SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Combatting the Precision Loss of Partial Contexts in Abstract Interpretation

Felix Sebastian Krayer

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Combatting the Precision Loss of Partial Contexts in Abstract Interpretation

Bekämpfung des Präzisionsverlust durch partielle Kontexte in Abstrakter Interpretation

Author:	Felix Sebastian Krayer
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Michael Schwarz
Submission Date:	15th of February 2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15th of February 2023 Felix Sebastian Krayer

Acknowledgments

Abstract

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background **2**
2.1 Static Analysis 2
2.1.1 Flow sensitive analysis 3
2.1.2 Constraint systems 3
2.1.3 Interprocedural analysis 4
2.1.4 Context sensitivity 5
2.1.5 Partial context sensitivity 6
2.1.6 Precision loss 6

3 Combatting Precision Loss **7**
3.1 Formal description 7
3.1.1 Taint analysis 7
3.1.2 Improving the values-of-variables analysis 8
3.2 Implementation 8

4 Evaluation **10**
4.1 Testing 10
4.2 Benchmarking 10

5 Conclusion **11**

Abbreviations **12**

List of Figures **13**

List of Tables **14**

Bibliography **15**

1 Introduction

Related work

Structure First we will introduce the basics of static analysis. This will go by introducing constraint systems and how these are used to gain information about the program statically. It will be accompanied by an example of a value-of-variables analysis acting on a toy language we will use for examples in this thesis. This will be extended to an interprocedural approach where partial context sensitivity will be introduced. Here the source of the precision loss will be pointed out. We then will propose an approach to combat this precision loss. The approach will first be introduced theoretically, after which we also present the challenges and results of implementing it in the Goblint analyzer. To give an evaluation to the proposed approach, a benchmark of the implementation will be performed and inspected. Our conclusions are presented in the last chapter.

2 Background

2.1 Static Analysis

Static analysis is defined by Rival [RY20] as "[...]an automatic technique that approximates in a conservative manner semantic properties of programs before their execution". This means that the program is analyzed just by the given source code without execution. The goal is to prove certain properties about the program in a "sound" manner i.e. any property that is proven to hold actually does hold. However, from failing to prove a property one cannot conclude that the given property does not hold.
In order to prove properties, e.g. finding that a program does not contain races or identifying dead code, we need to gain information about the program. This is done by performing various kinds of analyses. We will focus on flow sensitive analyses from now on i.e. analyses which find properties of the program dependent on the location within it. The semantic of these will be introduced in the following chapters.

// TODO: talk about Abstract Interpretation.

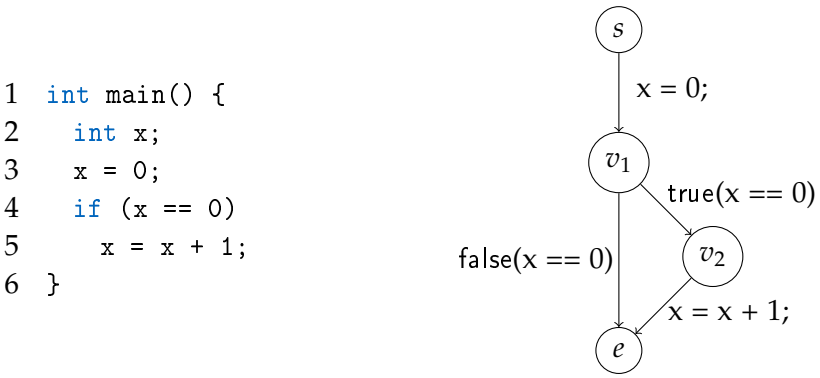

Figure 2.1: Example program (left) and corresponding CFG (right)

2.1.1 Flow sensitive analysis

As noted above flow sensitive analyses find properties of the program dependent on the point within the program. Expressed differently this means a flow sensitive analysis will find an overapproximation of states the program may be in for any given point within the program or "program point". This state can describe many things dependent on the analysis performed.
First let us define what a program point is: Imagine a Control flow Graph (CFG), where nodes represent points between instructions within the program. Edges are labeled with instructions or checks (from now on collectively called "actions") and describe the transitions between these points (see example Figure 2.1). Then any node on this CFG would be what we call a program point.
Concretely let N be the set of all program points. Furthermore, let $\mathbb{D}$ be a Domain containing abstract states describing concrete states of the program. This means that some $d \in \mathbb{D}$ can describe many states the program can be in.
Then an analysis is expected to find a mapping $\eta : N \to \mathbb{D}$ which maps program points to abstract states describing that location within the program i.e. for $[n] \in N$, $\eta\ [n]$ should be an abstract state describing all possible states (and possibly more) the program can be in at program point $[n]$.

As an example we will introduce a values-of-variables analysis for integers. This analysis finds a mapping from a set of variables X to abstractions of their possible values at any given program point. In the scope of this thesis we will focus on abstracting integer values by sets of integers. Thereby the goal of our values-of-variables analysis is to find a mapping $X \to 2^{\mathbb{N}}$ for each program point.
Combining this with the semantic of flow sensitive analysis from before, we get that the Domain $\mathbb{D}_\mathsf{v}$ for the values-of-variables analysis should be $\mathbb{D}_\mathsf{v} = X \to 2^{\mathbb{N}}$. Finally, the resulting $\eta_\mathsf{v} : N \to \mathbb{D}_\mathsf{v}$ for this analysis describes a mapping $\eta_\mathsf{v}\ [n]$ for some program point $[n] \in N$, where $\eta_\mathsf{v}\ [n]\ x$ is a set containing all values $x \in X$ may possibly hold at $[n]$. From this we can conclude that x cannot hold any value outside $\eta_\mathsf{v}\ [n]\ x$ at program point $[n]$.

2.1.2 Constraint systems

We now formulate a way in which we can describe an analysis in the form of constraints. For this we need a partial ordering $\sqsupseteq$ on the domain $\mathbb{D}$.
Then we create a system of constraints which can be solved for a solution. Consider the edges (u, A, v) of the CFG, where each edge denotes a transition from program point $[u]$ to program point $[v]$ via the action A. Now let each of these edges give raise to a

constraint

$$\eta\ [v] \sqsupseteq [\![A]\!]^{\#}\ (\eta\ [u])$$

where $[\![A]\!]^{\#}$ denotes the abstract effect of the action A defining our analysis. In addition, we need a start state. This is given by $\mathsf{init}^{\#} : \mathbb{D}$ which is defined depending on the analysis. This gives raise to the start constraint $\eta\ [s] \sqsupseteq \mathsf{init}^{\#}$.

We will show these ideas with our example of the values-of-variables analysis: Let us define the partial ordering $\sqsupseteq_{\mathsf{v}}$ that will be used in the constraints. We will do this by saying that a mapping $M_1 \in \mathbb{D}_{\mathsf{v}}$ is ordered above another mapping M_2 iff for every variable the set it is mapped to in M_1 is a superset of the one the variable is mapped to in M_2. Formulated formally this is:

$$M_1, M_2 \in \mathbb{D}_{\mathsf{v}} : M_1 \sqsupseteq_{\mathsf{v}} M_2 \iff \forall x \in X : M_1\ x \supseteq M_2\ x$$

Next we define the start state $\mathsf{init}^{\#} = M_{\top}$ for this domain as the mapping that maps every variable to the full set of integers $\mathbb{N}$ i.e. $\forall x \in X : M_{\top}\ x = \mathbb{N}$. This is because we assume variables to be randomly initialized in our toy language.
It remains to define the abstract effect of actions $[\![A]\!]^{\#}_{\mathsf{v}}$ for our values-of-variables analysis. We will just show the effect of a simple variable assignment:

$$[\![x = y;]\!]^{\#}_{\mathsf{v}}\ M = M \oplus \{x \mapsto (M\ y)\}$$

where $M \oplus \{x \mapsto s\}$ denotes that the mapping M is updated such that x will be mapped to the set s. The full definition of abstract effects of a values-of-variables analysis can be found at <// TODO>.

2.1.3 Interprocedural analysis

So far we only have defined how a program without procedure calls is analyzed. Now we want to introduce procedure calls of the form $f()$. Since a call has its own set of local variables to work with and a call stack can contain multiple of the same procedure (e.g. for recursion), we will analyze procedures in their own environment. However, we need to also consider global variables that may be affected by the analysis. The idea is to give procedures their own starting states and analyze them similarly as we have done before. The final state of the called procedure is then used to be combined back into the state of the caller before the call. Formalized for an edge $(u, f();, v)$ this looks as follows:

$$\eta\ [s_f] \sqsupseteq \mathsf{enter}^{\#}\ (\eta\ [u])$$

$$\eta\ [v] \sqsupseteq \mathsf{combine}^{\#}\ ((\eta\ [u]), (\eta\ [e_f]))$$

where $[s_f]$ and $[e_f]$ are the start and end node of the CFG for procedure $f()$. The functions $\mathsf{combine}^\# : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ and $\mathsf{enter}^\# : \mathbb{D} \to \mathbb{D}$ are defined by the analysis. $\mathsf{enter}^\#$ handles computing the start state or "calling context" for $f()$, while $\mathsf{combine}^\#$ describes in what way the caller state and the end state of the callee are merged after the call.
It is worth mentioning at this point that even though a procedure can be called from multiple points within the program we still only analyze the procedure once. For n procedure calls $(u_n, f();, v_n)$ we get n constraints for $[s_f]$: $\eta\ [s_f] \sqsupseteq \mathsf{enter}^\#\ (\eta\ [u_n])$. We can express this differently in a single constraint as follows:

$$\bigsqcup \{\mathsf{enter}^\#\ (\eta\ [u_n]) | \forall [u_n]\}$$

where $\bigsqcup$ is the least upper bound i.e. the least $d \in \mathbb{D}$ according to the ordering $\sqsupseteq$ that is ordered above all of its argument elements.
This behavior of only analyzing the procedures once with the least upper bound of all calling contexts is what we call "context insensitivity". It is quite efficient but at the same time not very precise.

For our values-of-variables analysis we will show how $\mathsf{enter}_\mathsf{v}^\#$ and $\mathsf{combine}_\mathsf{v}^\#$ are defined. We need to take global variables into account when computing the calling context and combining after the call. Therefore, we define the two functions as follows:

$$\mathsf{enter}_\mathsf{v}^\#\ M = M|_{Globals} \oplus \{x \mapsto \mathbb{N} | \forall x \in X\}|_{Locals_\mathsf{ce}}$$

$$\mathsf{combine}_\mathsf{v}^\#\ (M_\mathsf{cr}, M_\mathsf{ce}) = M_\mathsf{cr}|_{Locals_\mathsf{cr}} \oplus M_\mathsf{ce}|_{Globals}$$

where $M|_{Locals}$ and $M|_{Globals}$ refers to the mapping M restricted to only the local or global variables respectively. Note that $Locals_\mathsf{ce}$ refers to the locals of the callee while $Locals_\mathsf{cr}$ refers to the locals of the caller.
To explain these two function let us first look at $\mathsf{enter}_\mathsf{v}^\#$. This function takes the part of the mapping from the caller that contains information about global variables and adds the information of uninitialized local variables used in the procedure to the state. For $\mathsf{combine}_\mathsf{v}^\#$ the local part from the callee is kept, but it is updated with the global part of the callee return state as this contains the updated information about global variables after the procedure call.

2.1.4 Context sensitivity

The context insensitive approach in the previous chapter is not very precise. Imagine a procedure that simply increments a global variable a by one. Whenever this procedure is called, our values-of-variables

2.1.5 Partial context sensitivity

2.1.6 Precision loss

3 Combatting Precision Loss

In this chapter we will describe our approach to reduce the precision loss described in Subsection 2.1.6. We will first use the syntax for flow sensitive analyses from Chapter 2 to formally define the idea. After that we explain the concrete implementation of the approach into the GOBLINT analyzer.

3.1 Formal description

3.1.1 Taint analysis

The basic idea to combat the precison loss is to track for each procedure which variables have been written or have possibly been altered in some other way. This information is then used in the values-of-variables analysis when combining the abstract state from the caller with the abstract return state given by the callee at the end of the procedure. In the following we will call a variable that has been written or altered in the current procedure context "tainted". Therefore, we introduce a new taint analysis tracking which variables have been tainted within the context of the current procedure. It is worth mentioning that our notion of taintedness is related but different from other uses of this concept.

Let us now formulate the syntax for our taint analysis: Since we want to find a collection of tainted variables per program point, a suitable domain for this analysis is the powerset of the set of variables X ordered by the subset relation:

$$\mathbb{D}_t = 2^X \text{ with } \sqsupseteq_t = \supseteq$$

From that follows that we seek to compute a mapping from program points to sets of variables i.e. $\eta_t : N \to \mathbb{D}_t$. To interpret this with the goal of our taint analysis in mind, we note that $\eta_t[n] = T$ will denote that T is the set of possibly tainted variables at program point $[n]$. Expressed differently this means that for any variable $x \in T$ we cannot exclude that this variable was altered between the start of the procedure $[n]$ is in up until the program pint $[n]$.
It remains to define $\mathsf{init}^\#$, $\mathsf{enter}^\#$ and $\mathsf{combine}^\#$ as well as the abstract effects of actions $[\![A]\!]^\#$. Recall that the notion of a "tainted" variable is defined in relation to the current

procedure. This means we want to start fresh whenever we enter a procedure and start without any variable being initially tainted. Since the same holds for the initial state we have

$$\mathsf{enter}^{\#}\ T = \mathsf{init}^{\#} = \emptyset$$

When combining the caller state with the returned callee state, we note that anything that we need to keep the tainted set from before the call, as a tainted variable can get never get "untainted" again, no matter what the procedure does. In addition to that we will add the set returned by the callee, as anything tainted in the call needs to be considered tainted in the caller as well. This is because we want to know which variables have been altered in a procedure call, no matter if the tainting happened within the procedure itself or within a procedure called by the procedure. This leaves us with the following equation for the $\mathsf{combine}^{\#}$ function:

$$\mathsf{combine}^{\#}\ (T_{\mathsf{cr}}, T_{\mathsf{ce}}) = T_{\mathsf{cr}} \cup (T_{\mathsf{ce}} \backslash \mathit{Locals}_{\mathsf{ce}})$$

Note that we removed the callee local variables $\mathit{Locals}_{\mathsf{ce}}$ because these are not accessible by the caller and all of its callers anyway, so it is not useful to keep track of them. It is also worth pointing out that the function $\mathsf{enter}^{\#}T$ is always equal to the empty set irregardles of its argument T. Therefore it computes the same function context for each call of a certain procedure making our taint analysis inherently context insensitive.

// **WIP:**

Each edge $e = (u, A, v)$ introduces the constraint $\eta_{\mathsf{t}}\ [v] \sqsupseteq [\![A]\!]^{\#}(\eta_{\mathsf{t}}\ [u])$

$$[\![x = y;]\!]^{\#}\ T = T \cup \{x\}$$

3.1.2 Improving the values-of-variables analysis

//WIP:

$$\begin{aligned}\mathsf{combine}^{\#}\ M_{\mathsf{cr}}\ M_{\mathsf{ce}} = &\text{let } M'_{\mathsf{cr}} = M_{\mathsf{cr}}|_{\mathit{Locals}_{\mathsf{cr}} \cup (\mathit{Globals} \cap T_{\mathsf{ce}})} \text{ in} \\ &\text{let } M'_{\mathsf{ce}} = M_{\mathsf{ce}}|_{\mathit{Globals} \cap T_{\mathsf{ce}}} \text{ in} \\ &M'_{\mathsf{cr}} \oplus M'_{\mathsf{ce}}\end{aligned}$$

3.2 Implementation

//WIP:
The Goblint analyzer provides a framework for static analyses.

// signature from analysis.ml

As a domain we chose not a set of variables, but a set of lvalues, where an lvalue can be any left hand side of an assignment instruction. Examples of lvalues are: the variable x, the memory location $*xptr$ pointed to by the pointer $xptr$, the third place $a[3]$ in an array a, the member $frac.n$ of a struct $frac$ and many more. This allows us to find that only a part of an array or structure is tainted, so we can keep most of these and only update the tainted lvalues.
//TOP necessary
However analyzing the full language C instead of our toy language, it is necessary to take care of further challenges like pointers and library or unknown functions.
// Goblint provides MayPointTo()
// Goblint provides library descriptors -> unknown = top

4 Evaluation

4.1 Testing

4.2 Benchmarking

5 Conclusion

Abbreviations

CFG Control flow Graph

List of Figures

2.1 Example program (left) and corresponding CFG (right) 2

List of Tables

Bibliography

[RY20] X. Rival and K. Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.