path and thus it respects the intrinsic FIC property 2: $\exists s' \in [\![b_n]\!] \circ \cdots \circ [\![b_1]\!](s)$ such that $s \approx_{\text{uses}(b')} s'$.

6.2 Transformation of a SSA program form to FIC form

Our transformation algorithm is developed in Algorithm 1. It proceeds as such: for any block b' whose uses has not been checked, we explore the dominance tree of the program points from its entry to the top of the dominance tree. During this exploration, the path from the current program point p to entry(b') is constant for uses(b'). When the path is no longer constant, we introduce copies at the current program point p to ensure the intrinsic FIC property 1. The introduction of copies changes the uses of the predecessors blocks of p, which must be checked again and is placed in the workset W. It is thus more efficient to check the blocks whose entry point are the lowest in the dominance tree first (line 45).

6.3 Correction of the transformation

The algorithm preserves the invariants of the SSA form (unique definition point and dominance of the definitions over the uses). These properties are available as lemmas in Appendix D and rely on the following invariant on the call context of procedure Check point.

Lemma 4 (Program point invariant). *The procedure* Check point *is always called with a program point p which dominates the entry point of b'. Let $p \to \cdots \to \text{entry}(b')$ be the path in the dominance tree from p to the entry of b'. This path is constant for* uses(b').

Proof. The proof is made by recurrence on the recursive calls of Check point. If the invariant on the path does not hold we do not make another call. The complete proof is in Appendix D.

A direct consequence of this lemma is that p and b' preserve this relation in the call to procedure Add missing variables.

To prove that the algorithm ensures constant path completeness on the final program, we rely on the following lemma. When the algorithm terminates no block is left in W ensuring the completeness.

Lemma 5 (Constant paths enforcement). *At each iteration of the loop, line 44, if a block b' is not in W then for any other block b, if* $\text{defs}(b) \cap \text{uses}(b') \neq \emptyset$ *then there is a constant path from* exit(b) *to* entry(b') *for* uses(b').

Proof. The complete proof is in Appendix D. At the loop entry the invariant holds since all blocks are in W. It is then preserved through the iteration. For the preservation, we need to check the newly marked block b', selected in the loop iteration, and we need to check that the invariant still holds for the blocks that were and still are out of the workset W.

For b', the invariant on the program point is given by lemma 4.

As for the other blocks still out of W, we did not change their uses (or they would have been added to W). But we did not change the definition points either: we only add definitions, never remove them. Thus for all blocks b'' in W before

Algorithm 1 Transformation

function GET COPY(u, p) ▷ u is a source variable
  if p is a joining point **then**
    if $\exists u', \forall b'' \in \text{pred}(p), \exists u''$ such that $\text{source}[u''] = u \wedge u' \overset{\phi}{\leftarrow} u'' \in b''$ **then**
      return u'
    else
      Let u' be a fresh version of u
      $\text{source}[u'] \leftarrow u$
      for $\forall b'' \in \text{pred}(p)$ **do**
        Let u'' be a fresh version of u
        Add $u'' \leftarrow u$ in component c of b''.
        Add $u' \overset{\phi}{\leftarrow} u''$ in component ϕ b''.
        b'' is added to W
      return u'
  else
    $b \leftarrow \text{pred}(p)$
    if $\exists u'$ such that $\text{source}[u'] = u \wedge u' \in \text{defs}(b)$ **then**
      return u'
    else
      Let u' be a fresh version of u
      $\text{source}[u'] \leftarrow u$
      Add $u' \leftarrow u$ in component c of b
      b'' is added to W
      return u'
procedure ADD MISSING VARIABLES(p, b')
  for $m \in \text{uses}(b') \setminus \text{defs}(p)$ **do**
    $u \leftarrow \text{source}[m]$
    $u' \leftarrow$ GET COPY(u, p)
    Replace every use of m in b' by a use of u'
procedure CHECK POINT(p, b') ▷ p dominates b'
  if p is a joining point **then**
    if $\exists b'' \in \text{pred}(p)$, $\text{defs}(b'') \cap \text{uses}(b') \neq \emptyset$ **then**
      ADD MISSING VARIABLES(p, b')
    else
      CHECK POINT(Direct dominator of p, b')
  else
    $b \leftarrow \text{pred}(p)$
    if $\text{defs}(b) \cap \text{uses}(b') \neq \emptyset$ or b contains an assume **then**
      ADD MISSING VARIABLES(p, b')
    else
      CHECK POINT(Direct dominator of p, b')
procedure TRANSFORM( )
  $W \leftarrow$ all blocks
  For all variables v, $\text{source}[v] = v$
  while $W \neq \emptyset$ **do**
    Let b' be one of the lowest blocks of W (in the dominance tree)
    Mark b' as *unmodified*
    CHECK POINT(entry(b'), b')

and after the loop iterations, the uses have not changed and the definitions of these uses neither, the set of blocks b such that $\text{uses}(b'') \cap \text{defs}(b) \neq \emptyset$ remains the same. The paths are still constant as we did not add assumes nor did we add definitions for existing variables, which include $\text{uses}(b'')$ and $\text{defs}(b)$.

The loop invariant of line 44 thus holds.

A similar lemma can be proved to ensure comprehensive definition coverage.

Lemma 6 (Comprehensive definitions enforcement). *At each iteration of the loop, line 44, if a block b' is not in W then for any other block b, if $\text{defs}(b) \cap \text{uses}(b') \neq \emptyset$ then $\text{uses}(b') \subseteq \text{defs}(b)$.*

Proof. The proof is made on a similar fashion than the previous lemma.

Theorem 4 (Termination). *The procedure* Transform *terminates.*

Proof. The procedure terminates if each block can be added to W only a limited amount of time. To prove it, we show that the number of copies created is limited. In all the copies $\cdots \leftarrow u$ inserted by Get Copy, u is a variable from the source program (in SSA form). The function will not add a copy for the source variable u in block b if it already contains one. Even in the case where p is a joining point we will not add copies twice. Indeed if p is a junction point, then the first time Get copy will be called, all the direct predecessors of p will receive a copy of u, and therefore the condition line 3 will be satisfied at the next call. Since the variables of the source program and the program points are limited, the procedure will add blocks to W a limited amount of time.

Complexity We propose an asymptotic estimation of the time complexity of our transformation. The transformation maintains a workset of modified blocks. Each time a block is picked from this workset, it runs a number of operations that is proportional to the height of the dominance tree. We call H this height. It remains to over-approximate the size of the workset. Initially each block belongs to it. We call B the number of blocks. But a block b may be put again in the workset by function Get Copy after adding new variable copies to b. This operation can not occur more than the number of variables in the original SSA program. We call V this number. At worst, the number of operations is then proportional to $H \cdot V \cdot B$.

7 Experiments

For our experiments, we did not exercise a complete analysis because we don't have abstract domains that are well suited to our notion of flow-insensitive analysis. Instead, we measure the number of variables generated by our FIC transformation and compare the number of variables in a FIC program with the number in the original program.

We implemented the transformation described in Section 6.2 in OCaml on top of the Sawja library [9] which parses Java bytecode programs. The input of

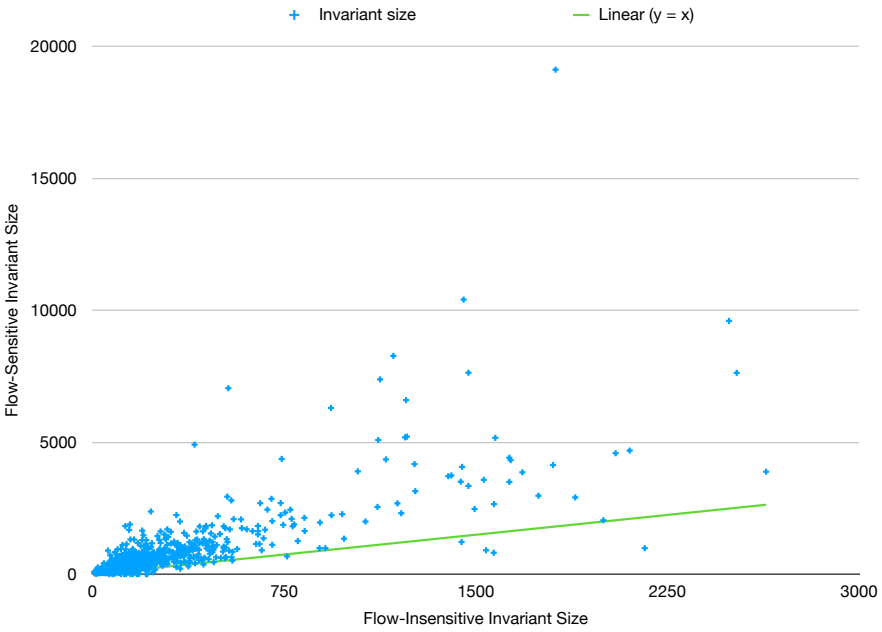


Fig. 5: Comparison of the number of variables introduced by the FIC transformation with the number of variables in a flow-insensitive analysis

our transformation is the JBirSSA intermediate representation which is already in SSA form. The benchmark used is composed of soot-2.5.0, an optimization framework, jtopas-0.8, a parsing java library, and finally ivy-2.5.0, a dependency manager and sub-project of the Apache Ant Project. The whole represents more than 40K functions. For graphs readability we remove 7 functions from this benchmark as the size of the invariants were important. We comment these missing points below.

In term of execution time the FIC transformation rarely dominates the time of the SSA transformation.

For the first experiment, we compare the number of variables in a FIC program with the expected size of invariants in a textbook flow-sensitive analysis (on the original program). This estimation is computed as the product:

$$|\text{number of variables}| \times |\text{number of program points}|$$

Figure 5 displays this comparison. A reference line of equation $y = x$ confirms that the textbook analysis globally requires to track more versions of variables than the FIC form. In this figure, for the removed functions, the number of FIC variables was greatly inferior to the product for all but one.

But some state of the art work try to keep their analysis as sparse as possible [8]. They keep the invariant only at junction points where the information

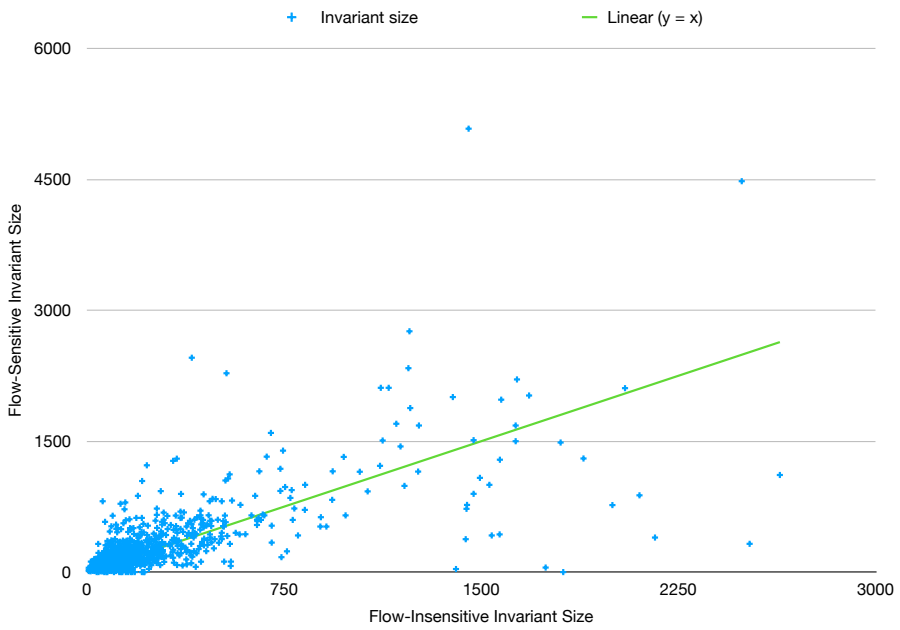


Fig. 6: Comparison of the number of variables introduced by the FIC transformation with the number of variables in a sparse flow-insensitive analysis

must be accumulated, while for other points it can be recomputed on demand. In a second experiment, we thus compare our number of variables to the number of joining points times the number of original variables. This corresponds to Figure 6 which also have a reference line of equation $y = x$. This figure shows that our number of variables is comparable to the number of versions required by sparse analyses. In this figure, the functions omitted had less FIC variables than the result of the product for all but three functions.

These results show that we can expect a flow-insensitive invariant whose size is in the same order of magnitude than flow-sensitive ones in state-of-the-art sparse analyses.

8 Related work

Flow-insensitive analyses have often been considered because of their efficiency, but few of them are able to provide relational invariants.

ABCD [3] is an analysis that check that array accesses are safe (that is within the bound of the array). Such analysis is used to remove the check around the accesses, hence speed up the program. To perform an efficient flow-insensitive analysis while keeping precision, ABCD uses the extended SSA form which is an intermediate form that closely resemble the SSI form. It uses the ϕ-functions at

junction point, but instead of σ-functions *before* the branching, it insert π-copies to the beginning of each branch. With its specific goal of ensuring inequalities, ABCD represents its invariant as a graph where an edge $v \to^c w$ denotes the constraint $w - v \leq c$ between the variables v and w and the constant c. This method cannot be applied to any relational abstract domain.

The idea to use an extended SSA form for relational analyses has been implemented to validate memory accesses [14] in a compiler setting. The analysis is based on abstract interpretation but not fully relational: it targets a semi-relational abstract domain of symbolic intervals. They do not provide semantic evidences of completeness.

Oh et al. present in [16] a general method for sparse analysis. Sparse analyses try to avoid unnecessary propagations in abstract fixpoint resolution. Their goal is then similar to us but they directly reason in term of abstract domain shape. We follow a more theoretical approach and directly reason on collecting semantics. We leave for further work the design of an abstract relational domain that would particularly fit our theoretical framework. Experiments in [16] are rather reassuring because they show a clear performance benefit when using flow insensitive analyses.

Hardekopf and Lin also demonstrate the benefit of sparse analysis for scalability of pointer analysis on large code bases [7]. They perform a first flow-insensitive analysis that generate conservative def-use information, and then use this information to perform a sparse flow-sensitive pointer analysis.

9 Conclusion

We provide a theoretical contribution to the quest for a fast but precise relational static analysis. We propose a variation of SSI program representation that permits to analyze a program in a flow insensitive manner without sacrificing the precision we could obtain with a flow sensitive approach.

The current work is a preliminary theoretical step before building a static analysis tool that would benefit from this idea. Our main theorem expresses a completeness property in term of collecting semantics but we do not provide guarantees about completeness of abstraction. The flow sensitive and flow insensitive semantics have different forms and their abstraction may behave differently. We believe the flow insensitive semantics has a promising potential for in-place abstraction algorithms. In particular, an abstract domain would greatly benefit from this semantics if it is equipped with an in-place abstract operator that over-approximates the operation $X \mapsto X \cup F(X)$. We believe a relational domain as Octogon could be enhanced with such features. This is left as future work.

An other requirement on the abstract domain is the capacity to track partial states. The global fixpoint represents properties on states with different domains and the analysis should not blur the information about one variable when it is potentially undefined on some paths. This problem has already been tackled by Liu and Rival [11] with relational domains.

Once we have equipped the FIC form with such analysis, we would like to perform experiments to measure efficiency gain and compare the abstract precision with a flow sensitive version.

Acknowledgments This work supported by a European Research Council (ERC) Consolidator Grant for the project VESTA, funded under the European Union's Horizon 2020 Framework Programme (grant agreement 772568).

References

1. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: POPL (1988)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language: Ph. D. Thesis. Datalogisk Institut (1994)
3. Bodík, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: PLDI. ACM (2000)
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conf. of POPL'78. pp. 84–96. ACM Press
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
6. Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: Giacobazzi, R. (ed.) Proc. of SAS'04. vol. 3148, pp. 212–227. Springer
7. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proc. of CGO'11. ACM Press
8. Henry, J., Monniaux, D., Moy, M.: PAGAI: A path sensitive static analyser. Electr. Notes Theor. Comput. Sci. **289** (2012)
9. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T.P., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static analysis workshop for java. In: Proc. fof FoVeOOS 2010. Lecture Notes in Computer Science, vol. 6528, pp. 92–106. Springer (2010)
10. Kildall, G.A.: A unified approach to global program optimization. In: Proc. of POPL'73. pp. 194–206. ACM Press
11. Liu, J., Rival, X.: Abstraction of optional numerical values. In: APLAS. pp. 146–166 (2015)
12. Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. Sci. Comput. Program. **75**(9), 796–807 (2010)
13. Miné, A.: The octagon abstract domain. In: Proc. of WCRE'01. p. 310. IEEE Computer Society
14. Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., Quintão Pereira, F.M.: Validation of memory accesses through symbolic analyses. ACM SIGPLAN Notices **49**(10), 791–809 (2014)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Berlin Heidelberg (2004), https://books.google.fr/books?id=RLjt0xSj8DcC
16. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for c-like languages. In: Proc. of PLDI'12. ACM Press
17. Pereira, F., Rastello, F.: Static Single Information form (2018), http://ssabook.gforge.inria.fr/latest/book.pdf, chapter 11 in the SSA-book
18. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. In: Proc. of POPL'85. pp. 291–299. ACM Press

A Collecting semantics

The proof of Lemma 1 is classical.

Proof. We prove both inclusions. Let us note $\textsc{Local}'(p)$ the right-hand of the equality.

$$\textsc{Local}'(p) = \bigcup_{p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p \text{ a path}} [\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset)$$

First, let us prove that

$$\textsc{Local}(p) \supseteq \textsc{Local}'(p)$$

We show by recurrence on n that for any path $p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p$, $[\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset)$ is included in $\textsc{Local}(p)$. If $n = 0$, the path is simply the program point p_{en} and we check that $\{s_\emptyset\} \subseteq \textsc{Local}(p_{\text{en}})$. For n fixed, let us suppose that for any path $p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p$, $[\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset)$ is included in $\textsc{Local}(p)$. Let us take a path $p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p' \xrightarrow{b_{n+1}} p$. Then $[\![b_{n+1}]\!] \circ [\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset) \subseteq [\![b_{n+1}]\!](\textsc{Local}(p')) \subseteq \textsc{Local}(p)$ by recurrence, monotony of $[\![b_{n+1}]\!]$ and by definition of $\textsc{Local}$ (since $p' \xrightarrow{b_{n+1}} p$). So the property holds for $n+1$ and thus by recurrence, for any $n \geq 0$.

Second, let us prove that for any p

$$\textsc{Local}(p) \subseteq \textsc{Local}'(p)$$

$\textsc{Local}(p)$ is defined as the least fixpoint of the equation, it is enough to prove that $\textsc{Local}'(p)$ is a fixpoint. First, $s_\emptyset \in \textsc{Local}'(p_{\text{en}})$ since the path of null length p_{en} can be considered. Then, let us take any p'.

$$\begin{aligned}
\bigcup_{p' \xrightarrow{b} p} [\![b]\!]\textsc{Local}'(p') &= \bigcup_{p' \xrightarrow{b} p} [\![b]\!] \bigcup_{p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p' \text{ a path}} [\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset) \\
&= \bigcup_{p' \xrightarrow{b} p} \bigcup_{p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p' \text{ a path}} [\![b]\!] \circ [\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset) \\
&= \bigcup_{p_{\text{en}} \xrightarrow{b_1} \dots \xrightarrow{b_n} p' \xrightarrow{b} p \text{ a path}} [\![b]\!] \circ [\![b_n]\!] \circ \dots \circ [\![b_1]\!](s_\emptyset) \\
&\subseteq \textsc{Local}'(p)
\end{aligned}$$

We can go from line 2 to 3 since the $[\![b]\!]$ is a join-morphism. So $\textsc{Local}'$ is a fixpoint of the equation, and since $\textsc{Local}$ is the least fixpoint it must be included in $\textsc{Local}'$.

Both terms are thus equal.

The proof of Lemma 2

Proof. The proof is made on the same principle than Lemma 1.

B Well-structured program

Let us consider structured programs from a While language: they are composed inductively of blocks b of atomic instructions (without assumes), sequence of sub-programs $P_1; P_2$, conditionals `if` c `then` P_1 `else` P_2 or finally of while-loop `while` c `do` P_1. These programs are transformed into a graph and then into the SSA form. By extension we call the SSA form obtained a *structured graph*.

The graph is build inductively on the syntax of the program. We suppose that sequence of atomic instructions are already packed into blocks. A statement in the program (block, sequence, conditional or loop) is transformed into a graph with a unique entry point and exit point.

Block The sequence of atomic instructions *body* is transformed into the simple node $\langle body, \emptyset, \phi \rangle$ in the graph, its entry and exit points are those of the graph.

Sequence Let $P_1; P_2$ be the program, and G_1 and G_2 the associated graphs for P_1 and P_2. Then we merge the exit of G_1 with the entry of G_2. The entry of the final graph is the entry of G_1 and the exit is the one of G_2.

Conditional Let `if` c `then` P_1 `else` P_2 be the program, and G_1 and G_2 the associated graphs for P_1 and P_2. We create two nodes, the affirmative $\langle assume(c), \emptyset, \emptyset \rangle$ and the negative $\langle assume(\neg c), \emptyset, \emptyset \rangle$. The entry of both new nodes are merged to be the entry of the final graph. Their exit are merged with the entry of G_1 and G_2 respectively. The exits of G_1 and G_2 are merged to be the exit of the whole graph.

Loop Let `while` c `then` P be the program, and G the associated graph for P. Again we create two nodes, an affirmative and a negative one for condition c. Their entries are merged to be the entry of the final graph. The exit of the affirmative node is merged with the entry of G. The exit of P is merged with the entry of the final graph. The exit of the negative node is the exit of the whole graph.

In case two nodes b_1 and b_2 are such that $\text{exit}(b_1) = \text{entry}(b_2)$ and $\text{exit}(b_1)$ has no other successors than b_2, and $\text{entry}(b_2)$ has no other predecessor than b_1, then the two nodes can be merged.

Definition 11 (Well-structured program). *A program as a graph is* well-structured *if it is the result of the transformation of a While-language program.*

We can make an inductive reasoning on the structure of such program., we suppose that the SSA transformation preserves the structure.

C Sufficient condition proof

The proof of lemma 3 require an induction on the structure of the program.

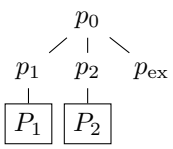
Proof. Let us suppose that there is no halting execution, i.e. no p such that $\Omega \in \textsc{Local}(p)$. By termination, for all p, for all states s reaching p, we will find

a path to the exit p_{ex}. Let us take any such p and s and let p' be a program point dominated by p such that there is a constant path from p to p'. We make a recurrence on the length of this path in the dominance tree. If the length of the path is null the invariant trivially holds. Otherwise, we suppose that we found a path up to point p_i of the dominance path, and we show that we can extend it to reach p_{i+1}. p_i strictly dominates p_{i+1}, and since the path is constant, p_{i+1} is either a joining point or its unique predecessor block b does not contain an assume. Let us prove by induction on the structure of the program that for any such pair of point (p_i, p_{i+1}) there is a non-blocking path not going through p_i twice. Let us take any $s \in \textsc{Local}(p_i)$ ($s \neq \Omega$ by hypothesis).

If the program is a block b of instructions, then in its SSA form there will be a unique block, without assume, such that the entry dominates the exit. The unique pair (p_i, p_{i+1}) possible is $(\text{entry}(b), \text{exit}(b))$. The path $\text{entry}(b) \xrightarrow{b} \text{exit}(b)$ is non-blocking so we found an adequate path for all pairs (p_i, p_{i+1}).

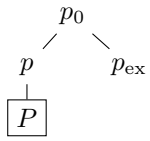
If the program is a sequence $P_1; P_2$, we suppose by induction that for all constant paths $p \to p'$ in P_1 or P_2, for all states s reaching p there is a non-blocking path to p' for s. Let us name p_1 the entry point of s_1, p_2 its exit which is also the entry of s_2 and finally p_3 the exit of s_2. For any pair (p_i, p_{i+1}) of direct dominance, either the pair is in P_1 or it is in P_2 (with p_2 being in both P_1 and P_2). The induction is enough to guarantee the existence of a path.

If the program is a conditional `if` c `then` P_1 `else` P_2, such that the invariant holds on P_1 and P_2, we suppose by induction that the invariant holds for any pair in P_1 and in P_1. Let b_1 and b_2 be the affirmative and negative blocks, and let p_0 be their common entry. p_1 and p_2 are the entry points of P_1 and P_2, and p_{ex} is their common exit point. We need to consider the following direct dominances: $p_0 \to p_i$, the ones inside P_i (including p_i but excluding p_{ex}) and $p_0 \to p_{\text{ex}}$.



The dominances $p_0 \to p_i$ are not control-free since p_i has a unique predecessor block containing an assume. For all direct dominances inside P_i we can use the induction. Finally we need to consider $p_0 \to p_{\text{ex}}$. Let $s \in \textsc{Local}(p_0)$. The state satisfy either c or its negation. So there is a branch i such that $[\![b_i]\!](s) = \{s\} \subseteq \textsc{Local}(p_i)$. The execution must terminate and there is no fail state so there exist a path $p_i \xrightarrow{b'_1} \dots \xrightarrow{b'_n} p_{\text{ex}}$ for $s \in \textsc{Local}(p_i)$. So we found a path $p_0 \xrightarrow{b_i} p_i \xrightarrow{b'_1} \dots \xrightarrow{b'_n} p_{\text{ex}}$ for s such that $[\![b'_n]\!] \circ \dots \circ [\![b'_1]\!] \circ [\![b_i]\!](s) \neq \emptyset$.

Finally, if the program is a while loop `while` c `do` P then we assume that any pair of program points in P satisfies the invariant. Let b_1 and b_2 be the affirmative and negative blocks respectively, let p_0 be their common entry point. Let p_{ex} be the exit of the negative block, p the exit of the affirmative one and entry of P.



We need to consider the following direct dominances: $p_0 \to p$, $p_0 \to p_{\text{ex}}$ and finally the ones inside P_1. The first two are not constant since there is a unique predecessor block to p and to p_{ex} which contains an assume. As for the ones in P_1 we simply use the induction.

Additionally, in all those inductive cases for the program, the path goes from p_i to p_{i+1} without going twice through p_i or a point dominating it.

So by induction on the syntax, for any pair (p_i, p_{i+1}) in the program such that there is a constant path from p_i to p_{i+1}, for any $s \in \textsc{Local}(p)$, there is a non-blocking path from p_i to p_{i+1} for s. This path does not go twice through p_i nor a point dominating it. By recurrence the path exists for any pair p, p' with a constant path between them and the path does not go twice through p.

D Algorithm proofs

The following lemmas 7 and 8 ensure that the transformed program has preserved its SSA invariant 1.

Lemma 7 (Uniqueness of definitions). *In the transformed program, for all variables v, v has a unique definition point.*

Proof. This property holds in SSA form and the algorithm preserves it. Indeed, new definitions are only added for fresh variables in function Get Copy and they are added for a unique program point p.

The dominance is proved through an invariant on call context of procedure Check point stated by Lemma 4 whose proof is developed below.

Proof. The proof is made by recurrence on the recursive calls. The first call is made by procedure Transform with $p = \text{entry}(b')$. The dominance is obvious and the path is only composed of point p. Such an empty path is necessarily constant. Now, let us suppose that we made a call of Check point with a point p and block b' which satisfies the invariant. The procedure Check point is called recursively on a direct dominator of p, let us ensure the invariant from this dominator p'. p' dominates p which dominates $\text{entry}(b')$ so p' obviously dominates $\text{entry}(b')$. Let us consider the path is the dominance tree from p' to $\text{entry}(b')$: $p' \to p \to \cdots \to \text{entry}(b')$, the subpath from p to $\text{entry}(b')$ is constant for $\text{uses}(b')$ by hypothesis. If p has a unique predecessor b, the recursive call is made only if b does not contains an assume nor definitions for $\text{uses}(b')$. Thus the path from p' to $\text{entry}(b')$ is constant. In the other hand, if p is a joining point, then the recursive call is made only if p is not a definition point for some variables of $\text{uses}(b')$. In this case, the path from p' to $\text{entry}(b')$ remains constant for $\text{uses}(b')$.

Thus the invariant on the call context of Check point holds.

Lemma 8 (Definition dominance preservation). *For any variable v in the transformed program, the definition point of v is unique and dominates all its uses.*

Proof. The entry program, in SSA form, is supposed to respect this lemma by Invariant 1. The algorithm changes the usage of the blocks but also introduces new variables. However, no definition is moved or removed.

First, we check that when definitions are added, they are unique and dominate their uses. Then, we check that when the uses are modified in a block, they are still dominated by the definition.

First, let us show that the new variables have a unique definition point dominating all its uses. New variables are introduced by the function GET COPY with two cases: the new variable is either the one u' returned or it is a local variable u''. The local variable u'' is defined in the copies c of block b'' and directly used in its ϕ-definitions ϕ. The definition is unique and dominates the use in block b''. As it is a variable internal to the block, it is not included in uses(b'') nor defs(b''), it cannot be used by another block. As for u', the definitions are only added in the predecessor blocks of p, which is thus the unique definition point for u'. For the dominance, u' will be used in b' after ADD MISSING VARIABLE, and entry(b') is dominated by the definition point p by lemma 4.

Now, let us check that the other uses, of the variable u, added by GET COPY in the predecessors of p, are dominated by the definition point of u, noted p_u. First, if we are adding uses of u in the predecessors of p, then $p_u \neq p$. Next, let us prove that p_u strictly dominates p in the call of GET COPY. By hypothesis, p_u dominates entry(b') because b' uses u. By lemma 4, the path from p to entry(b') in the dominance tree is constant for uses(b') and thus does not contain definition points for uses(b'). In particular, it cannot contain p_u. So p_u and p dominates entry(b'), $p_u \neq p$ and p does not dominate p_u. The only solution is that p_u strictly dominates p. The uses added by GET COPY fall in two cases: either p is a joining point in which case for all its predecessor b'' we add a use of u, or there is a unique predecessor b for p for which we add a use of u. In the first case, let us prove that p_u dominates all predecessors b'' of program point p. If they were not dominated by p_u, there would exist a path from the entry to p which goes through b'' but not through p_u. This contradicts the dominance of p_u over p, and thus p_u dominates all usages of u added in the direct predecessors of p. As for the second case, we add a use for u in the unique predecessor block b of p. The definition point of u strictly dominates $p = \text{exit}(b)$ and thus it dominates non-strictly entry(b) and the use of u in b.

In all the cases the uniqueness and the dominance of the definition points are preserved.

The algorithm ensures the constant path completeness, as stated by Lemma 5.

Proof. Initially, all blocks are in W so the invariant holds. Let us suppose that the invariant holds at the beginning of a loop iteration and that we selected some block b' to be marked *unmodified*. In the procedure some blocks may be

marked *modified* but those left untouched and marked *unmodified* should still satisfy the property.

First, let us check the property on b'. Let us prove that, after executing CHECK POINT, if $\text{defs}(b) \cap \text{uses}(b')$, then $\text{exit}(b)$ dominates $\text{entry}(b')$. The recursive calls of CHECK POINT will end with a call to ADD MISSING VARIABLES. Let p be the point on which ADD MISSING VARIABLES is called. By lemma 4, there is a constant path from p to $\text{entry}(b')$ for $\text{uses}(b')$. If we can prove that p is the only program point such that $\text{defs}(p) \cap \text{uses}(b') \neq \emptyset$, then the invariant holds.

Before the procedure, we can split the variables used by b' in two sets, the ones already defined by program point p, which is named C, and the ones missing M. p is the unique definition point of C. For all the missing variables M we call GET COPY to get a version u' defined by p (and only by p). This version replaces the uses of u in b'. b' now only uses variables whose definition point is p. Thus p is the only program point such that $\text{defs}(p) \cap \text{uses}(b') \neq \emptyset$ and the invariant holds.

Now let us check that the property was preserved for the other blocks that were marked *unmodified* at the loop entry. Whenever we change the use of a block in the procedures (except for b') we mark it as *modified*. Whenever we change the definitions of a block, we only add definitions, never remove them, and the new variables are not used by any other block than b' which we cover. Thus for all blocks b'' marked *unmodified* before and after the loop iterations, the uses have not changed and the definitions of these uses neither, the set of blocks b such that $\text{uses}(b'') \cap \text{defs}(b) \neq \emptyset$ remains the same. For each of these blocks we can still find a path from b to b'' for any state s out of b. We did not add assume that may block the execution. We did not modify the definitions of variables used by blocks b'' left *unmodified* and thus the effect of the blocks remains equivalent on $\text{uses}(b'')$. So this path is still constant after an iteration.

The loop invariant of line 44 thus holds.

The other property the transformation enforces is the comprehensive definitions coverage. The proof of Lemma 6 is below.

Proof. The proof is also made by recurrence on the loop iteration. The invariant holds at the loop entrance since no block is marked *unmodified*. We then suppose that it holds at the beginning of the loop and ensure that one iteration preserves it.

First, let us consider the block b' which has been selected. From the proof of the previous lemma we get that, after procedure CHECK POINT, all the blocks b such that $\text{uses}(b') \cap \text{defs}(b) \neq \emptyset$ have the same exit point that we note p. For any variable x in $\text{uses}(b')$ either it was in the set C of the variables defined by all the blocks b before the procedure (then $x \in \text{defs}(p) \subseteq \text{defs}(b)$), or it is a new version added by GET COPY. In this second case, the new version is defined by all blocks b preceding p. Thus, for all blocks b such that $\text{exit}(b) = p$, $\text{uses}(b') \subseteq \text{defs}(b)$.

Then let us consider the blocks b' that were and still are marked *unmodified*. By hypothesis, before the loop iteration, all the blocks b which defines some of $\text{uses}(b')$ defines all of these variables. After the loop iteration, if b' has not been

marked *modified* then its uses have not changed. The blocks b containing the definitions are also the same since we did not remove any definition and the ones we added are for new variables, that cannot be used by b'. So the invariant is preserved.