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 Präzisionsverlustes 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

1 Introduction

Abstract interpretation is a fascinating theory. When its principles are used in static analysis, one can proof certain properties about computer programs. Through the sound abstractions of abstract interpretation it is ensured that any proven property holds true for all possible executions of a program. The most interesting parts of this application are the different ways by which states of a program can be abstracted and how these abstractions can be combined to gain various kinds of information about a program.
The GOBLINT analyzer is a project that applies the principles of abstract interpretation to create a static analyzer [Goba].
This analyzer is specialized in, but not limited to finding concurrency bugs. These are some properties it aims to check:

- Race Detection: Checking that accesses to shared memory never happens simultaneously.
- Assertions and Dead Code: Checking whether specific logical expressions are definitely true at given points within the program.
- Integer Overflows: Verifying that no integer overflows occur in the program.

To gain information about a program, GOBLINT performs various kinds of analyses on the source code. These analyses abstract states of the program in different ways. They are also able to communicate with each other to profit from the information gained by other analyses. Because it is easily expandable, GOBLINT is an interesting framework to try out new approaches in static analysis.

The analyzer is highly configurable. This allows the user to fine-tune the degree of precision they wish. However, it has to be considered that usually a higher degree of precision also results in a higher computation time.
With such a configuration option the user is able to specify the degree of context sensitivity for each analysis. It is possible to set an analysis to be performed fully context sensitively, context insensitively or partially context sensitively. Context sensitivity describes the degree to which entry states of functions are differentiated.

Consider the program in Figure 1.1. Assume this program is analyzed with the goal to find which values the program variables can have during program execution. For that an analysis is used that tracks a set of integers for each variable, i.e., it computes an abstract state describing the possible values of all variables for each program point. The abstract state (in this case a set of values per variable) for a program point is computed by applying the effect of an action, e.g., an instruction, to the state before this action. Most actions have effects that are easily computable, however function calls are of a more complicated nature, as they can be called from multiple places in a program. The program in Figure 1.1 contains two calls to `function()`, one in Line 8 and the other in Line 11. If this program is analyzed context sensitively, the function is analyzed twice: Once with an abstract entry state describing that "`glob` = 1" and once with a state describing "`glob` = 2". However, if the analysis is performed context insensitively, both of these two entry states are joined into one abstract state which is used to analyze the function only once. This joined abstract state then has to describe the concrete states, where "`glob` = 1 or `glob` = 2", which is less precise than either of the individual states from before.
When the state after a function call is computed, the information about `glob` is taken from the state returned from the callee. This is because `glob` is a global variable and thus its value can be changed in function calls.
However, consider the case, where `glob` was not changed during the call to `function()`. Then the information about `glob` in the callee return state is the same as in the entry state for that specific call. This means, that in the case of the context-insensitive analysis, the less precise information "`glob` = 1 or `glob` = 2" from the joined entry state is used for `glob` after both calls. This is a loss of precision, considering that the value of `glob` was not altered in the call, and it would be sound to keep the information about `glob` from before each call.

We think this precision loss is avoidable in many cases, where a piece of less precise information is taken from the callee state, even though that piece of information was not changed during the call. Thus, in this thesis we explore a way to reduce this kind of precision loss of partial contexts in abstract interpretation.

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

```c
void function() {
  //...
}

int glob;
void main() {
  glob = 1;
  function();

  glob = 2;
  function();
}
```

Figure 1.1: C code sample with multiple function calls

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

Abstract interpretation is the theory of approximating computer programs in a sound manner. That means that an abstraction might not be absolutely precise, but it definitely is not wrong, i.e., all possible concrete states and properties of a program are described by their abstraction.

An application of abstract interpretation is static analysis. As Rival [RY20] defines it, static analysis is "[...]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 in this thesis, i.e., analyses which find properties of the program dependent on the location within it. We will introduce a syntax to formalize flow sensitive analyses in the following sections. This formalization approach is heavily based on [ASV12].

```
void main() {
  int x;
  x = 0;
  if (x == 0)
    x = x + 1;
}
```

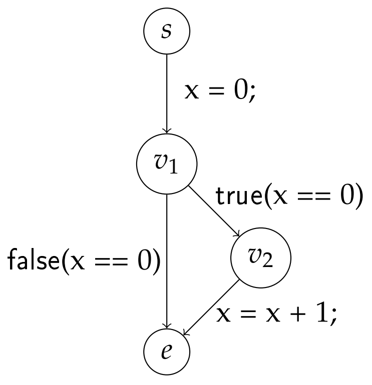

Figure 2.1: Example program (left) and corresponding CFG (right)

2.1 Flow sensitive analysis

As noted above flow sensitive analyses aim to 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, from now on called "program point". This state can describe many things dependent on the analysis performed.
First we define what a program point is: Consider 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 is 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. A concretization function $\gamma : \mathbb{D} \rightarrow 2^{\mathcal{C}}$ can be defined to extract the set of concrete states which are described by an abstract state. Here $\mathcal{C}$ is the set of all possible concrete states.
Then an analysis is expected to find a mapping $\eta : N \rightarrow \mathbb{D}$ which maps program points to abstract states describing that location within the program, i.e., for $[v] \in N$, $\eta\ [v]$ should be an abstract state describing all possible states (and possibly more) the program can be in at program point $[v]$.

As an example we will introduce a values-of-variables analysis for integers. This analysis finds a mapping from a set of program variables X to abstractions of their possible values at any given program point. Our toy language will support global variables (globals) as well as local variables (locals). The global variables can be accessed and changed by any procedure, while local ones are only visible to the procedure in which it was declared and can only be accessed and changed by this procedure. Therefore, our set X of variables is the disjoint union of globals G and locals L: $X = G \uplus L$. 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 $X \rightarrow 2^{\mathbb{N}}$ for each program point.
Combining this with the considerations from above, we chose the mapping $\mathbb{D}_{\mathsf{v}} = X \rightarrow 2^{\mathbb{N}}$ as the abstract domain for the values-of-variables analysis. To illustrate what an abstract state from this domain describes, we define the concretization function $\gamma_{\mathsf{v}} : \mathbb{D}_{\mathsf{v}} \rightarrow 2^{\mathcal{C}_{\mathsf{v}}}$. Here a concrete state is a mapping from variables to a single value each: $\mathcal{C}_{\mathsf{v}} = X \rightarrow \mathbb{N}$. Thus, we define the concretization function as follows:

$$\gamma_{\mathsf{v}}\ M = \{\widehat{M} \in \mathcal{C}_{\mathsf{v}} | \forall x \in X : (\widehat{M}\ x) \in (M\ x)\}$$

In summary this the resulting $\eta_{\mathsf{v}} : N \to \mathbb{D}_{\mathsf{v}}$ for this analysis describes a mapping $\eta_{\mathsf{v}}\ [v]$ for some program point $[v] \in N$, where $\eta_{\mathsf{v}}\ [v]\ x$ is a set containing all values $x \in X$ may possibly hold at $[v]$. From this we can conclude that x cannot hold any value outside $\eta_{\mathsf{v}}\ [v]\ x$ at program point $[v]$.

2.2 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 $\sqsubseteq$ 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 rise 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 rise to the start constraint $\eta\ [s] \sqsupseteq \mathsf{init}^{\#}$ for the starting point of the program $[s] \in N$.

We will show these ideas with our example of the values-of-variables analysis: For this a partial ordering $\sqsubseteq_{\mathsf{v}}$ on the domain $\mathbb{D}$ has to be defined. This ordering is needed to formulate the constraints. We define $\sqsubseteq_{\mathsf{v}}$ as follows: A mapping $M_1 \in \mathbb{D}_{\mathsf{v}}$ is ordered below or equal to another mapping M_2, if and only if for every variable $x \in X$, the set x is mapped to in M_1 is a subset of or equal to the one x is mapped to in M_2. Formulated formally this is:

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

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 (both locals and globals) 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.

In general, for assignments of expressions e we evaluate the expression abstractly, i.e., such that the result of the evaluation is the set that contains all integer values e could possibly evaluate to. For example

$$[\![x = x + 1;]\!]^{\#}_{\mathsf{v}} \{x \mapsto \{1,2\}\} = \{x \mapsto \{1,2\}\} \oplus \{x \mapsto \{2,3\}\} = \{x \mapsto \{2,3\}\}$$

A much deeper insight in how expressions are evaluated in abstract interpretation can be found in Patrick Cousot's book "Principles of Abstract Interpretation" in Chapter 3 [Cou21].

2.3 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 $f()$. For simplicity, we will only consider argumentless procedure calls without a return value in our formal descriptions. Arguments and return values can be simulated by using global variables.
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 consider global variables and how the procedure affects these.
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 for the procedure $f()$, 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:

$$\eta\ [s_f] \sqsupseteq \bigsqcup \{d \exists (u_n, f();, v_n) \in \mathit{Edges} :\ \mathsf{enter}^{\#}\ (\eta\ [u_n]) = d\}$$

where $\bigsqcup$ is the least upper bound, i.e., the least $d \in \mathbb{D}$ according to the ordering $\sqsubseteq$ that is ordered above all of its argument elements.

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 start state and combining the caller state with the returned callee state 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.
The $\mathsf{enter}^{\#}_{\mathsf{v}}$ function first takes the part of the mapping from the caller that contains information about global variables. To the resulting state it adds the local variables used in the procedure. These map to N because they randomly initialized in our toy language.
For $\mathsf{combine}^{\#}_{\mathsf{v}}$ the local part of the caller is kept, but it is updated with the global part of the callee return state. This is done, because the latter contains the updated information about global variables after the procedure call.

2.4 Context sensitivity

In the previous chapter we approached the analysis of procedures by analyzing them only once with an abstract start state describing all possible concrete states the procedure could start with. We call this behavior "context-insensitive" as the procedure is analyzed without differentiating between different states with which it is called.
This is not very precise as we will exemplify by applying the values-of-variables analysis to the program in Figure 2.2. We ignore the marked lines of the program for now. The procedure `incr()` is called twice: Once with $a = 1$ in Line 10 and once with $a = -3$ in Line 13. This leads to two constraints for node $[s_{incr}]$:

$$\eta_{\mathsf{v}}\ [s_{incr}] \sqsupseteq_{\mathsf{v}} \mathsf{enter}^{\#}_{\mathsf{v}}\ \eta_{\mathsf{v}}\ [v_2] = \{a \to \{1\}\}$$

$$\eta_{\mathsf{v}}\ [s_{incr}] \sqsupseteq_{\mathsf{v}} \mathsf{enter}^{\#}_{\mathsf{v}}\ \eta_{\mathsf{v}}\ [v_5] = \{a \to \{-3\}\}$$

leading to $\eta_{\mathsf{v}}\ [s_{incr}] = \{a \to \{-3, 1\}\}$. At the end point of the call the state will be $\eta_{\mathsf{v}}\ [e_{incr}] = \{a \to \{-2, 2\}\}$, which is then combined with the states of nodes in the main procedure. Therefore, the state at Node $[v_6]$ will be $\{a \to \{-2, 2\}\}$, which is used to check the `assert(a < 0);` in Line 14. The result of this assertion cannot be determined by the analysis even though it is easy for humans to see that it should hold.

This could have been avoided, if the procedure was analyzed twice, once with each entry state. To achieve this we modify our current approach as follows: Instead of searching a mapping $\eta : N \rightarrow \mathbb{D}$ we now seek $\eta : (N \times \mathbb{C}) \rightarrow \mathbb{D}$. We call the second part of $(N \times \mathbb{C})$ "context" and $\mathbb{C}$ the "context domain". For now, the context domain $\mathbb{C}$ will be the same as the domain of abstract states $\mathbb{D}$.
This allows us to have different states for the same program point. For now, we differentiate states corresponding to the same program point by the entry state, with which the current procedure was called. Therefore, we adjust the constraints for $\mathsf{enter}^\#$ and $\mathsf{combine}^\#$ as follows:

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

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

With these changes, we can track states for different entry states to procedure calls. In the combine, only the return state that corresponds to the entry state of this specific call is taken into account.
For our toy language, we assume that the main procedure cannot be called from another procedure. Thus, the context for its nodes can be chosen arbitrarily.

There are no changes we need to perform on the values-of-variables analysis to make it context-sensitive. Solely the changes to the general analysis framework above suffice. Applying this changed analysis to the example in Figure 2.2 would lead to the procedure `incr()` being analyzed twice with different contexts, assuming we still ignore the marked lines. This leads to the following two entry constraints for different unknowns of the constraint system:

$$\eta_{\mathsf{v}}\ [s_{incr}, \{a \rightarrow \{1\}\}] \sqsupseteq_{\mathsf{v}} \{a \rightarrow \{1\}\}$$

$$\eta_{\mathsf{v}}\ [s_{incr}, \{a \rightarrow \{-3\}\}] \sqsupseteq_{\mathsf{v}} \{a \rightarrow \{-3\}\}$$

For node v_6 only the state $\eta_{\mathsf{v}}\ [e_{incr}, \{a \rightarrow \{-3\}\}] = \{a \rightarrow \{-2\}\}$ is combined with the caller state from before the call. With this information we can safely say that the assertion in the following Line 14 will hold.

2.5 Partial context sensitivity

While the context-sensitive approach from the previous section might be very precise, it can be quite costly in terms of computation time. To reach a middle ground between

```
int a , c;

void incr() {
  a = a + 1;
}

void main () {
  a = 1;
  c = 10;
  incr();
  a = -3;
  c = -10;
  incr();
  assert(a < 0);
  assert(c < 0);
}
```

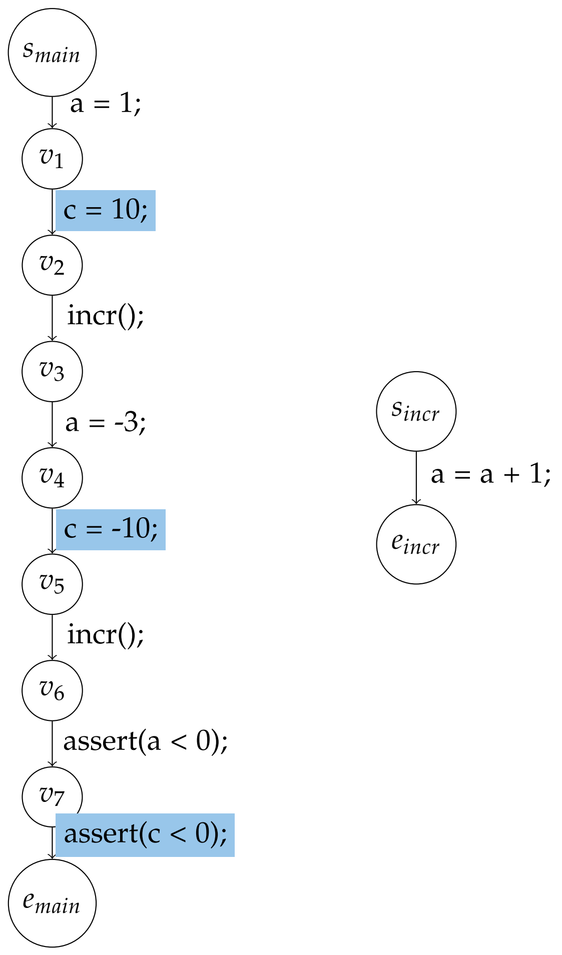

Figure 2.2: Example program (left) and corresponding CFGs for `main` (middle) and `incr` (right)

a context-insensitive and a fully context-sensitive analysis, we change the approach so that contexts are different from the entry state of a call. With this we can group entry states by contexts to analyze a procedure multiple times. This time we group not once per individual entry state, but once per group of entry states.
Thus, in the formal description we now lift the limitation that $\mathbb{C} = \mathbb{D}$. This allows for differentiating function calls not by the entry state but by something different defined by the analysis.
To compute the context when entering a procedure, we define a new function $\mathsf{context}^\#$: $\mathbb{D} \to \mathbb{C}$. Additionally, the constraints for $\mathsf{enter}^\#$ and $\mathsf{combine}^\#$ are changed as follows:

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

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

for an edge $(u, f();, v)$.
This formalization results in multiple constraints for a single contextualized starting variable $[s_f, c']$. We can alternatively formulate this as

$$\eta\ [s_f, c'] \sqsupseteq \bigsqcup \{\mathsf{enter}^\#\ (\eta\ [u_n, c_n]) | \exists (u_n, f();, v_n) \in Edges :\ \mathsf{context}^\#\ (\eta\ [u_n, c_n]) = c'\}$$

i.e., the constraint for the variable $[s_f, c']$ is the least upper bound of all entry states for some call of f, which have the same context c' as the constraint variable. Or expressed differently, all states computed by $\mathsf{enter}^\#\ d$ for f are grouped by the context $c' = \mathsf{context}^\#\ d$, where each group is joined by $\bigsqcup$ to produce a constraint for a starting variable $[s_f, c']$ with the respective context.
With this formal model we have the option to perform an analysis completely context sensitively ($\mathbb{C} = \mathbb{D}$ and $\mathsf{context}^\# = \mathsf{enter}^\#$), completely context insensitively ($\mathbb{C} = \{\bullet\}$) or anything in between. Note that we define $\{\bullet\}$ as the "unit domain" which contains exactly one element with the trivial ordering $\bullet \sqsubseteq \bullet$.

We have to note here that there are some severe issues with the approach for (partially) context-sensitive analyses described in this thesis: The resulting system of constraints may not be finite and some variables in the constraint system may depend on an infinite number of other variables. Thus, it is computationally complex to compute a solution to the system of constraints. For simplicity, we stick with the described methodology in the scope of this thesis.
An extension to the approach which addresses the mentioned issue can be found in [ASV12].

2.6 Precision loss

The main source of the precision loss in context-insensitive or partially context-sensitive analyses is the join over all states with the same context, i.e., when we take the least upper bound of a group of entry states. Consider a procedure f that has no effect, i.e., $s_f = e_f$. Even for this procedure, the combine$^\#$ function receives the less precise result of the join $\bigsqcup$ to combine it with the caller state. In this simple case, the result would be more precise if the combine$^\#$ function could directly use the result from the corresponding enter$^\#$ as the callee state for combining.
Even for procedures that do change the state, there might be some parts of the state which are untouched by the call. If we can identify these untouched parts, we could reduce the precision loss experienced by using partial contexts.

We clarify the source of the precision loss mentioned above with an example: For this we once again consider the example program Figure 2.2. This time we take the marked lines into account. When the program is analyzed context insensitively, not only does the state at the start node for `incr()` s_{incr} represent two possible values for the global variable `a`, but also for `c`. Therefore, the state for this node is

$$\eta_{\mathsf{V}}\ [s_{incr}, \bullet] = \{a \rightarrow \{-3, 1\}, c \rightarrow \{-10, 10\}\}$$

Even though the variable `c` is never changed within `incr()`, the mapping $c \rightarrow \{-10, 10\}$ is still copied into the caller state when combining the states for node v_6. Thus, the information gained by the context-insensitive values-of-variables analysis does not suffice to determine the assertion in Line 15 to hold. This loss of precision could easily be avoided if we had some idea which global variables are definitely not changed by a procedure call.

3 Combatting the Precision Loss of Variable Analyses

In this chapter we describe our approach to reduce the precision loss described in Section 2.6 for analyses tracking information in relation to variables. We call analyses of this kind "variable analyses".
First we use the syntax for flow sensitive analyses from Chapter 2 to formally explain the idea. After that we explain a concrete implementation of the approach into the Goblint analyzer.

3.1 Formal description

We want to reduce the mentioned precision loss for variable analyses. The basic idea to achieve this, is to track for each procedure which variables have been written or have possibly been altered in some other way within that procedure. This information is then used in variable analyses when combining the abstract state from the caller with the abstract return state given by the callee at the end of the procedure. We will exemplify this with the values-of-variables we introduced in Chapter 2.

In the following we 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 in the following section. It is worth mentioning that our notion of taintedness is related but different from other uses of the term "taint analysis".

3.1.1 Taint analysis

In this section we define the syntax for the taint analysis we propose in this thesis: 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:

$$\mathbb{D}_t = 2^X \text{ with } \sqsubseteq_t = \subseteq$$

To be compatible with the notion of partially context-sensitive analyses from Section 2.5 we need to also specify a context domain $\mathbb{C}_\mathsf{t}$ which we define later. Note that we seek to compute a mapping from program points (with context) to sets of variables, i.e., $\eta_\mathsf{t} : (N \times \mathbb{C}_\mathsf{t}) \rightarrow \mathbb{D}_\mathsf{t}$. To interpret this with the goal of our taint analysis in mind, we note that $\eta_\mathsf{t}[n, \bullet] = T$ denotes 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 current procedure up until the program point n. Note here that the tainted set T not only includes variables which have been tainted by statements of the current procedure, but also variables which have been tainted within procedures called by the current one.
It remains to define $\mathbb{C}_\mathsf{t}$, $\mathsf{init}^\#_\mathsf{t}$, $\mathsf{enter}^\#_\mathsf{t}$ and $\mathsf{combine}^\#_\mathsf{t}$ 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 without any variable being initially tainted when entering a procedure. It is worth pointing out that the entry to a procedure call does not depend on the state where it is called. Therefore, we design our analysis to be context-insensitive, i.e.,

$$\mathbb{C}_\mathsf{t} = \{\bullet\} \text{ and trivially } \mathsf{context}^\#_\mathsf{t}\ T = \bullet$$

With these considerations we can also define $\mathsf{enter}^\#_\mathsf{t}$ and $\mathsf{init}^\#_\mathsf{t}$ as follows:

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

It is worth pointing out here that the function $\mathsf{enter}^\#_\mathsf{t}\ T$ is always equal to the empty set irregardless of its argument T. Therefore, it computes the same entry state for each call of a certain procedure.
When combining the caller state with the returned callee state, we note that we need to keep the tainted set from before the call, as a tainted variable can never get "untainted" again, no matter what the procedure does. Additionally, we add the tainted set returned by the callee, since anything tainted in the call needs to be considered tainted after the call 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 further procedure call. This leaves us with the following equation for the $\mathsf{combine}^\#_\mathsf{t}$ function:

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

Note that we removed the callee local variables $Locals_\mathsf{ce}$ because these are not accessible by the caller and all of its callers anyway, so it is not useful to keep track of them.
Lastly we define the abstract effects of actions. Most of these (including checks) do not do anything besides propagating through the state from before. The only major exception are variable assignments. For these we note that the specific variable, which

the value is assigned to is added to the tainted set. This is independent of the expression that evaluates to the assigned value, as we are only interested in the fact that the variable on the left of the assignment is altered. This leaves us with the following abstract effects of actions:

$$[\![A]\!]^{\#}\ T = \begin{cases} T \cup \{x\} & \text{if } A \equiv (x = e;) \\ T & \text{else} \end{cases}$$

where e is any arbitrary expression.
This concludes our definition of the taint analysis.

3.1.2 Improving Variable Analyses

In this section we see how the information from the taint analysis helps us to improve variable analyses. We show this with the example of the values-of-variables analysis we introduced in Chapter 2.
Recall the source of the precision loss we want to reduce as described in Section 2.6. This happened when a global variable was updated with a less precise value after a procedure call even though this specific variable was not changed by the call.
Thanks to the taint analysis we defined in the previous section, we now have a way to get information about which variables can be altered by a procedure $f()$ and which surely stay untouched. The latter of which are exactly those variables which are not in the tainted set of the end node e_f for that procedure.
With this insight we can now update the $\mathsf{combine}^{\#}_{\mathsf{v}}$ function of our values-of-variables analysis as follows:

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

where $T_{\mathsf{ce}} = \eta_{\mathsf{t}}\ [e_f, c]$ for an edge $(u, f();, v)$ and the corresponding context c.
Similar to before the $\mathsf{combine}^{\#}_{\mathsf{v}}$ function takes the caller mapping, restricts it to a subset of caller reachable variables and updates this mapping with the callee mapping restricted to the rest of caller reachable variables. In other words, the caller reachable variables are partitioned into two sets such that one subset is taken from the caller state while the other one is taken from the callee state. Before our change, the partitioning was done strictly in such a way that only the caller local variables were taken from the caller state and all global variables from the callee state. After our improvement, the global variables that are not tainted by the callee are also taken from the caller state and not from the callee anymore. Thereby the precision loss for untainted variables is eliminated.
One might wonder if our change could lead to a case, where the callee state has a more precise value for a variable that is discarded because this variable is not in the tainted

set. Concretely this situation would be described by

$$\exists \text{ Edge } (u, f();, v),\ x \in \mathit{Globals} : x \notin \eta_\mathsf{t}\ [e_f, \bullet] \wedge (\eta_\mathsf{v}\ [e_f, \bullet]\ x \subsetneq \eta_\mathsf{v}\ [u, \bullet]\ x)$$

where we use the subset orderings $\subseteq$ and $\subsetneq$ to compare the sets of integer values, x could hold at different program points.
From $x \notin \eta_\mathsf{t}\ [e_f, \bullet]$ we know that x has not been altered in the procedure $f()$ since the node s_f up until e_f, and therefore it holds that

$$\eta_\mathsf{v}\ [e_f, \bullet]\ x = \eta_\mathsf{v}\ [s_f, \bullet]\ x$$

Because x is a global we get with the definitions of $\sqsubseteq_\mathsf{v}$ and $\mathsf{enter}^\#_\mathsf{v}$:

$$\eta_\mathsf{v}\ [s_f, \bullet]\ x \supseteq (\mathsf{enter}^\#_\mathsf{v}\ (\eta_\mathsf{v}\ [u, \bullet]))\ x = \eta_\mathsf{v}\ [u, \bullet]\ x$$

Therefore, $\eta_\mathsf{v}\ [e_f, \bullet]\ x \supseteq \eta_\mathsf{v}\ [u, \bullet]\ x$ which is a contradiction to the proposed case which we can therefore exclude.

3.2 Implementation

Before explaining the process of implementing the proposed taint analysis and its usage to improve other analyses, we introduce the Goblint analyzer and its structure in this paragraph. The core functionality of Goblint is to statically analyze C programs using an approach similar to the one described in Chapter 2. This generally works as follows: The C input file is preprocessed with `goblint-cil`, an adaption of the "C Intermediate Language (CIL)" project. CIL is a front-end that parses and compiles C programs into a simplified subset of C [Cil]. For example, it moves assignments which happen within conditional checks out of the check and puts them before it. From the output of the CIL front-end a CFG is generated. This graph is then used together with the specifications of various analyses to generate a constraint system. Goblint solves this constraint system and produces different kinds of outputs to the user according to the solution (e.g. notifications, warnings or a visualization of the full solution).
It is worth mentioning that Goblint can perform multiple analyses on a program at the same time. For this a compound domain is built (for the domain of states as well as for the context domain), that is a tuple of the domains corresponding to the analyses to be performed. To generate constraints, all activated analyses are taken into account, where the specification of each analysis acts on its corresponding part of the compound domain. Information can be transferred between the different analyses via a system called "queries".
Figure 3.1 shows the inner structure of the analyzer. We can see that Goblint provides

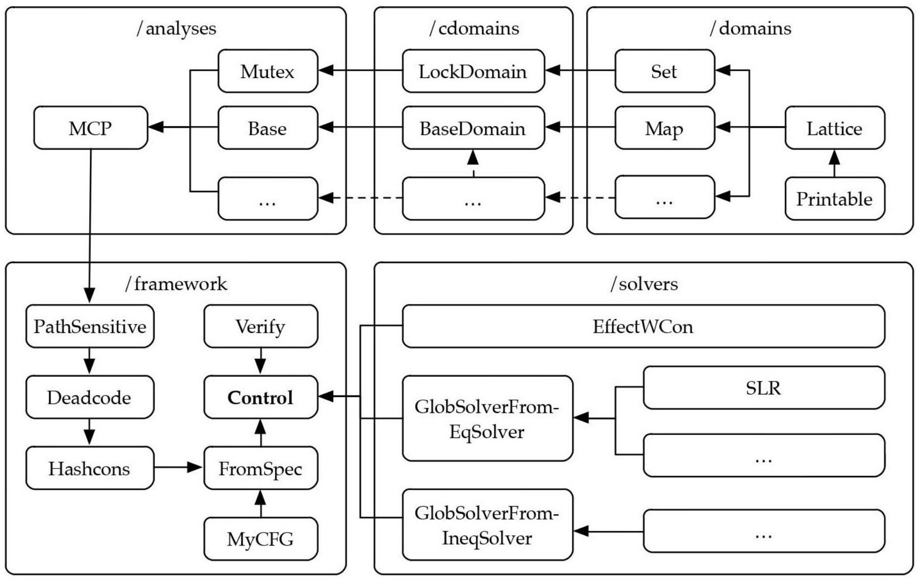

Figure 3.1: Schematic directory structure of Goblint. Adapted from [Api14]

parametrized domains which can be used in the specifications of the analyses, e.g., `Set` and `Map`. It is also shown that multiple analyses are then compounded into one Master Control Program (MCP) that is used by the framework to generate constraints from the CFG. The resulting constraint system is then solved by one of the solvers Goblint provides in `/solvers`.
For a deeper insight into the inner workings of Goblint refer to [Api14].

3.2.1 Taint analysis

To define an analysis the Goblint analyzer provides an interface, where the most relevant parts can be seen in Figure 3.2. This interface requires two modules `D` and `C` which define the domain and the context-sensitive part of the domain. It also requires the following functions:

- `name` to uniquely refer to an analysis.
- `startstate` to define the state used when entering the analysis (similar to $\mathsf{init}^\#$ from Chapter 2).
- `query` to implement the query system of Goblint. This allows an analysis to broadcast information to be used in analyses.

- *Transfer functions* which define the abstract effects of actions (similar to $[\![A]\!]^{\#}$ from Chapter 2).
- *Functions for interprocedural analysis*
- *Functions for the analysis of multithreaded programs*

For our taint analysis we create a new module implementing this interface.
As a `name` for GOBLINT internally we chose `taintPartialContexts` because `taint` was already used, and the `name` needs to be unique. In the following we explain in detail how our new analysis module implements the given interface:

Domain

We need to choose the domain `D` and the context domain `C`. According to the concept of our analysis described in Subsection 3.1.1 the domain should be a set of variables. However, we are now analyzing the C language instead of our toy language. In C not every left-hand side of an assignment is just a simple variable, but can be one of many more complex thing, e.g., the memory location `*xptr` pointed to by the pointer `xptr`, the fourth place `a[3]` in an array `a`, the member `frac.n` of a struct `frac` and many more. All these options are described by a concept called Left Value (of an assignment) (lval). There is an implementation of this type provided by GOBLINT in the `Lval.CilLval` module. To be as precise as possible we use a set of lvals instead of a set of variables for the implementation of the taint analysis.
Another point worth mentioning is that we sometimes need the notion of "all variables" (or rather "all lvals") when we want to express that everything is tainted. While conceptually using the full set X poses no issue, in a concrete implementation this is extremely impractical and not even realizable if the set is infinitely large. For this case GOBLINT provides a parametrized domain `ToppedSet(Base)`. This domain is either a set of elements of the `Base` type or alternatively a `Top` element which can be interpreted as the "full set of all `Base` elements". Therefore, we finally have `D = ToppedSet(Lval.CilLval)` for our domain. Note that this also defines the ordering on the domain to be the regular subset ordering.
It remains to define the module `C`: We noted in Subsection 3.1.1 that our analysis by itself is context-insensitive. Therefore, the context domain of our analysis `C` is empty, which is expressed with the `Unit` domain provided by GOBLINT. Note here, that this does not mean that the taint analysis is always performed context insensitively, i.e., it is only performed once per function. Since GOBLINT uses a compound domain, there may be other context-sensitive analyses, forcing the whole compound analysis to analyze a function multiple times. Our taint analysis however never contributes differing sub contexts to the compound context.

```ocaml
module type Spec =
sig
  (* Domain *)
  module D : Lattice.S
  module C : Printable.S

  val name : unit -> string
  val startstate : varinfo -> D.t
  val query : (D.t, C.t) ctx -> 'a Queries.t -> 'a Queries.result

  (* Transfer functions *)
  val assign: (D.t, C.t) ctx -> lval -> exp -> D.t
  val branch: (D.t, C.t) ctx -> exp -> bool -> D.t

  (* Functions for interprocedural analysis *)
  val special : (D.t, C.t) ctx -> lval option -> varinfo -> exp list ->
      D.t
  val enter : (D.t, C.t) ctx -> lval option -> fundec -> exp list ->
      (D.t*D.t) list
  val return : (D.t, C.t) ctx -> exp option -> fundec -> D.t
  val combine : (D.t, C.t) ctx -> lval option -> exp -> fundec -> exp
      list -> C.t option -> D.t -> Queries.ask -> D.t

  val context : fundec -> D.t -> C.t

  (* Functions for the analysis of multithreaded programs *)
  val threadenter : (D.t, C.t) ctx -> lval option -> varinfo -> exp list
      -> D.t list
  val threadspawn : (D.t, C.t) ctx -> lval option -> varinfo -> exp list
      -> (D.t, C.t) ctx -> D.t
end
```

Figure 3.2: Simplified Interface for implementing analyses in Goblint

The `startstate` function

This function computes the initial state for our analysis similar to the $\mathsf{init}^{\#}$ function we introduced in Chapter 2. As discussed in Subsection 3.1.1, we implement this function so that it returns the empty set.
We note here however, that in practice we do not use the way, `startstate` is implemented in the scope of our thesis. In GOBLINT this function is called before the main function is even entered. Thus, `enter` (which we define later) is still used to compute the entry state for the main function. We chose to implement `startstate` in this way for consistency.

Transfer functions

These functions implement the effect of actions on the state, similar to the abstract effects of actions $[\![A]\!]^{\#}$ in Chapter 2. FOr example `branch` handles checks for if-statements and loops. For this and most other actions our analysis just propagates the state from before, so they use the default implementation from the `Analysis.IdentitySpec` of GOBLINT. The default implementations of `Analysis.IdentitySpec` propagate the given state without change for any action.
Much more interesting is the case of the `assign` function which handles the effect of an assignment to an lval. For this case we want to add the lval to our tainted set. The parameters for the `assign` function are: `ctx` which amongst other things contains the state from before, the lval to which a value is assigned and an expression that evaluates to this value that is assigned. We are only interested in `ctx` and the lval, as for the taint analysis only the fact that a value is assigned is relevant and not its concrete value.
Tainting lvals is not as straightforward as it might seem at first. Just adding it to the state from before, i.e., the tainted set, only suffices if the lval is a specific location in the memory, e.g., a specific (local or global) variable. The lval could however also be a reference to a location in the memory, e.g., a pointer. For these it is not helpful to just taint the reference because we need to know the specific memory locations that are or could be tainted. To solve this issue we make use of GOBLINT's `MayPointTo` query. This takes a reference to the memory and asks all other activated analyses if they have any information about where this reference may point to. Just like everything else in the static analyzer GOBLINT, the answer is an overapproximation, so we can be sure not to miss any location that could be referenced.
In conclusion, tainting an lval goes as follows: If the lval is a specific memory location, this lval is added to the tainted set. If it is a reference to the memory described by some expression, send a `MayPointTo` query to ask other analyses which memory locations this expression may point to and add the returned set of lvals to the tainted set. We

```
let taint_lval ctx (lval:lval) : D.t =
  let d = ctx.local in
  (match lval with
  | (Var v, offs) -> D.add (v, resolve offs) d
  | (Mem e, _) -> D.union (ctx.ask (Queries.MayPointTo e)) d
  )

let assign ctx (lval:lval) (rval:exp) : D.t =
  taint_lval ctx lval
```

Figure 3.3: Implementation of the helper `taint_lval` and the `assign` function

implemented this functionality in a helper function `taint_lval`. Therefore, calling this function is the only thing the `assign` function needs to do as seen in Figure 3.3.

Functions for interprocedural analysis

Here we define the functions `context`, `enter` and `combine`. These functions work similar to their abstract counterparts as described in Chapter 2. In addition to these known functions, the interface also requires two additional functions: `return` which handles return statements right before a function is left and `special` which handles calls to library functions or other functions, for which we do not have the source code we can analyze.
Our implementation does not differ a lot from the proposed formal description in Subsection 3.1.1. Since we are analyzing C and not our toy language, the only major difference is that we need to handle return values and function arguments. Therefore, we implement these functions as follows:

`context`: Our context domain is the `Unit` domain, and we do not want to generate different contexts for this analysis. Thus, this function always returns the unit element.

`enter`: Like we discussed in Subsection 3.1.1, this function always returns the empty set, which is the entry state for the called function.

`combine`: The `combine` function first checks if there is an lval to which the return value is assigned. If so, it taints this respective lval in the caller state using the helper function `taint_lval` introduced in the "Transfer Functions" section. After that it computes the union of the resulting state with the returned callee state and returns it.

Summarized, the result of this function is the union of both states it receives for combining. It additionally adds lvals which are possibly tainted by the return value.

`return`: In our formal description in Subsection 3.1.1, the $\mathsf{combine}_t^\#$ function removed variables unreachable by the caller. In the concrete implementation, we give this functionality to the `return` function, so the removal happens right before the combine. We also remove function arguments as these are unreachable by the caller similar to local variables. It is worth pointing out that we do not just remove all lvals corresponding to local variables or arguments. A function might exist multiple times in the current call stack, e.g., when the function is recursive. This can result in the existence of multiple versions of the same local variable. Goblint treats these as being the same variable. Thus, when we remove a local variable we risk also removing a different version of it lower in the call stack, for which we still need the taintedness information. To address this issue, `return` sends an `IsMultiple` query for each variable to be removed and only removes those, that surely not have multiple versions. This query is already provided by Goblint.

`special`: This function addresses library functions or other functions, for which we do not have the source code to analyze. The simple way to handle these, is to just return Top, i.e., saying "everything could be tainted", after a special call.
This is how we handle unknown functions, however Goblint provides "Library Descriptors", which contain information about some known C library functions, e.g., `printf`, `malloc`, `cos`, etc. With the respective Library Descriptor of a function, we can gain information about which addresses are "shallowly" written and which are "deeply" written by the call. Shallowly written addresses point to lvals which might be directly written. Deeply written addresses however point to lvals where not only the lval itself, but possibly anything it might recursively point to could be written. Therefore, the `special` function makes use of Goblint's `MayPointTo` and `ReachableFrom` queries in the following way:
First the function checks if a Library Descriptor is available. If not, Top is returned. Otherwise, the shallowly and deeply written addresses are obtained from the Descriptor. Consequently, the union of

- the state before the call
- anything that is possibly tainted by the return value (using `taint_lval` like in `combine`)
- the set of lvals returned by the `MayPointTo` query for any shallowly written address

- the set of lvals returned by the `ReachableFrom` query for any deeply written address

is returned by the `special` function.

Functions for the analysis of multithreaded programs

To be able to analyze multithreaded programs, Goblint's analysis interface requires the following functions: `threadenter` to compute the startstate for the newly created thread and `threadspawn` which computes the effect of a thread creating instruction to the state of the creating thread.
We implement the former of these two functions similarly to our `startstate` and `enter`. Thus, `threadenter` returns the empty set.
To implement the other function, `threadspawn`, we consider how a thread creation effects the state of the creator. We note that for our notion of taintedness the only relevant effect is, that the thread creating function may write thread ID variables to which it receives a reference as an argument. Thus, this function uses the helper function `taint_lval` defined in the "Transfer Functions" section to add possibly tainted lvals to the state from before and returns the result.

The `query` function

We want to enable our taint analysis to tell other analyses which lvals are tainted at a specific program point. Therefore, we add a new query `MayBeTainted` to the query system of Goblint. The result of this query should be a set containing lvals which may be tainted, i.e., any lval that is not in the returned set is definitely untainted.
After this addition we are able to make our `taintPartialContexts` analysis answer to this query. Therefore, our analysis implements the `query` function in such a way that it answers only to `MayBeTainted` queries with the current state but does not answer other queries.

3.2.2 Benefiting other analyses

In this section we discuss how we improved other existing analyses in Goblint using the taint analysis we implemented in Subsection 3.2.1.

Improving the `base` analysis

The main analysis that benefits from the taint analysis is the `base` analysis of Goblint. This analysis implements a very much extended approach of the basic values-of-variables analysis we formally defined in Chapter 2. The `base` analysis is however still

based on the main goal and basic concept of finding a mapping from program variables to possible values at each program point. Therefore, this analysis uses a mapping from variables to their possible values as part of its domain. However, here the `ValueDomain` of the mapping is much more complex than just a set of possible integers. It provides abstractions for virtually any type in C, including arrays, structs and pointers. Even more though, the `ValueDomain` is highly configurable. Amongst other options it allows choosing between different ways of abstracting integer values or arrays. One interesting option related to the topic of this thesis is the possibility to choose between different degrees of context sensitivity: the analysis can be fully context-sensitive, insensitive with respect to integer variables (abstracted by intervals or in general), only sensitive with respect to pointers or completely insensitive. When choosing anything but the completely context-sensitive option, this analysis experiences the (avoidable) loss of precision described in Section 2.6.
To reduce this loss we need to change the `combine` function of the `base` analysis so that it uses the results of our `taintPartialContexts` analysis. In order to get a better understanding of what needs to be changed, we first describe how the `combine` function was implemented before our changes:

1. The return value is saved. Its value is removed from the callee state.
2. All globals are removed from the caller state.
3. Everything from the callee state is added to the caller, possibly overwriting caller values. This excludes the return value which is handled separately.
4. Some further adjustments according to the configuration are performed to the resulting state.
5. The saved return value is added to the state before it is returned.

To implement our changes we will focus on the steps 2 and 3, where the caller mapping is updated. The other steps will remain the same.
The core idea to implement the concept proposed in Subsection 3.1.2 is as follows: First we get the set of possibly tainted lvals from the callee. We then iterate over its elements one by one, where for each tainted lval we update the caller mapping with the corresponding value from the callee mapping, i.e., we get the value corresponding to that lval from the callee mapping and set the lval to map to this value in the caller mapping. This functionality of updating the caller mapping with the callee mapping using the tainted set is implemented in a helper function `combine_st`.
Before we explain how this helper function is implemented, we first show how it is embedded in the current implementation of the `combine` function. As discussed, we

alter steps 2 and 3: First we send a `MayPointTo` query to the return state of the callee. We then check if the query returned the Top set, i.e., the notion that everything is tainted. In this case we perform the unchanged steps 2 and 3 just like before. Amongst other cases, this can happen, when the `base` analysis is run without our taint analysis being activated.
We now define what happens, when the result of the query is an explicit set. Before calling the helper function `combine_st`, we have to handle two special cases here:

- For a global variable, there is no mapping in the callee state, but there is one in the caller state. This case can occur in multithreaded mode, if this variable was protected by a mutex before the call, but the mutex was released in the called function. In this case, the mapping for this variable would be removed from the state within the callee. In the `combine` function, we do need to keep this new information from the callee for such a variable, i.e., remove it from the caller mapping. Therefore, we filter over the caller mapping and remove all globals, for which there exists no mapping in the callee mapping.
- For a global variable, there is a mapping in the callee state, but there is none in the caller state. This case can occur if new information is gained within the call, e.g., some new memory is allocated. This information is not tracked by the tainted set and would therefore not be copied into the caller state. Since we still want to have this new information after the combine, we add all these mappings from the callee to the caller.

These cases have to be handled separately, as for these the corresponding lval is not necessarily contained in the tainted set. After the two special cases are handled, we use the `combine_st` helper function to finally update the tainted lvals in the caller state. We then proceed with the resulting state to the steps 4 and 5 like before.
We note here that we added a new parameter `f_ask` to the `combine` function. To do this we had to update the analysis interface and consequently all analyses implementing it. This new parameter allows the analyses to send queries to the returned callee state, which was not possible before.

`combine_st`: This helper function takes the caller state (updated according to the two special cases), the callee state and the set of tainted lvals. The difficulty here is, that while the tainted set is a set of lvals, the mappings from the states of the `base` analysis are mappings from variables to abstractions of their possible values. This means, that our tainted set may include specific places in an array or specific members of a struct, e.g., `a[3]` and `frac.n`. In contrast to this, the mappings we want to combine do not map `a[3]` and `frac.n` to abstract values, but rather map `a` to some abstraction of an

array and `frac` to some abstraction of a struct. To solve this issue we make use of the `get` and `set_savetop` functions provided by the module of the `base` analysis. With these functions it is possible to get and set values of addresses to specific lvals in a variable mapping of the `base` analysis.
Therefore, the implementation of this function goes as follows: We fold over the tainted set. For each lval we build an address to this lval. Then we try to `get` the value this address points to from the callee state. If this returns a value, we use `set_savetop` to update the caller state, i.e., set the address to the value we got. Otherwise, we proceed with the next lval.
There are however a few special cases to handle: One issue is that in Goblint there exists a domain for abstracting array values called "partitioned array". This abstraction saves an index which it uses to split an array into three parts: The group of all values to the left of the index, the value at the index itself and the group of all values to the right. Each of the three parts is abstracted with a collective value. The index can be either a specific integer or a variable.
For array variables abstracted with this "partitioned array" domain, copying lvals one by one does not work, as the information of the partitioning is lost when we attempt it. Therefore, we check if the current value corresponds to a place in an array abstracted with the "partitioned array" domain and if so, we copy the whole partitioned array from the callee mapping to the caller mapping.
A similar issue occurs with values of void type, as for these the `get` does not work correctly. Therefore, we get the value for the corresponding variable from the callee mapping itself and update the caller mapping with it.
The last issue we need to address is again related to partitioned arrays. Recall, that an array can be partitioned by the value of a variable. This means, that if a variable is tainted which is used as a partitioning index, the partitioned array in the caller mapping is invalid. Therefore, for each lval we check if it corresponds to a variable partitioning an array. If so, we update the caller state by copying all abstracted arrays which are partitioned by the variable in question from the callee mapping to the caller mapping. This is possible, because the state of the `base` analysis keeps track of which arrays are partitioned by certain variables.
Finally, after we are done folding over all tainted lvals, the function returns the modified caller state.

Improving other Variable Analyses

In Goblint there are multiple other analyses which can be described by our notion of a variable analysis. For some of these we are able to the precision loss of partial contexts by using the results from our `taintPartialContexts` analysis. We briefly discuss the details of these improvements in this section:

The `varEq` analysis:
This analysis tracks, which lvals definitely hold the same value, irregardless of what this value is. As an example, after an assignment `a[3] = y;` this analysis propagates the information that `a[3]` and `y` in an equality set, until either `a[3]` or `b` are written.
One can construct a case, where a function `f()` is called twice, once where some equality `x = y` holds and once where it does not hold. We assume that `x` and `y` are not altered within `f()`. If the `varEq` analysis is performed context insensitively, we would lose the equality information when the `combine` is performed after both calls. This is because the equality information is discarded when the entry states are joined.
So far, the `combine` function just propagated the callee return state. This can be improved in the following way: We obtain the tainted set and remove all equality sets containing at least one tainted lval from the caller state. Then we compute the greatest lower bound of the resulting caller state and the callee state. This means, we compute a state that unifies the information from the caller state without tainted lvals with the information from the callee state.
In the example from above this changed `combine` allows the analysis to keep the equality `x=y` when the taint analysis is activated.

The `relation` analysis:
This analysis tracks relations between variables. The analysis can be configured to use different kinds of relations implemented in different domains. An example for these domains is the octagon domain, which works with relations of the form $\langle x\rangle + \langle y\rangle \geq 0$, where $\langle x\rangle$ stands for either x or $-x$. The relation analysis is parametrized in such a way that our changes apply to the analysis independent of the selected relation domain.
The case where this analysis unnecessarily loses precision because of context insensitivity, is similar to the one discussed in the paragraph discussing the improvement of the `varEq` analysis: A function `f()` is called twice, where some relation between the variables `x` and `y` once holds and once does not hold. When the entry states for both calls are joined, the relation in question is removed and is therefore missing after the call. We once again assume that `x` and `y` are not altered within `f()`.
Before we apply our changes, the `combine` function of the `relation` analysis is implemented so that it removes all information related to global variables from the caller state and merges the result with the callee state. The result contains the relations from the caller that are not related to globals and all relations from the callee.
Accessing the result of our taint analysis via a query, we can improve this way of combining. We achieve this by only removing relations related to tainted global variables

and keep those that only relate to locals and untainted globals. After that, the result is merged with the callee state like before.
We note here, that this analysis tracks relations between variables while our tainted set contains tainted lvals. Therefore, we convert the set of tainted lvals to a set of tainted variables. This is easily possible, since all lvals in our tainted set refer to variables or specific parts of variables. In particular, they do not point to some memory location, which would make identifying a corresponding variable difficult.

The `condVars` analysis:
This analysis tracks equalities between variables and logical expressions. Take the following statement as an example: `tv = (c == 0);`. For this statement, the `condVars` analysis tracks, that the variable `tv` holds the value of the logical expression `c == 0`. This information can be used, when there is an `if` statement, checking whether `tv` is true or not. Knowing `tv` is equal to `c == 0`, this analysis can provide the information that `c == 0` evaluates to `true` in the `true`-branch of the `if` statement.
Currently, the `combine` function of this analysis is implemented to discard the callee return state and remove all information related to global variables from the caller state. This results in the loss of all information related to globals whenever a function is called, irregardless whether the `condVars` analysis is performer context sensitively or insensitively.
We can improve this by only removing information related to tainted globals from the caller state and keeping information related to untainted globals. This makes the `condVars` analysis more precise whenever the `taintPartialContexts` analysis is activated. Again we note here, that this analysis is concerned with information related to variables, not lvals. Therefore, we convert the set of tainted lvals to a set of tainted variables like in the previous paragraph.

4 Combatting the Precision Loss of Thread Analyses

To (((Something))) M. Schwarz et al. [Sch+23].

4.1 Theory

In Chapter 6 of their paper [Sch+23] they propose an analysis that identifies threads by their creation history. Among other things, this analysis helps to identify which threads are unique and which actions may or may not happen in parallel.
As mentioned, threads are identified by their creation history, which is used as an ID to identify different threads. This history is a sequence of create edges starting with main. To prevent such a history to grow to infinity, they define the notion of non-unique thread IDs which may identify multiple threads each.
Formally, the set of possible abstract thread IDs $\mathcal{I}^\#$ is $(\mathsf{main} \cdot \mathcal{P}^*) \times 2^{\mathcal{P}}$, where $\mathcal{P}$ is the set of create edges and $\mathcal{P}^*$ a sequence of such edges. $\langle u, f \rangle \in \mathcal{P}$ refers to an outgoing edge from program point u which creates a thread starting at f. In this notion, IDs of the form $(i, \emptyset) \in \mathcal{I}^\#$ are unique, while $(i, s) \in \mathcal{I}^\#$ are not unique if $s \neq \emptyset$.

As mentioned, these abstract thread IDs are found by a dedicated thread ID analysis. In order to work correctly, this analysis needs to track the current thread ID as well as a set of create edges already encountered. Thus, the abstract domain for this analysis is $\mathbb{D}_{\mathsf{tID}} = (\mathcal{I}^\# \times 2^{\mathcal{P}})$. An element (id, es) of this domain is the product of the current thread ID id and the set of encountered create edges es.
The main part of the analysis now works as follows: Assume a thread-create edge $(u, create(f), v)$ is encountered with a state (id, es). When the id is already non-unique, the new thread is identified with a (possibly new) non-unique ID. In the other case id is unique, i.e., $id = (i, \emptyset)$. Then the analysis checks whether the currently encountered edge was already encountered before, i.e., if it is present in the set of encountered edges es. If not, the new thread is identified with a new unique ID that is created by appending the current create edge to the first part of $id = (i, \emptyset)$. Otherwise, the new thread is identified with a non-unique ID.
It is worth mentioning, that the $\mathsf{combine}^\#_{\mathsf{tID}}$ function is implemented to ignore the caller

state and propagate the callee return state for this analysis.
As of now the thread ID analysis is always run context sensitively, i.e., $\mathbb{C}_{\mathsf{tID}} = \mathbb{D}_{\mathsf{tID}}$ and $\mathsf{context}^{\#}_{\mathsf{tID}}\ D = \mathsf{enter}^{\#}_{\mathsf{tID}}\ D$. Performing it context insensitively is not practical. The reason for that is leads to cases where a least upper bound of two states (id_1, es_1) and (id_2, es_2) has to be computed. Since abstract IDs from $\mathcal{I}^{\#}$ are not reasonably comparable, this leads to a notion of "unknown thread ID" or "any possible thread ID" which we want to avoid.
It is much more reasonable to run the thread ID analysis partially context sensitively, where contexts are differentiated with respect to the current *id* but not the set of encountered create edges *es*. Computing the least upper bound of sets of encountered create edges is reasonably possible by taking the union of the sets. With these insights we can perform the thread ID analysis partially context sensitively with

$$\mathbb{C}_{\mathsf{tID}} = \mathcal{I}^{\#} \text{ and } \mathsf{context}^{\#}_{\mathsf{tID}}\ (id, es) = id$$

When we now run this partially context-sensitive analysis, it is possible to encounter cases where precision is lost similarly to how we described it in Section 2.6. We show this with the example program from Figure 4.1. In this program, `procedure()` is called two times, once without a thread being created beforehand and once with a thread created. For now this thread created in Line 14 has the unique ID $([main \cdot \langle u_2, s_{tproc}\rangle])$, assuming the program point before the `create(tproc)` is u_2 and s_{tproc} is the start point of the thread procedure `tproc()`. The two entry states for `procedure()` are $(([main], \emptyset), \emptyset)$ and $(([main], \emptyset), \{\langle u_2, s_{tproc}\rangle\})$, as once the edge $\langle u_2, s_{tproc}\rangle$ was encountered and once it was not. These states both have the same context, i.e., $([main], \emptyset)$, and thus their entry states are joined. The result is the same as the second entry state $(([main], \emptyset), \{\langle u_2, s_{tproc}\rangle\})$. Since we assume for this example, that no thread was created in `procedure()`, the return state is the same as the joined entry state. With the way, the $\mathsf{combine}^{\#}_{\mathsf{tID}}$ function is defined, this state is used for the point after both calls. In particular, it is used for the state after the call in Line 11. Thus, the `create()` action in Line 14 is actually encountered again with the state $(([main], \emptyset), \{\langle u_2, s_{tproc}\rangle\})$. The result of this is, that the thread created here is now identified with a non-unique abstract ID $([main], \{\langle u_2, s_{tproc}\rangle\})$, because this create edge is in the set of encountered create edges.
With this result, some possible race condition are detected, even though in fact there are none in this program. For once, The thread created in Line 14 is no longer identified with a unique edge. Thus, it is no longer possible to know that only one thread runs with the procedure `tproc()` and consequentially, only one thread can perform the assignment instructions in Line 4. Therefore, a data race is found for this assignment. Furthermore, it is no longer possible to identify, that no thread has been crated before the variable assignment in Line 13, because $\langle u_2, s_{tproc}\rangle$ is in the set of encountered

```c
int a;

void *tproc() {
  a = 2;
}
int procedure() {
  //... (no thread creation)
}

int main() {
  procedure();

  a = 1;
  create(tproc);

  procedure();
}
```

Figure 4.1: Example program to illustrate the precision loss of partial contexts in the thread ID analysis

create edges of the state at this point. Thus, a possible data race is found also for this assignment instruction.
It is worth mentioning here, that the process we described above is adapted to be better understandable for the reader. Following the actual definitions of static analysis, the corresponding system of constraints for the program points is generated. For this system a solution can be computed with a fix-point solver.

To combat this loss of precision, we propose a "thread-create" analysis. This analysis checks for each procedure, whether a thread is possibly created between the entry to it and the return. Note that it does not matter, if a thread is created in the procedure itself or in another procedure which the procedure called. The domain we use for this analysis is the set of boolean values $\mathbb{D}_{\mathsf{tc}} = \{\mathsf{true}, \mathsf{false}\}$. The analysis tracks whether a procedure *may* create a thread. Thus, we encode uncertainty, i.e., "a thread *may* have been created" with true. Similar to the taint analysis from Subsection 3.1.1, this analysis is context insensitive by itself and thus, $\mathbb{C}_{\mathsf{tc}} = \{\bullet\}$.
In conclusion, the state at some program point $\eta_{\mathsf{tc}}\ [v, \bullet]$ answers the question "*May* a thread have been created since the entry of the current procedure up to the node v?".

In the following we give the definitions of the analysis functions for this function:

$$\begin{aligned}
\mathsf{init}^{\#}_{\mathsf{tc}} &= \mathsf{false} \\
[\![A]\!]^{\#}_{\mathsf{tc}}\ c &= \begin{cases} \mathsf{true} & \text{if } A \equiv (create(f);) \\ c & \text{else} \end{cases} \\
\mathsf{enter}^{\#}_{\mathsf{tc}}\ c &= \mathsf{false} \\
\mathsf{combine}^{\#}_{\mathsf{tc}}\ (c_{\mathsf{cr}}, c_{\mathsf{ce}}) &= c_{\mathsf{cr}} \lor c_{\mathsf{ce}} \\
\mathsf{context}^{\#}_{\mathsf{tc}}\ c &= \bullet
\end{aligned}$$

We note that $\mathsf{init}^{\#}_{\mathsf{tc}}$ and $\mathsf{enter}^{\#}_{\mathsf{tc}}$ return false. The reasoning behind this is that when the program begins and when a function is entered, the analysis should start with a state describing that no thread has been created. After a call, the state should be true if a thread was created by the caller before or if one was created by the callee. Thus, the $\mathsf{combine}^{\#}_{\mathsf{tc}}$ function performs a locical "or" operation on both states and returns the result.

With this analysis, we can improve the $\mathsf{combine}^{\#}_{\mathsf{tID}}$ of the thread ID analysis. Before, this function was defined as $\mathsf{combine}^{\#}_{\mathsf{tID}}\ (D_{\mathsf{cr}}, D_{\mathsf{ce}}) = D_{\mathsf{ce}}$. We change this, so this analysis only returns the callee state when a thread was possibly created in the call and otherwise returns the caller state from before:

$$\mathsf{combine}^{\#}_{\mathsf{tID}}\ (D_{\mathsf{cr}}, D_{\mathsf{ce}}) = \begin{cases} D_{\mathsf{ce}} & \text{if } \eta_{\mathsf{tc}}\ [e_f, c] \\ D_{\mathsf{cr}} & \text{else} \end{cases}$$

for an edge $(u, f();, v)$ where c is the corresponding context.
This modification allows the analysis to keep the callee state with the respective set of encountered edges. In the example from Figure 4.1 this means, that after the call in Line 11, the state from before with no encountered edges is kept. This results in no race being detected in Line 13. Furthermore, because the thread created in Line 14 is known to be unique, no race is detected in Line 4.

4.2 Implementation

We briefly described in this section, how we implemented the thread-create analysis from the previous chapter in Goblint and how we used it to improve the `threadId` analysis that already exists in the analyzer.

Similar to the `taintPartialContexts` analysis from Subsection 3.2.1, we implement a new module `threadCreate` that implements the interface seen in Figure 3.2. The implementation is very similar to our formal definition of this analysis from the previous chapter. In the following we discuss some noteworthy points:

Domain:
The Domain `D` for this analysis only contains the two boolean values, where `false` stands for "definitely no thread was created" and `true` for "maybe a thread was created". For this GOBLINT provides the `MayBool` domain implementing this notion with the corresponing ordering.
Similar to the taint analysis, this analysis does not contribute to the overall context of the analyzer, and thus we chose `Unit` for the module `C`.

Analysis functions:
We handle thread creation in two different functions:
One is the `threadspawn` function, that exists for exactly this purpose. We implement this function to always returns `true` for the thread create analysis, as this corresponds to a create edge.
The other function where we handle thread creations is the `special` function. We implement it so that it returns `true` not only when the special function that is called is a thread creating function, e.g., `pthread_create()`, but also when it is an unknown function. Since we know nothing about unknown functions, we cannot exclude that such a function creates a thread. In any other case besides thread creating and unknown functions, the `special` function propagates the state from before.

All other analysis functions either return the state from before (e.g., `assign` and `return`) or they are implemented exactly as defined in Section 4.1 (e.g. `combine` returns the result of a locical "or" on the caller and callee state).

The `query` function:
To allow the new thread-create analysis to broadcast its information to other analyses we add a `MayThreadCreate` query. This query is answered by the thread-create analysis with the current state. Other analysis can send a `MayThreadCreate` query to gain the information, whether a thread has been created from the beginning of the current function up to the point where the query is sent. In the case, that the `threadCreate` analysis is not performed, this query is answered with `true` by default, i.e., "A thread *may* have been created".

Improving the `threadId` analysis
In Goblint there exists an implementation of the thread ID analysis we introduced in Section 4.1, namely the `threadId` analysis. This analysis was always performed context-sensitively. Thus, we add the option `ana.thread.context.createEdges` to the analyzer. This option is enabled by default, but it allows for removing the set of encountered create edges from the context when it is disabled. We implement this by changing the `context` function of the `threadId` analysis, so that it checks this option.
To implement our proposed improvements, we change the `combine` function of the `threadId` analysis according to our proposal from Section 4.1. It now sends a `MayThreadCreate` and returns the caller state when the answer is `false`. Otherwise, the callee state is returned like before our changes.

5 Evaluation

In this chapter we evaluate our approach. This is split into two parts: First we test, whether our implementation is sound, i.e., it does not lead to wrong results. After that we benchmark the implementation on some real world C programs. The goal of the benchmark is to get a perspective on the precision and computation time of our proposed improvements from Chapter 3 and Chapter 4. Thus, we compare context-insensitive runs with our improvement to context-insensitive ones without the improvement and to context-sensitive runs. In particular, we want to know how much of the precision lost by context insensitivity can be recovered through our proposed improvements.

5.1 Testing

The aim of this section is to ensure, that the addition of the `taintPartialContexts` analysis to GOBLINT as described in Section 3.2 does not lead to wrong results. Additionally, we want to make sure that just activating the taint analysis on a fully context-sensitive analysis run does not lead to less precise results.

GOBLINT provides an extensive set of regression test cases already. These test edge cases of various features of the analyzer. Each test case comes with a specific configuration, that should be used when executing the test.
To verify our implementation, we ran all regression test cases with their specified configuration, but additionally activated the taint analysis. This helps to ensure that no precision is lost, just by activating the taint analysis. These test runs helped to find some bugs, which then were fixed.
For the `threadCreate` analysis we used the same approach, where we ran all regression tests with the thread-create analysis additionally activated.
We also contributed a few new regression tests which specifically test edge cases of the taint and the thread-create analysis. These include tests for each existing analysis, where we included either of our two new analyses, as well as tests for bugs we cleared to show the correct behavior after the fix. The new regression tests also aim to demonstrate how our changes improve the precision of existing analyses.

Additionally, we investigate the results of the benchmarks we describe in the following section. These reinforce the verdict that the taint analysis and the thread-create analysis are sound in their implementation.

5.2 Benchmarking the improved `base` analysis

In this section we investigate, if and to what extent the changes we proposed in Subsection 3.2.2 provide an improvement. In particular, we compare analyses with different configurations in terms of precision and computation time. However, we focus on benchmarking the changes to the `base` analysis as this is the main variable analysis used in GOBLINT. We do perform benchmarks with the other variable analyses we improved in Subsection 3.2.2 besides the `base` analysis.

5.2.1 SV-Benchmarks

The first benchmarking approach we will describe in this section uses the SV-Benchmarks' "Collection of Verification Tasks" [sos]. This collection of verification tasks is "constructed and maintained as a common benchmark for evaluating the effectiveness and efficiency of state-of-the-art verification technology" [sos]. Each verification task consists of a program and a corresponding specification, i.e., a set of properties. The verifier to be benchmarked, i.e., in our case the GOBLINT analyzer with our changes, then performs an analysis run on a given program. After that it is checked, whether the verifier was able to proof the given set of properties. The properties we focus on for our benchmarks are the following:

- `unreach-call`: A specified function `reach_error()` in the program is never called during runtime.
- `no-overflow`: No integer overflow occurs in the program.
- `no-data-race`: The program contains no race condition.

For each of these properties, we perform three benchmark runs with different configurations on each program for which it is specified that the property holds. The three configurations we use for the benchmark runs are:

- *sens*: The `base` analysis is performed context sensitively.
- *insens*: The `base` analysis is performed context insensitively.
- *insens taint*: The `base` analysis is performed context insensitively with the improvements of the taint analysis.

During each benchmark run with a certain configuration we save different kinds of information to compare later. The most important of which are the computation time per program and the total number of programs, for which the property to prove was proven.
We first investigate the precision. For this we compare the number of programs for which the property was proven per configuration and property: The raw numbers can be found in Table 5.1, while the graph in Figure 5.1 shows a visualization of the results.

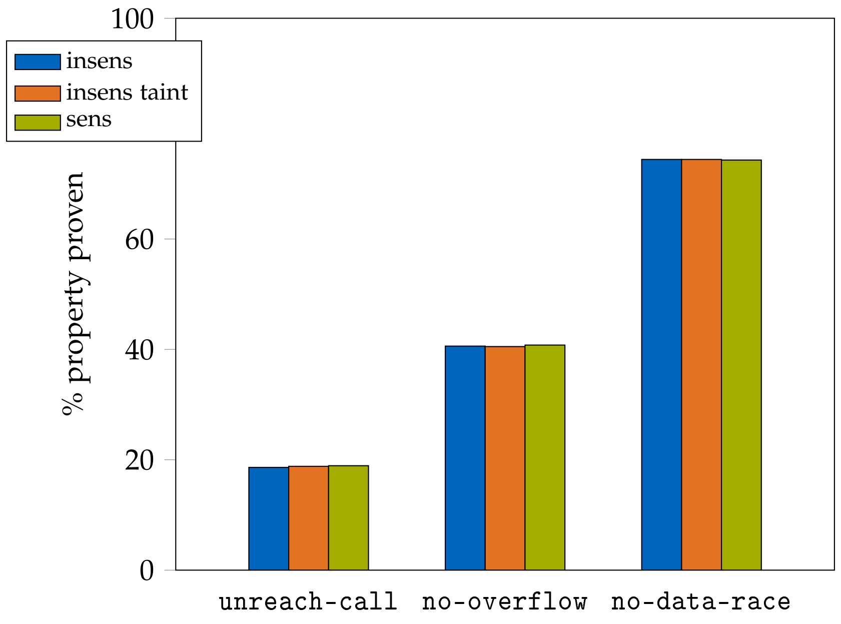

Figure 5.1: Bar graph for the comparison of the three configurations per property. A bar represents the percentage of programs, for which the corresponding property was proven using the corresponding configuration

As we can see in Figure 5.1, there is in fact not a big difference between the three configurations for each property. Our interpretation of this result is, that the choice between our three configurations only has a small effect on the precision of GOBLINTwhen analyzing real world programs. In other words this means, that not a lot of precision is lost by performing the `base` analysis context insensitively. The taint analysis can only recover lost precision, so little over all precision loss means, that the taint analysis can only provide little over all improvement.

We now take a closer look at Table 5.1. First we talk about the `unreach-call` property:

property	*config.*	*# proven*	*# total*	*# loss by insens*	*# recovered*	*% recovered of loss*
`unreach-call`	*sens*	1925	10173	31	20	64.52
	insens	1894				
	insens taint	1914				
`no-overflow`	*sens*	3456	8474	16	-17	-43.75
	insens	3440				
	insens taint	3433				
`no-data-race`	*sens*	671	903	-1	0	0.00
	insens	672				
	insens taint	672				

Table 5.1: Table showing the results of the SV-Benchmarks

We see, that with the *insens* configuration, the analyzer was able to proof the property 31 times fewer than with the *sens* configuration. However, it proved the property 20 times more often with the *insens taint* configuration compared to the *insens* configuration. This leads us to the conclusion, that for this property the taint analysis helped to recover about 65% of the precision lost.
As for the results of the `no-overflow` property, it seems like the usage of the taint analysis leads to less precise results. This property was proven for 17 fewer programs, when using the *insens taint* configuration instead of the *insens* configuration. In fact these results helped us find a bug in our implementation. This bug is now fixed and the `no-overflow` property can now be proven for these 17 programs with the *insens taint* configuration. In conclusion however, we cannot say that the taint analysis provided an improvement for proving the `no-overflow` property.
Similarly, the *insens taint* configuration is exactly as precise as the `insens` configuration for the `no-data-race` property. The reason, that the *sens* configuration proved the property for one fewer program than both of the other configurations is a timeout. Considering this, we come to the conclusion, that for this property all three configurations are equally precise.

<TODO: TIMINGS!!!>

property	*config.*	*cputime all (s)*	*cputime correct (s)*	*# Timeouts*
`unreach-call`	*sens*	1.940.000	67.600	1712
	insens	1.910.000	64.000	1563
	insens taint	1.920.000	65.800	1658
`no-overflow`	*sens*			
	insens			
	insens taint			
`no-data-race`	*sens*			
	insens			
	insens taint			

5.2.2 GNU Coreutils

We also benchmark the improved `base` analysis with another approach. For this we use modified versions of C programs from the GNU Core Utilities "coreutils" [GNU]. These programs implement "the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system."[GNU]
Combined versions of the coreutil programs are found in a benchmark repository dedicated for benchmarking the Goblint analyzer [Gobb]. A "combined version" of a program is a code file, where all dependencies of included files of the program are merged into one single code file.
For these combined programs we use a feature from the Goblint analyzer itself to generate assertions at different points within the program. An assertion is an equality or inequality involving program variables, that holds for every concrete execution of the program. To generate these, the analyzer performs an analysis with a given configuration on a given program. Goblint then uses the information it gains to place assertions which it knows are true in the program and produces an output file.
We can then use the resulting file with generated assertions to compare other analysis runs with different configurations. The metric for these comparisons is the number of proven assertions. It is necessary that the configuration which is used to generate the assertions is at least as precise as the most precise configuration of the runs we want to compare. Only the can this benchmarking approach to produce meaningful results.
For this reason we generate assertions with a configuration that performs the `base` analysis context sensitively. In order to do this we followed an approach similar to Julian Erhard, where he generated asserts for theses programs context insensitively. A detailed description of his approach can be found in [Erh].
We compare the same configurations we compared in Subsection 5.2.1.

program	*config.*	*# asserts proven*	*# total asserts*	*% loss by insens*	*% recovered of total*	*% recovered of loss*
`cksum_comb.c`	*sens*	1998	1998	1.30	0.00	0.00
	insens	1972	1998			
	insens taint	1972	1998			
`cut_comb.c`	*sens*	3992	3992	0.30	0.00	0.00
	insens	3979	3992			
	insens taint	3979	3992			
`dd_comb.c`	*sens*	4462	4464	2.97	0.49	16.54
	insens	4337	4472			
	insens taint	4359	4472			
`df_comb.c`	*sens*	8834	8834	12.68	0.05	0.36
	insens	7714	8834			
	insens taint	7718	8834			
`du_comb.c`	*sens*	9810	9810	3.41	0.05	1.50
	insens	9464	9798			
	insens taint	9469	9798			
`nohup_comb.c`	*sens*	3397	3397	0.77	0.00	0.00
	insens	3371	3397			
	insens taint	3371	3397			
`ptx_comb.c`	*sens*	5786	5786	4.46	0.07	1.55
	insens	5528	5786			
	insens taint	5532	5786			
`tail_comb.c`	*sens*	4806	4806	0.50	0.00	0.00
	insens	4782	4806			
	insens taint	4782	4806			

Table 5.2: Table showing the results of the coreutils with generated assertions

The results of the benchmark runs on the coreutil programs can be seen in Table 5.2. We omit the results for two further programs `cp_comb.c` and `mv_comb.c` because for these the analyzer proved some assertions to evaluate to "false" with the *insens* and *insens taint*. This hints at bugs in the analyzer which are unrelated to our changes.
The numbers of total assertions differs between the *sens* and both *insens (taint)* configurations for some programs. The reason for this is, that an assertion in a function can be checked multiple times, once for each context the function is evaluated with. Therefore, we compare percentages for this benchmark.
We make the following observations: The percentage of the precision loss that is recovered reaches over 16% for one program while it stays below 2% for all other programs. For three of the eight programs shown, no improvement was achieved at all. From this we conclude that it depends a lot on the program which analyzed, how much the taint analysis can improve the precision loss.
We also note that the "*% recovered of total*" is rather low. That is the percentage of assertions, which were found with the *insens taint* but not the *insens* configuration, never surpasses 0.5%. Therefore, we conclude that the effect of our changes on the overall precision of the analyzer is very limited.

5.3 Benchmarking the improved `threadId` analysis

We only performed a partial benchmark for our improvement of the `threadId` analysis using the thread-create analysis. For this we again used the GNU coreutil programs, which we described in Subsection 5.2.2. This time however, instead of comparing the number of correctly proven assertions, we compare the number of race conditions, the Goblint analyzer finds in each program with each configuration. We compared these configurations:

- *sens*: The `threadId` analysis is performed context sensitively.
- *part-sens*: The `threadId` analysis is performed partially context sensitively, i.e., only sensitive with respect to the thread ID but not with respect to the set of encountered create edges. Concretely, the option `ana.thread.context.createEdges` we added is disabled in this configuration.
- *part-sens taint*: Same configuration like *part-sens* but with the improvements of the thread-create analysis enabled.

The results of this benchmark are as follows: For each of the coreutil programs, the analyzer finds between 5 and 8 race conditions. However, there is no difference between the three configurations. This means that for each program, the number of

race conditions found is the same irregardless of the configuration chosen. This means that with this benchmarking setup we do not record any precision loss that results from partial context-sensitivity. Since no precision is lost, we cannot see, if the thread-create analysis provides an improvement.
In conclusion there definitely exist cases, where the running the `threadId` partially context-sensitive leads to precision loss, which the thread-create analysis can reduce. We have shown this by adding a regression test of one such case to the GOBLINT regression tests.
However, such cases may only exist very rarely in real world programs.
It has to be noted that our benchmark for this analysis was only very small in scale. More extensive benchmarks are necessary to make a more meaningful statement. The next step in the future is to perform a benchmark run of the SV-Benchmarks' `unreach-call` property as described in Subsection 5.2.1.

6 Conclusion

- **Summary:**
- Source of Precison loss identified
- Two ideas for reducing the precision loss for two types of analyses
- implemented in GOBLINT and tested to prove viability
- benchmarked to check if a noticeable improvement is achieved
- **Conclusion:**
- => result: little improvement, but not time costly???
- -> future: more extensive benchmark for thread Create
- -> future: Not that great???

Abbreviations

CFG Control flow Graph

lval Left Value (of an assignment)

List of Figures

List of Tables

Bibliography

[Api14] K. Apinis. "Frameworks for analyzing multi-threaded C." PhD thesis. Technische Universität München, 2014.

[ASV12] K. Apinis, H. Seidl, and V. Vojdani. "Side-effecting constraint systems: a swiss army knife for program analysis." In: *Asian Symposium on Programming Languages and Systems*. Springer. 2012, pp. 157–172.

[Cil] Cil. *C Intermediate Language (CIL)*. `https://github.com/goblint/cil`. Last Accessed: 2023-02-01. Goblint.

[Cou21] P. Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[Erh] J. Erhard. *sv-benchmarks/c/goblint-coreutils/README.txt*. `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/9930354ef471336665f94a756e07755b5` `c/goblint-coreutils/README.txt`. Last Accessed: 2023-02-01.

[GNU] GNU. *Coreutils - GNU core utilities*. `https://www.gnu.org/software/coreutils/`. Last Accessed: 2023-02-01. GNU Operating System.

[Goba] Goblint. *Goblint - A static analyzer for multi-threaded C programs, specializing in finding concurrency bugs*. `https://goblint.in.tum.de/home`. Last Accessed: 2023-02-01. Goblint.

[Gobb] Goblint. *Goblint benchmark suite*. `https://github.com/goblint/bench/tree/57cfb3d93e3414527d5a24d30af48242a1e6f1eb`. Last Accessed: 2023-02-01. Goblint.

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

[Sch+23] M. Schwarz, S. Saan, H. Seidl, J. Erhard, and V. Vojdani. "Clustered Relational Thread-Modular Abstract Interpretation with Local Traces." In: *arXiv preprint arXiv:2301.06439* (2023).

[sos] sosy-lab. *SV-Benchmarks*. `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/3d3b5d7d91b78f49e6cc3874002d23fdc4b3d1b4`. Last Accessed: 2023-02-01. sosy-lab.