Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

Flow Insensitive Relational Static Analysis

Solène Mirliaz[1] and David Pichardie[1]

Univ Rennes, Inria, CNRS, IRISA

Abstract. Relational static analyses allow to keep track of the relations between variables in a program. They rely on relational abstract domains like Octagon or Polyhedra. While expressive, these analyses are often costly. To reduce these computations, we design a flow insensitive static analysis that provides a single relational invariant for a whole program. A specific representation of the program, namely the *Static Single Information* (SSI) form, allows us to preserve a good precision (compare to a flow-sensitive analysis) thanks to the indexing of the variables. We report on the design and the soudness of the analysis, using an abstract interpretation methodology. A prototype has been developped to test the usefulness of the analysis on small programs.

Abstract Interpretation provides expressive symbolic techniques that are able to automatically infer relations between program variables. They rely on relational abstract domain and generally attach one (flow sensitive) abstract element to every program point of a program. At an other side of the spectrum, flow-insensitive analyses [9] (such as Andersen's pointer analysis [2]) compute one global invariant for the whole program, sparing time and memory. Our goal is to design an analysis with *one* relational invariant for the whole program, that remains precise, using an intermediate representation borrowed from advanced compiler techniques: the Single Static Information (SSI) form. This representation increases the number of variables but partitions their lifetime in a way that benefits to flow-insensitive analyses. Indeed, this splitting guarantees that the abstract state of a variable is invariant at each program point where the variable is alive. It has been explored for non-relational or semi-relational [8] analyses. This paper is the first to explore its usefulness for relational domains.

Outline We starts with the formal definition of the SSI form of a program as well as the concrete semantics we consider (Section 1). This semantics is a set of partial functions representing the environments. We then explain the need to track precisely the domain of definitions of environments. Then in Section 3, the abstract semantics of the program can be defined, using the principles of abstract interpretation (the abstract domain is defined in Section 2). This abstract semantics consists in a single invariant that can be concretized as an overapproximation of the set of environments of the concrete semantics. Finally, we present a prototype implementation and its preliminary results in Section 4. This paper is a short version of a full companion report [7].

Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

1 SSI Form and Concrete Semantics

Our analysis is performed on the SSI form of programs. The Static Single Information form (SSI) [10], is an intermediate representation of programs. It extends the SSA form to guarantee that any property infered on a variable holds from its assignment (unique due to the SSA property) to its last usage. Control-flow junction points are handled with a classic SSA ϕ operator [1], but branching points may also receive a new σ information. When a variable is used in a condition, then after this condition has been passed, we acquire new information on the variable, and to guarantee the single information property we introduce a new variable, which is a copy of the original one, that passes the test. Consider a branching with tests $j < 10$ and $j \geq 10$, then two new variables are introduced such that $j_1 < 10$ and $j_2 \geq 10$. This copy are assigned through a new intruction σ, placed before the conditions: $j_1, j_2 \overset{\sigma}{\leftarrow} j$. The variable j_1 is assigned only if the test $j_1 < 10$ holds after the copy. This ensures that the property $j_1 < 10$ holds starting at the assignment. An example of a program in SSI form can be found in Figure 1. Assignments and assumes are attached to edges while $\phi-$copies and $\sigma-$copies are attached to nodes.

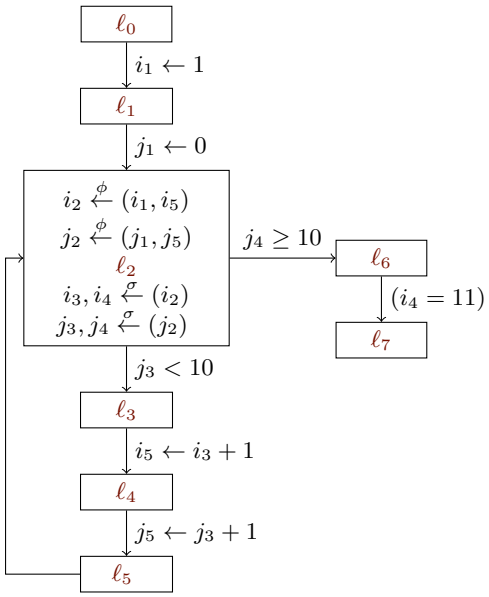

Fig. 1: SSI program example

Program representation The SSI intermediate representation of the program is a tuple $P = (E, A, J, B)$ such that: E is the set of edges $(\ell_1 \to \ell_2)$ in the CFG

Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

(ℓ_1 and ℓ_2 are labels in $\mathcal{L}$); $A \in E \to$ Atom associates to each edge $\ell_1 \to \ell_2$ its atomic instruction given by the CFG; J (*join*) is a map from a label to the set of ϕ-functions defined at this label; B (*branch*) is the equivalent map for σ-functions. The set of programs is noted $\mathbb{P}$. Let $x \overset{\phi}{\leftarrow} (\ell_1 : x_1, \ldots, \ell_k : x_k)$ be a ϕ-function attached to ℓ. The term $J(\ell)(x)(\ell_1) = x_1$ can be read as "At ℓ, when coming from ℓ_1, the program assign x_1 to x". A σ-function is represented the same way. If $(\ell_1 : x_1, \ldots, \ell_k : x_k) \overset{\sigma}{\leftarrow} x$ is attached to ℓ, then the term $B(\ell)(x)(\ell_1) = x_1$ can be read as "At ℓ, when going to ℓ_1, the program assign x to x_1".

Single entry point We consider that the CFG (and so the SSI form) has only *one* entry point ℓ_0 and that this entry point has no predecessors.

Concrete collecting semantics Programs manipulate *variables* (from set $\mathbb{V}$) whose values are stored in *environments* (from set $\mathcal{E}$). The concrete semantics of a program P is given by the function $[\![P]\!] \in \mathcal{L} \to \mathcal{P}(\mathcal{E})$ that attaches to each CFG label the corresponding set of reachable environments. It is defined as the least fixed point of a *global transfer function* F which applies the transfer function of each edge of the CFG, then joins the results.

$$[\![P]\!] \overset{\text{def}}{=} \text{lfp}(F)$$

We will first detail the transfer function of an edge then detail F.

Let ℓ be a program point (different from ℓ_0). An enviroment s is reachable at ℓ if there exists an environment s' at a predecessor ℓ' of ℓ, such that s is s' on which was applied the following transformations (in order): (i) the potential σ-copies from ℓ' to ℓ, (ii) the atomic instruction from ℓ' to ℓ, and finally (iii) the potential ϕ-copies at ℓ, coming from ℓ'. We note $T_{\ell' \to \ell} : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ the corresponding edge transfer function (details in [7]).

Let $S \in \mathcal{L} \to \mathcal{P}(\mathcal{E})$ be the current map of set of reachable environments. The global transfer function F computes the reachable environments at label ℓ by merging all edge transfer functions applied to predecessor sets.

$$F(S) \overset{\text{def}}{=} \ell \mapsto \begin{cases} S_{\ell_0} & \text{if } \ell = \ell_0 \\ \bigcup\limits_{(\ell' \to \ell) \in E} T_{\ell' \to \ell}(S_{\ell'}) & \text{otherwise} \end{cases}$$

In the end, the semantics of the SSI program is the fixpoint of F.

Domain coherence As a result from this union, we have a set of partial functions that may not have the same domain (some variables have been defined on an edge and not on the others). In our example, the joining point ℓ_2 is associated with the union of the environments from ℓ_1 and the ones from ℓ_5. The environments from ℓ_1, after applying the transfer function from ℓ_1 to ℓ_2, have as a domain $\{i_1, j_1\}$. On the other hand, the environments of ℓ_5 have as a domain $\{i_1, j_1, i_2, j_2, i_3, j_3, i_5, j_5\}$. So the environments at ℓ_2 can have different domains.

Accepted for NSAD 2019 informal proceedings (https://staticanalysis.org/nsad2019/)

2 Abstract domain

With one invariant for the whole program, our analysis is control flow-insensitive and cannot determine if a variable may or may not be defined. As we update this invariant, we need to determine if a variable is newly introduced or if the property computed must be join with the one already present in the invariant. For that, we need to represent, in the abstract domain, partial functions, which is not possible in state of the art abstract domains [5]. There, the abstract domains concretize into *total* functions (or at least partial functions with fixed common domain).

To achieve our goal, we build an overlay around such abstract domain $\mathcal{D}$, which can be any numerical abstract domain. This overlay keeps track of which variables may be defined, and which must not. It is equipped with abstract union $\cup^\sharp$, intersection $\cap^\sharp$, monotone concretization $\gamma_{\mathcal{D}^+} \in \mathcal{D}^+ \to \mathcal{P}(\mathcal{E})$ and sound abstractions of atomic instructions (details in [7]).

To compare our abstract semantics to the concrete one, we need to transform our global invariant, which represent a set of environments in $\mathcal{P}(\mathcal{E})$, into a set of reachable states in $\mathcal{P}(\mathcal{L} \times \mathcal{E})$ (as defined in the concrete semantics). This is formalized with the concretization $\gamma : \mathcal{D}^+ \to \mathcal{P}(\mathcal{L} \times \mathcal{E})$ defined as follow:

$$\gamma(X^+) = \{(\ell, s) | \ell \in \mathcal{L}, s \in \gamma_{\mathcal{D}^+}(X^+)\}$$

Associating the complete concretization to each program point is a significant overapproximation but it is compensated by SSI intrinsic properties that restrict strongly the set of variables that may have been defined at each label.

3 Abstract semantics

We define the abstract semantics as the least fixpoint of the following global abstract function $F^\sharp$

$$F^\sharp \stackrel{\text{def}}{=} F_0^\sharp \circ \bigcirc_{(\ell \to \ell') \in E} F_{\ell \to \ell'}^\sharp$$

with $F_{\ell \to \ell'}^\sharp(X^+) \stackrel{\text{def}}{=} X^+ \cup^\sharp T_{\ell \to \ell'}^\sharp(X^+)$ and $F_0^\sharp(X^+) \stackrel{\text{def}}{=} X_0^+ \cup^\sharp X^+$.

Here, X_0^+ is the abstract environment for the entry point, $(S_0 \subseteq \gamma_{\mathcal{D}^+}(X_0^+))$ and $T_{\ell \to \ell'}^\sharp$ is the abstract semantics of edges $(T_{\ell \to \ell'} \circ \gamma_{\mathcal{D}^+} \subseteq \gamma_{\mathcal{D}^+} \circ T_{\ell \to \ell'}^\sharp)$. Following the design of the collecting semantics, each $T_{\ell \to \ell'}^\sharp$ is the composition of (i) abstract parallel copies for σ-copie, (i) abstract atomic instruction between ℓ and ℓ', (iii) abstract parallel copies for ϕ-copies (details in [7]).

Theorem 1. *$F^\sharp$ is sound with respect to F and γ, that is to say $F \circ \gamma \subseteq \gamma \circ F^\sharp$.*

4 Implementation

We developped an OCaml prototype for this analysis. It takes as input a program in a While language with assertions and the relational abstract domain to use (any Apron domain [5]).

The output is the global invariant of the program. The analysis is iterative and does not implement (yet) acceleration techniques. To increase precision (and to delay unions), we build a single transfer function for each *block* (sequence of connected edges without branching).

We did some experiments on small programs and compare them with Interproc[1] flow-sensitive analyzer. We observe similar precision, except for some programs where the lack of widening prevents us from terminating the fixpoint computation.

5 Related work

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

Global Value Numbering (GVN) is an example of such analysis. The goal of GVN is to detect when two computations have equivalent results. The result of this analysis can be used for optimizations: instead of computing again the result, it can be copied. In 1988 Alpern et al. [1] proposed an algorithm for GVN. Their main contribution was to detect equivalence despite control structures such as conditionals. Their analysis requires the SSA form and builds a *value graph* based on this form. This graph represent the symbolic computation made at the assignments. The analysis compares the *congruence* of nodes in this graph to determine the equivalence of the computation. This analysis is flow-insensitive, and uses special nodes in this value graph to represent a conditional computation, due to a branching in the control-flow graph.

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

This two examples do not use the concepts of abstract interpretation, but they show that with an appropriate intermediate form, one can build a flow-insensitive invariant. Also these two examples are relational analyses which encouraged us in the possibility of building such analysis.

An example of relational analyzer using abstract interpretation is PAGAI [4]. Although it is not sparse, it lightens the cost of the analysis by computing the invariants at some program points only. These program points include the loop heads, the assertions and user-defined program points. Although it is not flow-insensitive, it is efficient. Several optimizations developped for this tool can be

[1] `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

applied to our analysis, for instance to reduce the number of variables actually manipulated in the invariant.

Our abstract domain that represent partial environments can be compared to the Maya domain [6]. In this domain, a variable v will be divided into two variables v_1 and v_2. Each variable has its own constraints. If the variable v is never defined, the constraints on v_1 and v_2 are chosen such that their conjunction applied on v is impossible: for instance $v_1 < 0$ and $v_2 > 0$. This can express interesting relations that our domain cannot but it not clear yet if our analysis would benefit from it.

6 Conclusion

Our goal is to design a static analysis that combines efficiency – the analysis must be sparse with only one invariant for the whole program – with precision – it must keep properties between several variables, not just properties on single variables. Abstract interpretation theory designs analyses with one invariant per program point, but sparser relational analysis are possible, as shown with the tool PAGAI [4]. This analyzer computes invariant per assertion points (chosen by the end user) and at the loop head. We want to reach higher sparsity with a unique invariant.

To compute a unique invariant, while keeping precision, we used or designed several elements for the analysis. First we used the SSI form [10] that allows us to differenciate versions of the variables. It is central for the flow-insensitivity. Then the main element that we present in this paper is an overlay we designed for abstract domains that can represent partial functions.

This preliminary works provides encouraging results for designing a sparse relational static analysis. It should now be compared experimentally with existing analyzers such as PAGAI, both in term of efficiency and precision. On a more theoretical exploration, we would like to reason on SSI intrinsic properties to formalize theorems about the precision of the analysis, compare to a flow-sensitive version on a non-SSI program.

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. Henry, J., Monniaux, D., Moy, M.: PAGAI: A path sensitive static analyser. Electr. Notes Theor. Comput. Sci. **289** (2012)
5. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV. vol. 5643. Springer (2009)
6. Liu, J., Rival, X.: Abstraction of optional numerical values. In: APLAS. pp. 146–166 (2015)

7. Mirliaz, S., Pichardie, D.: Flow insensitive relational static analysis, http://perso.eleves.ens-rennes.fr/people/solene.mirliaz/docs/FlowInsRel-MirliazPichardie-19.pdf, complete report
8. 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)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Berlin Heidelberg (2004), https://books.google.fr/books?id=RLjt0xSj8DcC
10. Pereira, F., Rastello, F.: Static Single Information form (2018), http://ssabook.gforge.inria.fr/latest/book.pdf, chapter 11 in the SSA-book