# Unconstrained Variable Oracles for Faster Numeric Static Analyses

Vincenzo Arceri[1][0000-0002-5150-0393], Greta Dolcetti[1][0000-0002-2983-9251], and Enea Zaffanella[1][0000-0001-6388-2053]

Department of Mathematical, Physical and Computer Sciences
University of Parma, Parma I-43124, Italy
`vincenzo.arceri@unipr.it`, `greta.dolcetti@studenti.unipr.it`, `enea.zaffanella@unipr.it`

**Abstract.** In the context of static analysis based on Abstract Interpretation, we propose a lightweight pre-analysis step which is meant to suggest, at each program point, which program variables are likely to be unconstrained for a specific class of numeric abstract properties. Using the outcome of this pre-analysis as an oracle, we simplify the statements of the program being analyzed by propagating this lack of information, aiming at fine-tuning the precision/efficiency trade-off of the analysis. A preliminary experimental evaluation shows that the idea underlying the approach is promising, as it improves the efficiency of the more costly analysis while having a limited effect on its precision.

**Keywords:** Abstract Interpretation · Static Analysis · Unconstrained Variables · Abstract Compilation.

## 1 Introduction

Static analyses based on Abstract Interpretation [12] correctly approximate the collecting semantics of a program by executing it on an abstract domain modeling the properties of interest. In the classical approach, which follows a pure program interpretation scheme, the concrete statements of the original program are abstractly executed step by step, updating the abstract property describing the current program state: while being correct, this process may easily incur avoidable inefficiencies and/or precision losses. To mitigate this issue, static analyzers sometimes apply simple, safe program transformations that are meant to better tune the trade-off between efficiency and precision. For instance, when trying to improve efficiency, the evaluation of a complex nonlinear numeric expression (used either in a conditional statement guard or as the right hand side expression in an assignment statement) may be abstracted into a purely nondeterministic choice of a value of the corresponding datatype; in this way, the overhead incurred to evaluate it in the considered abstract domain is avoided, possibly with no precision loss, since its result was likely imprecise anyway. On the other hand, when trying to preserve precision, a limited form of constant propagation may be enough to transform a nonlinear expression into a linear one, thereby allowing for a reasonably efficient and precise computation on commonly used abstract domains tracking relational information. As another example, some tools apply a limited form of loop unrolling (e.g., unrolling the first iteration of the loop [8]) to help the abstract domain in clearly separating those control flows that cannot enter the loop body from those that might enter it; this transformation may trigger significant precision improvements when the widening operator is applied to the results of the loop iterations.

Sometimes, the program transformations hinted above are only performed at the semantic level, without actually modifying the program being analyzed; hence, the corresponding static analysis tools can still be classified as pure program interpreters. However, in principle the approach can be directly applied at the syntactic level, so as to actually translate the original program into a different one, thereby moving from a pure program interpretation setting to a hybrid form of (abstract) compilation and interpretation. Note that the term *Abstract Compilation*, introduced in [21,31], sometimes has been understood under rather constrained meanings: for instance, [17] and [1] assume that the compiled abstract program is expressed in an existing, concrete programming language; [9] and [32] focus on those inefficiencies that are directly caused by the interpretation step, without considering more general program transformations. Here we adopt the slightly broader meaning whereby portions of the approximate computations done by the static analysis tool are eagerly performed in the compilation (i.e., program translation) step and hence reflected in the abstract program representation itself. Clam/Crab [18] and IKOS [10] are examples of tools adopting this hybrid approach for the analysis of LLVM bitcode, leveraging on specific intermediate representations designed to accomodate several kinds of abstract statements. A similar approach is adopted in LiSA [16,27], to obtain a uniform program intermediate representation when analyzing programs composed by modules written using different programming languages.

*Paper contribution.* Adopting the Abstract Compilation approach, we propose a program transformation that is able to tune the trade-off between precision and efficiency. The transformation relies on an *oracle* whose goal is to suggest, for each program point, which program variables are *likely unconstrained* for a target numeric analysis of interest. By systematically propagating the guessed lack of information, the oracle will guide the program transformation so as to simplify those statements of the abstract program for which the target analysis is *likely* unable to track useful information.

We model our oracles as pre-analyses on the abstract program, considering two Boolean parameterizations and thus obtaining four possible oracle variants: we will have *non-relational* or *relational* numeric analysis oracles and each of them can be *existential* or *universal*. The proposed program transformation can be guided by any one of these variants, allowing different degrees of program simplification, thereby obtaining different trade-offs between the precision and the efficiency of the target analysis. It is important to highlight that the oracles we are proposing have no intrinsic correctness requirement; as we will discuss in Section 2, whatever oracle is adopted to guess the set of unconstrained variables, its use will always result in a correct program transformation. The (im-)precision of the oracle guesses can only affect the precision and efficiency of the target numeric analysis: *aggressive* oracles, which predict more variables to be unconstrained, will result in faster but potentially less precise analyses; *conservative* oracles, by predicting fewer unconstrained variables, will result in slower analyses, potentially preserving more precision.

Our proposal will be experimentally evaluated on two benchmark suites: the first one, distributed with PAGAI [20], is a classical set of benchmarks for WCET (worst-case execution time) analysis; the second one contains 10 Linux drivers taken from the SV-COMP repository. We will test the four oracle variants on these benchmarks, considering as target analyses the classical numeric analyses using the abstract domains of intervals and convex polyhedra. This experimental evaluation allows us to measure the trade-off between efficiency and precision of the proposed variants, showing that it is possible to fine-tune the efficacy of the oracle and, in turn, of the program transformation.

*Paper structure.* In Section 2, after introducing the notion of likely unconstrained variable for a target numeric analysis, we define four different oracles as variants of a dataflow analysis tracking variable unconstrainedness; we also formalize the program transformation that, guided by these oracles, simplifies the abstract program being analyzed. The design and implementation of our experimental evaluation are described in Section 3, where we also comment the results obtained on the considered benchmarks. Related work is briefly discussed in Section 4, while Section 5 concludes, also describing future work.

## 2 Detecting Likely Unconstrained Variables

In the concrete (resp., abstract) semantics of programming languages, the evaluation of an expression is formalized by a suitable set of semantic equations, which specify the result of the expression by using concrete (resp., abstract) operators to combine the current values of program variables, as recorded in the concrete (resp., abstract) environment. The efficiency of the evaluation process can be improved by propagating *known* information (e.g., constant values). In the abstract evaluation case, efficiency improvements may also be obtained by propagating *unknown* information. As an example, when evaluating the numeric expression $x + expr$ using the abstract domain of intervals [12], if no information is known about program variable $x$, then it is likely that no information at all will be known about the whole expression. Even when considering the more precise abstract domain of convex polyhedra [14], if $x$ is unconstrained and $expr$ is a rather involved, non-linear expression, then it is likely that little information will be known about the whole expression. Hence, in both cases, there is little incentive in providing an accurate (and maybe expensive) over-approximation for the subexpression $expr$.

In this section we propose a heuristic approach to efficiently detect and propagate this lack of abstract information. We focus on the concept of *likely* *unconstrained* (LU) variables: we say that $x \in \mathbb{Var}$ is an LU variable (at program point $p$) if the considered static analysis is likely unable to provide useful information on $x$. Thus, whenever $x$ is LU, the static analysis can just forget it, since it brings little knowledge. It is worth stressing that the one we are proposing is an informal and heuristics-based definition, with no intrinsic correctness requirement: as we will see, whatever technique is adopted to compute the set of LU variables, its use will always result in a correct static analysis; the only risk, when forgetting too many variables, is to suffer a greater precision loss.

### 2.1 A dataflow analysis for LU variables

We now informally sketch several variants of a forward dataflow analysis for the computation of LU variables, to be used on a CFG representation of the source program; if needed, the approach can be easily adapted to work with alternative program representations.

*The transfer function for non-relational analyses.* Let $\mathbb{Stmt}$ be the set of statements occurring in the CFG basic blocks, which for simplicity we assume to resemble 3-address code. Then, given the set $lu \subseteq \mathbb{Var}$ of variables that are LU before (abstractly) executing $s \in \mathbb{Stmt}$, the transfer function

$$[\![\cdot]\!] \colon \mathbb{Stmt} \times \wp(\mathbb{Var}) \to \wp(\mathbb{Var})$$

computes the set $[\![s]\!](lu)$ of variables that are LU after the execution of $s$. Clearly, the definition of $[\![\cdot]\!]$ depends on the target analysis: a more precise abstract domain will probably expose fewer LU variables. We first consider, as the reference target analysis, the *non-relational* abstract domain of intervals [12]. Intuitively, in this case a variable is LU if the corresponding interval is (likely) unbounded, i.e., $[-\infty, +\infty]$. Note that, in our definitions, we explicitly disregard those constraints that can be implicitly derived from the variable datatype; for instance, for a nondeterministic assignment $(x \leftarrow ?) \in \mathbb{Stmt}$, even when knowing that $x$ is a signed integer stored in an 8-bit word, we will ignore the implicit constraints $-128 \leq x \leq 127$ and flag the variable $x$ as LU. Hence, the transfer function for nondeterministic assignments is

$$[\![x \leftarrow ?]\!](lu) = lu \cup \{x\}.$$

The transfer function for the assignment statement $(x \leftarrow y \mathbin{op} z) \in \mathbb{Stmt}$, where $op \in \{+, -, *, /, \%\}$ is an arithmetic operator and $x, y, z \in \mathbb{Var}$, is

$$[\![x \leftarrow y \mathbin{op} z]\!](lu) = \begin{cases} lu \smallsetminus \{x\}, & \text{if } y \notin lu \text{ and } z \notin lu, \\ & \text{or if } op = \% \text{ and } z \notin lu, \\ & \text{or if } op = - \text{ and } y = z; \\ lu \cup \{x\}, & \text{otherwise.} \end{cases}$$

Namely, $x$ is going to be constrained (and hence removed from set $lu$) when both $y$ and $z$ are constrained, or when $z$ is constrained and $op$ is the modulus operator, or when hitting the corner case $x \leftarrow y - y$. For the special case when the third variable $z$ is replaced by a constant argument $k \in \mathbb{Z}$, we can define

$$[\![x \leftarrow y \mathbin{op} k]\!](lu) = \begin{cases} lu \setminus \{x\}, & \text{if } y \notin lu, \text{ or if } op = \%, \\ & \text{or if } op = * \text{ and } k = 0; \\ lu \cup \{x\}, & \text{otherwise.} \end{cases}$$

When evaluating Boolean guards, the abstract semantics works in a similar way: letting $(x \bowtie y) \in \mathbb{Stmt}$, where $x, y \in \mathbb{Var}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$, we can define

$$[\![(x \bowtie y)]\!](lu) = \begin{cases} lu \setminus \{x\}, & \text{if } y \notin lu; \\ lu \setminus \{y\}, & \text{if } x \notin lu; \\ lu, & \text{otherwise.} \end{cases}$$

Similar definitions can be easily provided for all the other statements of the language.

As already said above, the transfer function we are proposing is just a way to heuristically *suggest* LU variables and hence it is subject to *both* false positives and false negatives. A false positive is obtained, for example, when processing the assignments $y \leftarrow 0$ and $x \leftarrow y * z$: since $z$ is LU, the transfer function will also flag $x$ as LU, even though the interval analysis would compute $x \in [0, 0]$. A false negative is obtained by processing the Boolean guards $(y \geq 0)$, $(z \leq 0)$ and the assignment $x \leftarrow y + z$: the transfer function will predict variable $x$ to be constrained, even though the interval analysis computes $x \in [-\infty, +\infty]$.

*The case of relational analyses.* If the target static analysis is based on an abstract domain tracking relational information, such as the domain of convex polyhedra [14], then the notion of LU variable no longer corresponds to the notion of unboundedness (as an example, consider the constraint $x = y$). Hence, the definition of the transfer function can be refined accordingly. As an example, when assuming that the domain is able to track linear constraints, a relational version $[\![\cdot]\!]^{\mathrm{rel}} \colon \mathbb{Stmt} \times \wp(\mathbb{Var}) \to \wp(\mathbb{Var})$ of the transfer function for the arithmetic assignment statements can be defined as follows:

$$[\![x \leftarrow y \mathbin{op} z]\!]^{\mathrm{rel}}(lu) = \begin{cases} lu \setminus \{x, y, z\}, & \text{if } op \in \{+, -\}; \\ [\![x \leftarrow y \mathbin{op} z]\!](lu) & \text{otherwise;} \end{cases}$$

$$[\![x \leftarrow y \mathbin{op} k]\!]^{\mathrm{rel}}(lu) = \begin{cases} lu \setminus \{x\}, & \text{if } op = \%; \\ lu \setminus \{x, y\}, & \text{otherwise.} \end{cases}$$

Similarly, the relational version for the evaluation of Boolean guards can be defined as follows:

$$[\![(x \bowtie y)]\!]^{\mathrm{rel}}(lu) = lu \setminus \{x, y\}.$$

Once again, the definition of $[\![\cdot]\!]^{\mathrm{rel}}$ for the other kinds of statements poses no problem.

*The propagation of LU information.* Starting from the set $lu_{\mathrm{pre}}$ of variables that are LU at the start of a basic block, by applying function $[\![\cdot]\!]$ (resp., $[\![\cdot]\!]^{\mathrm{rel}}$) to each statement in the basic block we can easily compute the set $lu_{\mathrm{post}}$ of LU variables at the end of the basic block. In order to complete the definition of our dataflow analysis we need to specify how this information is propagated through the CFG edges. As a first option we can say that a variable $x$ is LU at the start of a basic block if there *exists* an edge entering the block along which $x$ is LU; this corresponds to an analysis defined on the usual powerset lattice

$$\mathsf{LU}^{\exists} \triangleq \langle \wp(\mathbb{Var}), \subseteq, \varnothing, \mathbb{Var}, \cap, \cup \rangle,$$

having set inclusion as partial order and set union as join operator. This *existential* approach may be adequate when our goal is to obtain an *aggressive* LU oracle, which eagerly flags variables as LU, in particular when adopting the non-relational transfer function.

As an alternative, we can say that a variable $x$ is LU at the start of a basic block only if $x$ is LU along *all* the edges entering the block; this corresponds to an analysis defined on the dual lattice

$$\mathsf{LU}^{\forall} \triangleq \langle \wp(\mathbb{Var}), \supseteq, \mathbb{Var}, \varnothing, \cup, \cap \rangle,$$

having set intersection as join operator. When using this *universal* alternative, we will obtain a more *conservative* LU oracle, in particular when adopting the relational transfer function.

In all cases, the dataflow fixpoint computation is going to converge after a finite number of iterations, since the two transfer functions are monotone and the two lattices are finite. In summary, we have obtained four simple LU oracles ($\mathsf{LU}^{\exists}$, $\mathsf{LU}^{\forall}$, $\mathsf{LU}^{\exists}_{\mathrm{rel}}$, $\mathsf{LU}^{\forall}_{\mathrm{rel}}$) that, to some extent, should be able to guess which variables can be forgotten with a limited effect on the precision of the analysis; each of these can be used to guide a program transformation step that simplifies the target analysis, with the goal of improving its efficiency.

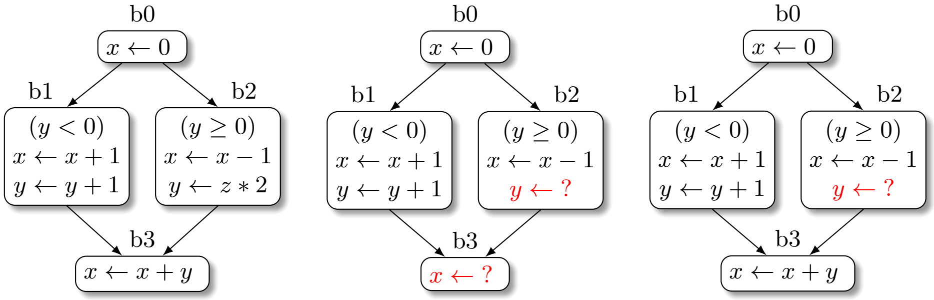
### 2.2 The program transformation step

Algorithm 1 describes how the information about LU variables computed by any one of the oracles described before can be exploited to transform the input CFG. Intuitively, the program transformation should instruct the target static analysis to forget those variables that are not worth tracking. Hence, for each basic block $bb \in N$, we retrieve the corresponding set $lu = \mathrm{LU}_{\mathrm{post}}(bb)$ of program variables that, according to the chosen oracle, are LU at the exit of the basic block; then, each assignment statement $x \leftarrow \mathit{expr}$ in $bb$ having as target a variable $x \in lu$ is replaced with the nondeterministic assignment $x \leftarrow ?$. Note that, for simplicity and ease of exposition, we are assuming that each program variable is assigned at most once in each basic block; this is not a significant restriction, since in most cases the input CFG satisfies much stronger assumptions, such as SSA form.

We now provide an example simulating the LU variable analysis and transformation steps on a simple portion of code, focusing on the $\mathsf{LU}^{\exists}$and $\mathsf{LU}^{\forall}$oracles, i.e., the non-relational case.

**Algorithm 1:** Program transformation

```
Input: ⟨$N, E$⟩ (input CFG), $\mathsf{LU}_{\text{post}}\colon N \to \wp(\mathbb{V}\mathrm{ar})$ (LU variables map)
foreach $bb \in N$ do
    let $lu = \mathsf{LU}_{\text{post}}(bb)$
    foreach $s = (x \leftarrow expr) \in bb$ do
        if $x \in lu$ then
            replace $s$ with $s' = (x \leftarrow ?)$ in $bb$
        end
    end
end
```



(a) Original CFG. (b) $\mathsf{LU}^{\exists}$ CFG. (c) $\mathsf{LU}^{\forall}$ CFG.

Fig. 1: The effect of LU variable propagation on a simple CFG.

*Example 1.* Consider the simple CFG in Figure 1a, defined on the set of program variables $\mathbb{V}\mathrm{ar} = \{x, y, z\}$. Note that the CFG has no loops at all: this is a deliberate choice for exposition purposes, since our goal here is to show the basic steps of the LU analysis, rather than any detail related to the fixpoint computation (whose convergence poses no problems at all, as explained before). When considering the $\mathsf{LU}^{\exists}$ variant of the analysis, the set of LU variables at the start of the initial block b0 is initialized as $\mathrm{LU}_{\mathrm{pre}}(\mathrm{b0}) = \mathbb{V}\mathrm{ar}$, i.e., all variables are assumed to be initially unconstrained. After processing the assignment in b0, we have that variable $x$ is constrained, so that $\mathrm{LU}_{\mathrm{post}}(\mathrm{b0}) = \{y, z\}$; this set is propagated to the start of basic blocks b1 and b2. The abstract execution of the Boolean guard statement at the start of b1 causes variable $y$ to become constrained too; the following two assignments keep both $x$ and $y$ constrained, so that we obtain $\mathrm{LU}_{\mathrm{post}}(\mathrm{b1}) = \{z\}$. Similarly, the Boolean guard statement at the start of b2 causes variable $y$ to become constrained; however, the last assignment in b2 reinserts $y$ in the LU set, because variable $z$ is unconstrained; hence we obtain $\mathrm{LU}_{\mathrm{post}}(\mathrm{b2}) = \{y, z\}$. The LU set at the start of block b3 is computed as

$$\mathrm{LU}_{\mathrm{pre}}(\mathrm{b3}) = \mathrm{LU}_{\mathrm{post}}(\mathrm{b1}) \cup \mathrm{LU}_{\mathrm{post}}(\mathrm{b2}) = \{z\} \cup \{y, z\} = \{y, z\}.$$

Hence, after processing the assignment in b3, we obtain

$$\mathrm{LU}_{\mathrm{post}}(\mathrm{b3}) = [\![x \leftarrow x + y]\!](\{y, z\}) = \{x, y, z\}.$$

At the end of the $\mathsf{LU}^{\exists}$ analysis, the CFG transformation of Algorithm 1 is applied, producing the CFG shown in Figure 1b: here, two of the assignment statements have been replaced by nondeterministic assignments (highlighted in red).

When considering the $\mathsf{LU}^{\forall}$ heuristic variant, the analysis goes on exactly as before up to the computation of the LU set at the start of block b3: since in the universal variant the join operator is implemented as set intersection, we have

$$\mathrm{LU}_{\mathrm{pre}}(\mathrm{b3}) = \mathrm{LU}_{\mathrm{post}}(\mathrm{b1}) \cap \mathrm{LU}_{\mathrm{post}}(\mathrm{b2}) = \{z\} \cap \{y, z\} = \{z\}$$

so that, when processing the assignment in b3, we obtain

$$\mathrm{LU}_{\mathrm{post}}(\mathrm{b3}) = [\![x \leftarrow x + y]\!](\{z\}) = \{z\}.$$

Therefore, when using the universal variant, the CFG transformation step will not be able to replace the assignment in block b3, producing the more conservative CFG shown in Figure 1c.

## 3 Implementation and Experimental Evaluation

The ideas presented in the previous section have been implemented and experimentally evaluated by adapting the open source static analysis tool Clam/Crab [18]. In the Crab program representation (CrabIR), a nondeterministic assignment to a program variable `var` is encoded by the abstract statement `havoc(var)`: the variable is said to be *havocked* by the execution of this statement. By adopting the Crab terminology, we will call *havoc analyses* the heuristic pre-analyses detecting LU variables, described in Section 2.1; similarly, we will call *havoc transformation* the program transformation described in Section 2.2; and we will call *havoc processing* the combination of these two computational steps. In contrast, we will call *target analysis* the static analysis phase collecting the invariants that are of interest for the end-user.

We now describe the steps of the overall analysis process, which are summarized in Figure 2.

**Step A** The input program under analysis is parsed by Clang/LLVM, producing the corresponding LLVM bitcode representation which is then fed as input to `clam-pp`, the Clam preprocessor component. By default, `clam-pp` applies a few program transformations, such as the lowering of switch statements into chains of conditional branches; more importantly, in our experiments we systematically enabled the inlining of known function calls, so as to improve the call context sensitivity of the analysis when performing an intra-procedural analysis. Note that Clam/Crab also supports inter-procedural analyses: these are typically faster than full inlining, but quite often produce less precise results.

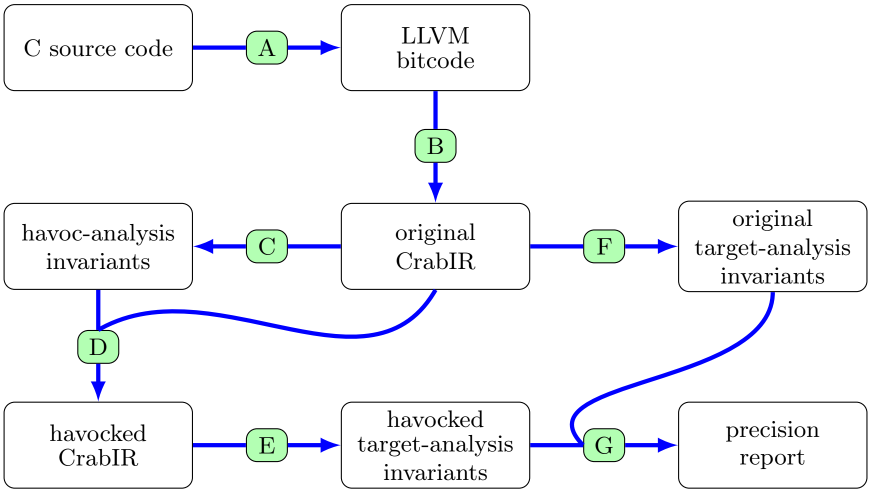

C source code
A
LLVM bitcode
B
havoc-analysis invariants
C
original CrabIR
F
original target-analysis invariants
D
havocked CrabIR
E
havocked target-analysis invariants
G
precision report

Fig. 2: Processing steps of the Clam/Crab toolchain, also including the precision comparison. Legenda: A=Clang/LLVM+clam-pp; B=Clam; C=havoc-analysis; D=havoc-propagation; E/F=target-analysis; G=clam-diff.

**Step B** The Clam component translates the LLVM bitcode representation into CrabIR, which is an intermediate representation specifically designed for static analysis; in this translation phase a few program constructs that the analysis is unable to model correctly and precisely, e.g., calls to unknown external functions, are replaced by (sequences of) `havoc` statements.

**Step C** The *havoc analysis* computes the set of program variables that are likely unconstrained at the exit of each basic block of the CrabIR representation. As discussed in Section 2, this step is not subject to a strict safety requirement and hence, in principle, its implementation could be based on any reasonable heuristics; in our prototype we decided to model it as a classical static analysis and we implemented it by using the Crab component itself.

**Step D** This step performs the *havoc transformation*, using the results of the analysis of Step C to rewrite the CrabIR representation produced by Step B; this is implemented as a simple visitor of the CrabIR CFG, corresponding to Algorithm 1, replacing assignment statements with `havoc` statements.

**Step E** The final processing step is the target static analysis, which reuses the Crab component to compute an over-approximation of the semantics of the *havocked* CrabIR representation produced by Step D, using the target abstract domain chosen at configuration time. In our experimental evaluation, we considered the classical abstract domains of intervals and convex polyhedra. The invariants computed are stored and made available to the post-analysis processing phases (assertion checks, program annotations, etc.).

**Step F** In contrast, when the havoc analysis is disabled (i.e., when the analysis toolchain is used without modification), the target static analysis is computed as described in Step E above, but starting from the *original* CrabIR representation produced by Step B.

**Step G** The loop invariants produced in Steps E and F are systematically compared for precision using the `clam-diff` tool.

| | | stmts havocked | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | non-relational | | | | relational | | | |
| test | stmts | $\mathsf{LU}^{\exists}$ | time | $\mathsf{LU}^{\forall}$ | time | $\mathsf{LU}^{\exists}_{\mathrm{rel}}$ | time | $\mathsf{LU}^{\forall}_{\mathrm{rel}}$ | time |
| all tests | 12294 | 4578 | 35 | 2248 | 51 | 264 | 42 | 263 | 65 |
| decompress | 4332 | 2757 | 13 | 472 | 19 | 185 | 17 | 184 | 22 |
| nsichneu | 3582 | 630 | 10 | 630 | 14 | 0 | 12 | 0 | 20 |
| cover | 779 | 386 | 3 | 386 | 4 | 0 | 3 | 0 | 5 |
| adpcm | 671 | 194 | 2 | 194 | 3 | 25 | 2 | 25 | 4 |
| statemate | 562 | 5 | 2 | 5 | 2 | 0 | 2 | 0 | 2 |
| ndes | 372 | 103 | 1 | 102 | 2 | 33 | 1 | 33 | 2 |
| edn | 263 | 122 | 1 | 119 | 1 | 4 | 1 | 4 | 1 |
| compress | 214 | 30 | 1 | 29 | 1 | 2 | 1 | 2 | 1 |

Table 1: WCET benchmarks: number of havocked statements by LU oracles and time spent in the havoc processing (ms). Note: details are shown only for tests having more than 200 abstract statements after inlining.

### 3.1 The impact of the havoc transformation

In our preliminary experimental evaluation, we first considered the C source files distributed with PAGAI [20], which are variants of benchmarks taken from the SNU real-time benchmark suite for WCET (worst-case execution time) analysis. When evaluating the precision of the target analyses we will focus on the invariants computed at widening points; for this reason, we discarded those few tests having no widening point at all, leaving us with 34 benchmarks. Table 1 summarizes the effects, on the CrabIR representation, of the 4 havoc transformation variants: the number of abstract statements in the original CrabIR is shown in the 2nd column; in the following 4 pairs of columns, for each variant of the havoc analysis, we show the number of assignment statements that are havocked when adopting this variant, as well as the time spent in the havoc processing steps (steps C and D of Figure 2).

According to Table 1, the overall effect of the havoc transformation varies significantly depending on the considered test; also, the choice between non-relational and relational variants of the havoc analysis seems to have a much greater impact than the choice between existential and universal variants. Some of the tests are completely unaffected by the transformation (i.e., no statement at all is havocked): this happens only 8 times for the non-relational variants (only for the smallest benchmarks), but as many as 25 times for the relational variants (also including some of the biggest benchmarks). The percentage of havocked statements on all tests is 37.2% for the most aggressive variant $\mathsf{LU}^{\exists}$, while being 2.1% for the most conservative one $\mathsf{LU}^{\forall}_{\mathrm{rel}}$. Focusing on the 8 tests reported in the table, which are those having more than 200 statements after inlining, when using $\mathsf{LU}^{\exists}$ the percentage of havocked statements ranges from 0.9% to 63.6% (median value 28.3%); in contrast, when using $\mathsf{LU}^{\forall}_{\mathrm{rel}}$ the percentage ranges from 0% to 8.9% (median value 1.2%).

| | | stmts havocked | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | non-relational | | | | relational | | | |
| test | stmts | $\mathsf{LU}^{\exists}$ | time | $\mathsf{LU}^{\forall}$ | time | $\mathsf{LU}^{\exists}_{\mathrm{rel}}$ | time | $\mathsf{LU}^{\forall}_{\mathrm{rel}}$ | time |
| all tests | 353544 | 119509 | 7014 | 104509 | 12653 | 14591 | 6771 | 14411 | 12533 |
| wl12xx | 109101 | 89543 | 332 | 75667 | 578 | 942 | 453 | 915 | 726 |
| rtlwifi | 53456 | 5873 | 1481 | 5873 | 3391 | 5813 | 1418 | 5813 | 3049 |
| w83781d | 52909 | 5452 | 2933 | 5450 | 4366 | 2605 | 2645 | 2605 | 4706 |
| brocade | 37587 | 4703 | 308 | 4495 | 666 | 2098 | 309 | 2039 | 614 |
| libfc | 31272 | 3108 | 1285 | 2811 | 2472 | 66 | 1263 | 65 | 2227 |
| vmxnet3 | 19598 | 3983 | 175 | 3849 | 311 | 496 | 184 | 495 | 327 |
| mdc | 17104 | 2333 | 154 | 2168 | 314 | 1017 | 149 | 981 | 322 |
| firewire | 13196 | 1902 | 185 | 1701 | 284 | 800 | 185 | 747 | 282 |
| solos | 12465 | 2302 | 122 | 2301 | 199 | 732 | 131 | 730 | 210 |
| abituguru | 6856 | 310 | 39 | 194 | 72 | 22 | 34 | 21 | 70 |

Table 2: SV-COMP benchmarks: number of havocked statements by LU oracles and time spent in the havocking process (ms).

Since many of considered benchmarks are synthetic ones, we extended our experimental evaluation by also considering 10 Linux drivers from the SV-COMP repository.[1] The results for these bigger benchmarks, shown in Table 2, confirm most of the observations done above, except that now the havoc transformation affects all the tests. Considering first the most aggressive variant $\mathsf{LU}^{\exists}$, the percentage of havocked statements on all tests is 33.8%, ranging from 4.5% to 82.1% (median value 13.1%); in contrast, when considering the most conservative variant $\mathsf{LU}^{\forall}_{\mathrm{rel}}$, the percentage of havocked statements on all tests is 4.1%, ranging from 0.2% to 10.9% (median value 5.2%).

In summary, on the considered benchmarks, the transformations based on the relational variants seem really conservative, whereas the non-relational ones look rather aggressive, probably leading to significant effects on the precision and efficiency of the target analysis. Regarding efficiency, it should be stressed that the current implementation of the havoc analysis and transformation steps is just a prototype, hence subject to optimizations; this holds in particular for

[1] https://github.com/sosy-lab/sv-benchmarks/c/ldv-linux-4.2-rc1

step C (havoc analysis), as only a small percentage of the havoc processing time is spent in step D (havoc transformation).

### 3.2 Precision and efficiency of the target analyses

The next step in our experiments is the evaluation of the effect of the considered program transformation on the precision and efficiency of the target analyses. To this end, we consider the classical numerical analyses based on the abstract domains of intervals [12] and convex polyhedra [14]. For the first domain we adopted the implementation that is built-in in Crab, whereas for the latter we opted for the domain of convex polyhedra provided by the PPLite library [5,6,7], which is accessible in Crab via the generic Apron interface [23]. Note that we are considering the Cartesian factored variant [19,29] of the domain of convex polyhedra, which greatly improves the efficiency of the classical polyhedral analysis by dynamically computing *optimal* variable packs, thereby incurring no precision loss; a recent experimental evaluation [2] has shown that the PPLite's implementation of this domain is competitive with the one provided by ELINA [29], which is considered state-of-the-art. We stress that the goal of our oracle-based program transformation is to obtain further, significant efficiency improvements and, to this end, a limited precision loss is an acceptable trade-off.

**Intervals** When considering a non-relational target analysis, it is natural to also consider a non-relational variant for the havoc analysis step. For the WCET benchmarks, even when using the most aggressive havoc analysis $\mathsf{LU}^{\exists}$, we obtain the same precision of the original interval analysis for all the 34 C programs of the benchmark suite; that is, we compute the same invariants on all the 288 widening points. The differences in precision for the Linux drivers are summarized in Table 3. The first (resp., second) row shows the results of the overall comparison in terms of number (resp., percentage) of widening points on which the invariant computed on the havocked CrabIR is equivalent (EQ), stronger (LT), weaker (GT) and uncomparable (UN) with respect to the invariant computed on the original CrabIR. In this case, we record precision losses for 2 tests for the $\mathsf{LU}^{\exists}$ havoc analysis and 1 test for the $\mathsf{LU}^{\forall}$ havoc analysis, whose details are shown in the next two rows of the table; the same precision is obtained on the other 8 tests, whose details are omitted. For the $\mathsf{LU}^{\exists}$ analysis, the precision loss affects 121 invariants (∼4% of all invariants computed); this number decreases to 97 invariants (∼3%) when using $\mathsf{LU}^{\forall}$. Note that, quite often, these precision losses are due to just one or two interval constraints missing from the weaker invariants: as a matter of fact, when counting the number of constraints occurring in the invariants, it can be seen that the havocked analyses are able to compute ∼ 99% of the constraints computed by the original interval analysis. This is an impressive result, when considering that it has been obtained by using rather aggressive program transformations; for instance, we obtain no precision loss on driver `wl12xx` even when havocking more than 80% of its 109K statements (see the first line in Table 2).

| | | $\mathsf{LU}^{\exists}$ *vs* original | | | | $\mathsf{LU}^{\forall}$ *vs* original | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | EQ | LT | GT | UN | EQ | LT | GT | UN |
| # INVS | 3102 | 2981 | 0 | 121 | 0 | 3005 | 0 | 97 | 0 |
| % INVS | 100.00 | 96.10 | 0 | 3.90 | 0 | 96.87 | 0 | 3.13 | 0 |
| vmxnet3 | 513 | 416 | 0 | 97 | 0 | 416 | 0 | 97 | 0 |
| **mdc** | 112 | **88** | 0 | **24** | 0 | **112** | 0 | **0** | 0 |

Table 3: SV-COMP benchmarks: invariant comparison for the interval domain. Boldface text highlights the differences between $\mathsf{LU}^{\exists}$ and $\mathsf{LU}^{\forall}$.

Regarding the impact on the efficiency of the target analysis, our experiments on the domain of intervals have shown that the havocked analysis pipeline seems unable to trigger significant efficiency improvements: no matter if considering the WCET or the SV-COMP benchmarks, the original analysis is as efficient as (sometimes even more efficient than) the havocked ones. This was somehow expected, since in our prototype we are only replacing deterministic assignment statements with nondeterministic ones and, in both cases, the abstract execution on the interval domain is very efficient; hence, any small efficiency improvement obtained is likely masked by the overheads of the modified analysis toolchain. Roughly speaking, in order to obtain a measurable effect on the precision/efficiency trade-off, we have to consider abstract domains that are computationally more expensive.

| | | $\mathsf{LU}^{\exists}$ *vs* original | | | | $\mathsf{LU}^{\forall}$ *vs* original | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | EQ | LT | GT | UN | EQ | LT | GT | UN |
| # INVS | 288 | 244 | 0 | 44 | 0 | 249 | 0 | 39 | 0 |
| % INVS | 100.00 | 84.72 | 0 | 15.28 | 0 | 86.46 | 0 | 13.54 | 0 |
| decompress | 79 | 67 | 0 | 12 | 0 | 67 | 0 | 12 | 0 |
| adpcm | 27 | 12 | 0 | 15 | 0 | 12 | 0 | 15 | 0 |
| edn | 12 | 9 | 0 | 3 | 0 | 9 | 0 | 3 | 0 |
| lms | 12 | 10 | 0 | 2 | 0 | 10 | 0 | 2 | 0 |
| ndes | 12 | 9 | 0 | 3 | 0 | 9 | 0 | 3 | 0 |
| **qsort-exam** | 6 | **1** | 0 | **5** | 0 | **6** | 0 | **0** | 0 |
| cover | 3 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 |
| insertsort | 2 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

Table 4: WCET benchmarks: invariant comparison for the polyhedra domain. Boldface text highlights the differences between $\mathsf{LU}^{\exists}$ and $\mathsf{LU}^{\forall}$ analyses.

**Convex polyhedra** When considering a relational target analysis, it would seem natural to consider the relational variants, $\mathsf{LU}^{\exists}_{\text{rel}}$ and $\mathsf{LU}^{\forall}_{\text{rel}}$, of the havoc analysis. Our first experiments, however, have shown that these variants are too conservative and hence probably unable to obtain the efficiency improvements we are looking for. Since our end goal is to obtain a practical way to effectively tune the efficiency/precision trade-off of the target analysis, we keep focusing on the non-relational variants $\mathsf{LU}^{\exists}$ and $\mathsf{LU}^{\forall}$, trading the corresponding precision losses for efficiency.

In Table 4, having the same structure of Table 3, we report the precision comparison for the WCET benchmarks. In this case, we obtain the same results, no matter if using $\mathsf{LU}^{\exists}$ or $\mathsf{LU}^{\forall}$, for 26 of the 34 tests; hence the table shows the details of the precision regressions for the remaining 8 tests. When comparing the precision of $\mathsf{LU}^{\exists}$ and $\mathsf{LU}^{\forall}$ with respect to the classification of the invariants into the EQ, LT, GT and UN categories, we observe a single difference on the `qsort-exam` test, where $\mathsf{LU}^{\forall}$ is able to maintain the same precision of the original analysis on all the 6 widening points. Note that $\mathsf{LU}^{\forall}$ obtains other precision improvements with respect to $\mathsf{LU}^{\exists}$ on test `decompress`, in terms of the number of constraints computed, but these are not sufficient to influence the invariant classification (i.e., $\mathsf{LU}^{\forall}$ obtains smaller precision losses).

| | | $\mathsf{LU}^{\exists}$ *vs* original | | | | $\mathsf{LU}^{\forall}$ *vs* original | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | EQ | LT | GT | UN | EQ | LT | GT | UN | | | |
| # INVS | 3102 | 1389 | 0 | 1713 | 0 | 1485 | 0 | 1617 | 0 | time (secs) | | speed-up |
| % INVS | 100.00 | 44.78 | 0 | 55.22 | 0 | 47.83 | 0 | 53.17 | 0 | original | $\mathsf{LU}^{\forall}$ | |
| vmxnet3 | 513 | 190 | 0 | 323 | 0 | 190 | 0 | 323 | 0 | 134.21 | 36.39 | 3.69 |
| brocade | 511 | 154 | 0 | 357 | 0 | 154 | 0 | 357 | 0 | 141.10 | 105.92 | 1.33 |
| firewire | 499 | 319 | 0 | 180 | 0 | 319 | 0 | 180 | 0 | 65.39 | 20.92 | 3.13 |
| w83781d | 356 | 95 | 0 | 261 | 0 | 95 | 0 | 261 | 0 | 110.76 | 110.47 | 1.00 |
| solos | 303 | 104 | 0 | 199 | 0 | 104 | 0 | 199 | 0 | 79.56 | 9.67 | 8.23 |
| libfc | 295 | 295 | 0 | 0 | 0 | 295 | 0 | 0 | 0 | 167.17 | 104.68 | 1.60 |
| rtlwifi | 208 | 5 | 0 | 203 | 0 | 5 | 0 | 203 | 0 | 79.86 | 68.64 | 1.16 |
| **abituguru** | 169 | **91** | 0 | **78** | 0 | **168** | 0 | **1** | 0 | 94.16 | 54.22 | 1.74 |
| wl12xx | 136 | 92 | 0 | 44 | 0 | 92 | 0 | 44 | 0 | 182.22 | 22.29 | 8.17 |
| **mdc** | 112 | **44** | 0 | **68** | 0 | **63** | 0 | **49** | 0 | 107.30 | 24.31 | 4.41 |

Table 5: SV-COMP benchmarks: invariant comparison for the polyhedra domain; Boldface text highlights the precision differences between $\mathsf{LU}^{\exists}$ and $\mathsf{LU}^{\forall}$ analyses. The last 3 columns report the efficiency comparison for $\mathsf{LU}^{\forall}$.

The precision comparison for the Linux driver benchmarks is reported in Table 5: on these bigger tests the havocked target analysis reports precision losses on all tests with the exception of `libfc`. The universal variant $\mathsf{LU}^{\forall}$ is able to significantly improve precision with respect to $\mathsf{LU}^{\exists}$, increasing the number of EQ invariants for tests `abituguru` and `mdc`; precision improvements are also obtained by $\mathsf{LU}^{\forall}$ on another three drivers (`vmxnet3`, `brocade`, `wl12xx`), but again these do not affect the invariant classification.

A more detailed analysis of the experimental results shows that, when comparing the overall number of constraints that compose the computed invariants, the havocked target analyses are able to produce approximately 90% of all the constraints that are obtained when using the original target analysis, with the $\mathsf{LU}^{\forall}$ variant scoring $\sim 1\%$ better than the $\mathsf{LU}^{\exists}$ variant.

Our efficiency comparison for the domain of convex polyhedra focuses on the havoc transformation based on the $\mathsf{LU}^{\forall}$ oracle, which as discussed above is able to obtain slightly better results with respect to the $\mathsf{LU}^{\exists}$ oracle. The results for the SV-COMP benchmarks are reported in the last three columns of Table 5: the first of these columns shows the baseline for the comparison, i.e., the time spent when the convex polyhedra analysis works on the original CrabIR; the second column shows the time spent when the target analysis works on the havocked CrabIR obtained when using the $\mathsf{LU}^{\forall}$ oracle; in the third column we report the speed-up obtained. We observe a speed-up for all the tests with the exception of `w83781d`; the positive speed-ups range from 1.16 (`rtlwifi`) to 8.23 (`solos`); the geometric mean of the speed-ups, computed on all the 10 tests, is 2.61.

We are omitting the details of a corresponding efficiency comparison for the WCET benchmark, mainly due to the synthetic nature of the tests: in practice, most of them complete the analysis immediately, making a reliable comparison almost impossible; the others are characterized by (relatively) much higher analysis times, so that they could be interpreted as outliers when compared to the first group. When restricting attention to the few WCET tests whose target analysis takes more than a second, the speed-up obtained ranges between 1.11 and 2.49 (geometric mean 1.59).

It is worth stressing that the time shown are those spent in the *overall* analysis pipeline, only excluding the post-analysis steps that are meant to store and later compare the loop invariants. Namely, with reference to Figure 2, the time spent in steps A, B and F by the original analysis is compared to the time spent in steps A, B, C, D and E by the modified pipeline. The reader interested in factoring out the time spent during the havoc processing phases (C and D) is referred to Tables 1 and 2 discussed previously.

## 4 Related Work

The four variants of LU variable analysis described in Section 2.1 and the abstract program transformation outlined in Section 2.2, as a whole, can be seen as an instance of the Abstract Compilation approach [21,31] to Abstract Interpretation [12]. A few examples[1,9,17,32] of application of Abstract Compilation have already been briefly recalled in Section 1; other examples have been recently discussed in [15]. A notable distinction between the current proposal and most of the approaches in the literature is that we do not require the program transformation to *fully* preserve the abstract semantics of the program: since our goal is to tune the efficiency/precision trade-off, we explicitly allow for (hopefully limited) precision losses in the transformation step. In our opinion, this is one of the most relevant differences between abstract and concrete (i.e., traditional) compilation.

It would be tempting to cast our pre-analyses as instances of the $\mathrm{A}^2\mathrm{I}$ framework [13] and, in particular, as examples of offline meta-abstract interpretations.

This would not be fully appropriate: as we already noted, our oracles have no formal correctness requirement (e.g., they can incur both false positives and false negatives, without affecting the correctness of the target analysis) and hence they cannot be seen as *proper* abstract interpretations. This is also the main difference between our current proposal and similar approaches that, in contrast, are firmly based on the theory of Abstract Interpretation. As notable examples we mention the research work on abstract program slicing (e.g., [22,25]) and more generally dependency analysis [11]. For instance, abstract program slicing aims at removing from the program those instructions that are definitely not affecting the end result of the analysis (possibly modulo a specific slicing criterion); hence, precision losses are explicitly forbidden, while they are legitimate when using our havoc transformation.

The idea to use heuristic approaches to tune the precision/efficiency trade-off of a static analysis is clearly not new to this paper. For the case of numeric properties, the most known example is probably the variable packing technique adopted by Astrée [8, Section 7.2.1]: here, a pre-analysis phase based on a syntactic heuristics, with no formal correctness requirement as in our case, computes for each portion of the program some relatively small variable packs, which are later used during the proper analysis phase to enable the precision of a relational analysis (in that case, the abstract domain of octagons) while keeping under control its computational cost. Similarly, [28] proposes a pre-analysis phase to estimate the impact on precision of context-sensitivity for an inter-procedural program analysis, so as to enable it (and the corresponding computational costs) only when a precision improvement is likely obtained. The overall approach has been extended to non-numeric properties: for instance, [30] and [24] propose two lightweight pre-analyses that can improve the precision of pointer analyses by driving them to dynamically switch between different levels of (context, object, type) sensitivity.

## 5 Conclusion

In this paper we have proposed a program transformation that improves the efficiency of a classical numeric static analysis by trading some of its precision: this works by selectively havocking some of the abstract assignments in the program, so as to forget all information on the assigned variable. The havocking process is guided by an oracle, whose goal is to predict whether or not the assigned variable would likely be unconstrained anyway, thereby limiting the precision loss. A main contribution of this paper is the design and experimental evaluation of four variants of a lightweight dataflow analysis that implement reasonably precise oracles. The precision and efficiency of the target static analysis using the program transformation as a preprocessing step have been evaluated on two benchmark suites, identifying the universal non-relational oracle as the most promising one. When the havoc transformation is guided by this oracle, the static analysis based on the domain of convex polyhedra is able to achieve significant efficiency improvements, while facing limited precision losses.

Other aspects of the proposed technique will be investigated in future work. In the first place, the current oracle can be refined by applying other heuristics or machine learning techniques, so as to further mitigate the precision losses in the target analysis while maintaining an aggressive transformation leading to efficiency gains. Secondly, the program transformation itself can be enhanced, by considering the propagation of unknown information to other kinds of abstract statements and the interaction of the havocking process with other program transformations, such as abstract dead code elimination and CFG simplifications. Also, the analysis and program transformation can be extended to be integrated in source-level program analysis tools, such as MOPSA [26], so as to investigate its effectiveness when going beyond 3-address code syntax and considering more general arithmetic expressions. Finally, it would be interesting to evaluate the applicability of the approach to non-numerical analyses, such as string analyses [3,4].

**Acknowledgements** The authors would like to thank the anonymous reviewers for their useful comments and suggestions. This work was partially supported by Bando di Ateneo per la ricerca 2022, founded by University of Parma, project number: MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN.

## References

1. Amato, G., Spoto, F.: Abstract compilation for sharing analysis. In: Kuchen, H., Ueda, K. (eds.) Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2024, pp. 311–325. Springer (2001). https://doi.org/10.1007/3-540-44716-4_20
2. Arceri, V., Dolcetti, G., Zaffanella, E.: Speeding up static analysis with the split operator. In: Ferrara, P., Hadarean, L. (eds.) Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023. pp. 14–19. ACM (2023). https://doi.org/10.1145/3589250.3596141
3. Arceri, V., Mastroeni, I.: Analyzing dynamic code: A sound abstract interpreter for *Evil* eval. ACM Trans. Priv. Secur. **24**(2), 10:1–10:38 (2021). https://doi.org/10.1145/3426470
4. Arceri, V., Olliaro, M., Cortesi, A., Ferrara, P.: Relational string abstract domains. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 20–42. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_2
5. Becchi, A., Zaffanella, E.: A direct encoding for NNC polyhedra. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 230–248. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_13
6. Becchi, A., Zaffanella, E.: An efficient abstract domain for not necessarily closed polyhedra. In: Podelski, A. (ed.) Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings. Lecture Notes in

Computer Science, vol. 11002, pp. 146–165. Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_11

7. Becchi, A., Zaffanella, E.: PPLite: Zero-overhead encoding of NNC polyhedra. Inf. Comput. **275**, 104620 (2020). https://doi.org/10.1016/j.ic.2020.104620
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Cytron, R., Gupta, R. (eds.) Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003. pp. 196–207. ACM (2003). https://doi.org/10.1145/781131.781153
9. Boucher, D., Feeley, M.: Abstract compilation: A new implementation paradigm for static analysis. In: Gyimóthy, T. (ed.) Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1060, pp. 192–207. Springer (1996). https://doi.org/10.1007/3-540-61053-7_62
10. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: A framework for static analysis based on abstract interpretation. In: Giannakopoulou, D., Salaün, G. (eds.) Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8702, pp. 271–277. Springer (2014). https://doi.org/10.1007/978-3-319-10431-7_20
11. Cousot, P.: Abstract semantic dependency. In: Chang, B.E. (ed.) Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11822, pp. 389–410. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_19
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
13. Cousot, P., Giacobazzi, R., Ranzato, F.: $A^2I$: abstract$^2$ interpretation. Proc. ACM Program. Lang. **3**(POPL), 42:1–42:31 (2019). https://doi.org/10.1145/3290355
14. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770
15. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and transformation of constrained horn clauses for program verification. Theory Pract. Log. Program. **22**(6), 974–1042 (2022). https://doi.org/10.1017/S1471068421000211
16. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Do, L.N.Q., Urban, C. (eds.) SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021. pp. 1–6. ACM (2021). https://doi.org/10.1145/3460946.3464316
17. Giacobazzi, R., Debray, S.K., Levi, G.: Generalized semantics and abstract interpretation for constraint logic programs. J. Log. Program. **25**(3), 191–247 (1995). https://doi.org/10.1016/0743-1066(95)00038-0
18. Gurfinkel, A., Navas, J.A.: Abstract interpretation of LLVM with a region-based memory model. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Software Verification - 13th International Conference, VSTTE 2021, New Haven,

CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13124, pp. 122–144. Springer (2021). https://doi.org/10.1007/978-3-030-95561-8_8

19. Halbwachs, N., Merchat, D., Gonnord, L.: Some ways to reduce the space dimension in polyhedra computations. Formal Methods Syst. Des. **29**(1), 79–95 (2006). https://doi.org/10.1007/s10703-006-0013-2
20. Henry, J., Monniaux, D., Moy, M.: PAGAI: A path sensitive static analyser. In: Jeannet, B. (ed.) Third Workshop on Tools for Automatic Program Analysis, TAPAS 2012, Deauville, France, September 14, 2012. Electronic Notes in Theoretical Computer Science, vol. 289, pp. 15–25. Elsevier (2012). https://doi.org/10.1016/j.entcs.2012.11.003
21. Hermenegildo, M.V., Warren, R.A., Debray, S.K.: Global flow analysis as a practical compilation tool. J. Log. Program. **13**(4), 349–366 (1992). https://doi.org/10.1016/0743-1066(92)90053-6
22. Hong, H.S., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary. pp. 25–34. IEEE Computer Society (2005). https://doi.org/10.1109/SCAM.2005.2
23. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52
24. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: A principled approach to selective context sensitivity for pointer analysis. ACM Trans. Program. Lang. Syst. **42**(2), 10:1–10:40 (2020). https://doi.org/10.1145/3381915
25. Mastroeni, I., Zanardini, D.: Abstract program slicing: An abstract interpretation-based approach to program slicing. ACM Trans. Comput. Log. **18**(1), 7:1–7:58 (2017). https://doi.org/10.1145/3029052
26. Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native C extensions. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12913, pp. 323–345. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_16
27. Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: LiSA: A generic framework for multilanguage static analysis. In: Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (eds.) Challenges of Software Verification, pp. 19–42. Springer Nature Singapore, Singapore (2023). https://doi.org/10.1007/978-981-19-9601-6_2
28. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective x-sensitive analysis guided by impact pre-analysis. ACM Trans. Program. Lang. Syst. **38**(2), 6:1–6:45 (2016). https://doi.org/10.1145/2821504
29. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 46–59. ACM (2017). https://doi.org/10.1145/3009837.3009885
30. Tan, T., Li, Y., Xue, J.: Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and

Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 278–291. ACM (2017). https://doi.org/10.1145/3062341.3062360

31. Warren, R.A., Hermenegildo, M.V., Debray, S.K.: On the practicality of global flow analysis of logic programs. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes). pp. 684–699. MIT Press (1988)
32. Wei, G., Chen, Y., Rompf, T.: Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. Proc. ACM Program. Lang. **3**(OOPSLA), 126:1–126:32 (2019). https://doi.org/10.1145/3360552