SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Combatting the Precision Loss of Partial Contexts in Abstract Interpretation

Felix Sebastian Krayer

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Combatting the Precision Loss of Partial Contexts in Abstract Interpretation

Bekämpfung des Präzisionsverlust durch partielle Kontexte in Abstrakter Interpretation

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

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

Munich, 15th of February 2023 Felix Sebastian Krayer

Acknowledgments

Abstract

Contents

1 Introduction

WIP:
- introduce Goblint
- show problem on a small example

Related work

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

2 Background

2.1 Static Analysis

Static analysis is defined by Rival [RY20] as "[...]an automatic technique that approximates in a conservative manner semantic properties of programs before their execution". This means that the program is analyzed just by the given source code without execution. The goal is to prove certain properties about the program in a "sound" manner i.e. any property that is proven to hold actually does hold. However, from failing to prove a property one cannot conclude that the given property does not hold.
In order to prove properties, e.g. finding that a program does not contain races or identifying dead code, we need to gain information about the program. This is done by performing various kinds of analyses. We will focus on flow sensitive analyses from now on i.e. analyses which find properties of the program dependent on the location within it. We will introduce a syntax to formalize flow sensitive analyses in the following sections. This formalization approach is heavily based on [ASV12].
// TODO: talk about Abstract Interpretation.

```
1 int main() {
2   int x;
3   x = 0;
4   if (x == 0)
5     x = x + 1;
6 }
```

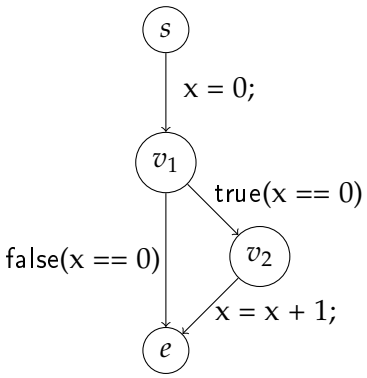

Figure 2.1: Example program (left) and corresponding CFG (right)

2.1.1 Flow sensitive analysis

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

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

2.1.2 Constraint systems

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

constraint

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

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

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

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

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

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

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

2.1.3 Interprocedural analysis

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

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

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

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

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

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

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

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

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

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

2.1.4 Context sensitivity

//WIP:
The context insensitive approach in the previous chapter is not very precise.

- example: procedure that simply increments a global variable *a* by one, called with different values for *a*

2.1.5 Partial context sensitivity

2.1.6 Precision loss

- example from above expanded with a second global variable that is not altered but experiences precision loss

3 Combatting Precision Loss

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

3.1 Formal description

3.1.1 Taint analysis

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

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

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

From that follows that we seek to compute a mapping from program points to sets of variables i.e. $\eta_t : N \to \mathbb{D}_t$. To interpret this with the goal of our taint analysis in mind, we note that $\eta_t[n] = T$ will denote that T is the set of possibly tainted variables at program point $[n]$. Expressed differently this means that for any variable $x \in T$ we cannot exclude that this variable was altered between the start of the procedure $[n]$ is in up until the program pint $[n]$.
It remains to define $\mathsf{init}_t^\#$, $\mathsf{enter}_t^\#$ and $\mathsf{combine}_t^\#$ 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 fresh whenever we enter a procedure and start without any variable being initially tainted. Since the same holds for the initial state we have

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

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

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

Note that we removed the callee local variables $\mathit{Locals}_\mathsf{ce}$ because these are not accessible by the caller and all of its callers anyway, so it is not useful to keep track of them.
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 is a variable assignment. For these we note that this 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. In the following chapter we will see how this information helps us to improve the values-of-variables analysis.

3.1.2 Improving the values-of-variables analysis

Recall the source of the precision loss we want to reduce. 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 above, we now do have the information which

variables can be altered by a procedure $f()$ and which surely stay untouched. These 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:

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

where for an edge $(u, f();, v)$ we have $T_{\mathsf{ce}} = \eta_{\mathsf{t}}\ [e_f]$.
Similar to before the $\mathsf{combine}^{\#}_{\mathsf{v}}$ function takes the caller mapping, restricts is 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 this change the partitionig was done strictly in such a way that the local variables were taken from the caller state and all global variables from the callee state. After this change, 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 this 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

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

From $x \notin \eta_{\mathsf{t}}\ [e_f]$ we know that x has not been altered in the procedure $f()$ since the node $[s_f]$, and therefore it holds that

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

By the definitions of $\sqsupseteq_{\mathsf{v}}$ and $\mathsf{enter}^{\#}_{\mathsf{v}}$ we get:

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

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

3.2 Implementation

We will quickly introduce the Goblint analyzer and its structure before we explain the process of implementing the proposed taint analysis as well as its usage to improve other analyses. 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: After the C input file is preprocessed, a CFG is generated. This 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 that is a tuple of all the domains of 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 parametrized domains which can be used in the specifications of the analyses. It is also shown that multiple analyses are then combined into one MCP that is then used with the CFG to generate constraints which are solved.

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 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. After that some functions are required:

- `name` to uniquely refer to an analysis.
- `startstate` to define the state used when entering the analysis (similar to $\mathsf{init}^\#$).
- *Transfer functions* which define the abstract effects of actions (similar to $[\![A]\!]^\#$)
- *Functions for interprocedural analysis*
- *Function for analysis of multithreaded programs*

For our taint analysis we created 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.

Domain

The next step was to choose `D` and `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 C 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 things 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. In Goblint there is a concept incorporating all of them called Left Value (of an assignment) (lval). To be as precise as possible we will use a set of lvals instead of a set of variables.
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 set X poses no issue, in a concrete implementation this is extremely unpractical 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)` 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 is inherently context insensitive. Therefore, the context-sensitive part of our analysis `C` empty, which is expressed with the `Unit` domain provided by goblint.

Transfer functions

However analyzing the full language C instead of our toy language, it is necessary to take care of further challenges like pointers and library or unknown functions.
- Goblint provides MayPointTo()
- Goblint provides library descriptors -> unknown = top

3.2.2 Benefiting other analyses

- main part: base Analysis:

- mappings of lvalues new to the caller are taken from callee e.g. newly allocated memory on callee
- mappings not in callee state but are in caller context need to be removed -> in multithreaded programs, if in the caller a mutex was held but unlocked by the callee
- other: keep values from caller if untainted. Lvalues present in the tainted are overwritten with values from callee by folding over tainted set.

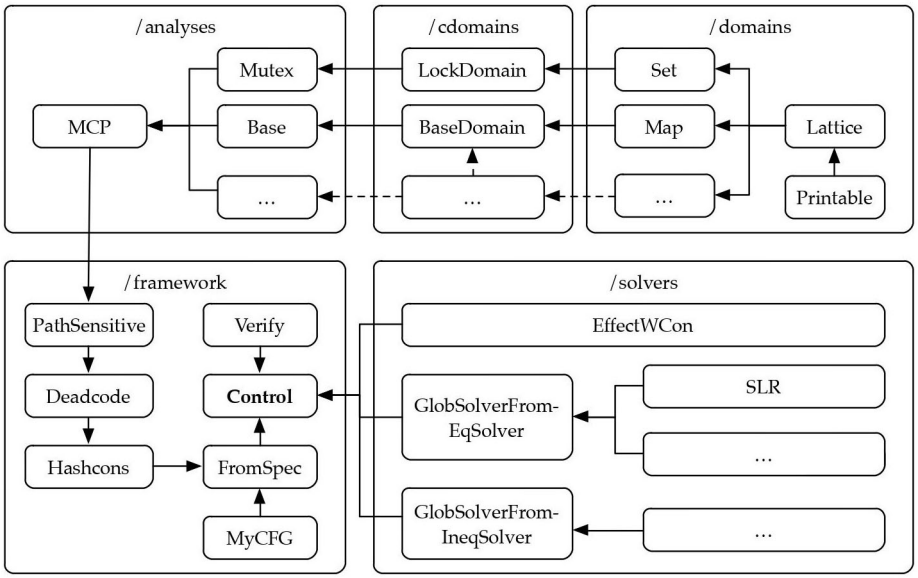

Figure 3.1: Schematic directory structure of Goblint. Adapted from [Api14]

- relation analysis (apron) benefited in a similar way
- mention varEq and condVars for completion??

– Full New Section: ThreadCreate analysis

```
1  module type Spec =
2  sig
3    (* Domain *)
4    module D : Lattice.S
5    module C : Printable.S
6
7    val name : unit -> string
8    val startstate : varinfo -> D.t
9
10   (* Transfer functions *)
11   val assign: (D.t, C.t) ctx -> lval -> exp -> D.t
12   val vdecl : (D.t, C.t) ctx -> varinfo -> D.t
13   val branch: (D.t, C.t) ctx -> exp -> bool -> D.t
14   val query : (D.t, C.t) ctx -> 'a Queries.t -> 'a Queries.result
15
16   (* Functions for interprocedural analysis *)
17   val special : (D.t, C.t) ctx -> lval option -> varinfo -> exp list -> D.t
18   val enter : (D.t, C.t) ctx -> lval option -> fundec -> exp list -> (D.t * D.t) list
19   val return : (D.t, C.t) ctx -> exp option -> fundec -> D.t
20   val combine : (D.t, C.t) ctx -> lval option -> exp -> fundec -> exp list -> C.t option -
21
22   val context : fundec -> D.t -> C.t
23
24   (* Function for analysis of multithreaded programs *)
25   val threadenter : (D.t, C.t) ctx -> lval option -> varinfo -> exp list -> D.t list
26   val threadspawn : (D.t, C.t) ctx -> lval option -> varinfo -> exp list -> (D.t, C.t) ctx
27 end
```

Figure 3.2: Simplified Interface for implementing analyses in Goblint

4 Evaluation

4.1 Testing

- soundness checked with regression tests from GOBLINT

4.2 Benchmarking

- coreutil as benchmarking programs
- various analysis runs performed with goblint: ctx insensitive with and without taint, precision compared, checks passing compared.

5 Conclusion

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.

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