SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Weakly-Relational Pointer Analysis for the Goblint Abstract Interpreter

Rebecca Ghidini

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Weakly-Relational Pointer Analysis for the Goblint Abstract Interpreter

Schwach-relationale Pointer-Analyse für den abstrakten Interpreter Goblint

Author:	Rebecca Ghidini
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Julian Erhard, M.Sc.
Submission Date:	15.09.2024

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

Munich, 15.09.2024 Rebecca Ghidini

Acknowledgments

TODO

Abstract

C is widely used in safety-critical systems, therefore it is important to have static analysis tools that can automatically identify program properties and thus help in the search for bugs. Here, we introduce C-2PO, a weakly-relational pointer analysis that can infer relationships between pairs of pointer terms in C programs, including pointer arithmetic and dereferencing. It is based on the 2-Pointer Logic introduced by Seidl et al. [Sei+24a] and is extended here with the concept of *block disequalities* that express the confinement of pointers to single memory objects in C. Abstract operations are introduced that allow the implementation of C-2PO in an abstract interpretation framework, including the operations *meet*, *join*, *widening*, and *narrowing*, as well as the abstract effects for assignments and interprocedural analysis. Two versions of the *join* and *equal* operation are introduced and compared.

C-2PO is implemented in the Goblint static analyzer. For 95% of SV-COMP tasks, the slowdown incurred by adding C-2PO is below a factor of 3. To measure precision, coreutil programs were instrumented with assertions computed by C-2PO. For an existing non-relational pointer analysis, 80% of the assertions are out of reach.

Contents

1 Introduction

Using automatic analysis tools for software is essential in order to ensure its reliability, especially in safety-critical applications. Many of these safety-critical applications are developed in C, which has strong low-level capabilities but lacks inherent safety features. C allows for complex pointer manipulations, which can introduce subtle bugs that are difficult to detect. To help find these bugs, static analysis tools can automatically analyze the behavior of programs and prove properties about programs without executing them.

Current static analysis techniques for pointers infer different properties and vary in precision and efficiency. There are fast but relatively imprecise methods, such as those proposed by Steensgaard [Ste96] and Andersen [And94], as well as more precise approaches that determine, for each program point and variable, the set of potential addresses it can point to. Examples of such methods are implemented in tools like Mopsa [MOM21] and Goblint [Voj+16]. On the other hand, there exist highly precise analyses capable of accurately modeling the shape and content of dynamic data structures in the heap [ILR21; KSV10; DPV13]. However, these methods are less efficient and thus not suitable for large-scale programs.

Here, we introduce a relational pointer analysis that can infer relationships between pointer variables in C programs, including pointer arithmetic and dereferencing.

Example 1.1. Consider a C program with multiple possible implementations for the functions `malloc` and `free`. A pair of these functions is defined in a struct `alloc_functions`:

```
struct alloc_functions {
 void *(*malloc)(size_t);
 void (*free)(void *);
};
```

Assume there are two variables `malloc_f` and `free_f` that point to the currently used implementation of the functions `malloc` and `free`, respectively.

An interesting property to confirm is whether the two variables `malloc_f` and `free_f` point to matching implementations, thus ensuring that the correct `free` function is called for each `malloc` call. This prevents bugs that arise from incompatible implementations of the two functions. Our analysis can prove the absence of such bugs by

inferring the proposition by showing that the two variables belong to the same struct `alloc_functions`, i.e.,

$$\texttt{free\_f} = 64 + \texttt{malloc\_f}.$$

In order to not have to deal with the complexity of relationships between an arbitrary number of variables, our analysis is *weakly*-relational [Min01; Sei+24b], as it only considers relationships between pairs of variables. The primary advantage of *weakly*-relational analyses lies in their simplicity, which allows for a polynomial-time implementation at the cost of losing some of the precision with respect to other relational analyses.

Here, we design a weakly-relational pointer analysis and implement it in Goblint. The analysis focuses on C programs and is based on 2-Pointer Logic [Sei+24a], thus it is called C-2PO. At each program point, it infers a set of propositions that consistently hold across all possible executions. These inferred properties are expressed using the 2-Pointer Logic introduced by Seidl et al. [Sei+24a], consisting of equalities and disequalities formed from pairs of terms, which may include variable names, dereferencing, and pointer arithmetic. An example of such a property is

$$(x = \&y + 3) \wedge (*(\&z + 2) \neq \&y + 1), \tag{1.1}$$

where $*$ is the dereferencing operator, & is the address-of operator and x, y, z are program variables.

In addition to the propositions introduced by Seidl et al. [Sei+24a], this thesis introduces the concept of *block disequalities*, which express that two pointers refer to different memory objects that we call *blocks*. Memory objects are, for example, distinct variables or separate memory blocks allocated by `malloc`. For two variables x and y, we know that their addresses $\&x$ and $\&y$ belong to different memory blocks, allowing us to infer that $bl(\&x) \neq bl(\&y)$, where $bl(p)$ denotes the memory block of the pointer p. Moreover, for two variables x and z that are initialized with different `malloc` calls, we can infer that $bl(x) \neq bl(z)$.

Analyzing disequalities and block disequalities alongside equalities is essential for achieving a precise analysis during assignments. When assigning a value to a dereferenced pointer $*p$, the operation overwrites a value in memory, meaning that all values stored at the same address as p are altered. As a result, any propositions involving values that may be equal to p become invalid. The equalities inferred by C-2PO are must-equalities, which are guaranteed to hold at a specific program point, but they do not indicate whether two pointers *may* point to the same address. Disequalities, on the other hand, help determine which pointers *may not* point to the same address, thus allowing us to retain more information after an assignment. Additional disequalities are inferred from the block disequalities and by leveraging information from other non-relational pointer analyses. It is easily possible to implement the collaboration

of the different analyses, as we implemented the C-2PO analysis in Goblint, which already includes a non-relational pointer analysis.

1.1 Thesis Structure

The thesis is structured as follows:

- Chapter 2 gives an overview of the abstract interpretation method, which builds the theoretical foundation on which the C-2PO analysis is designed. It also includes a brief description of the abstract interpreter Goblint where the C-2PO analysis is implemented, and some Goblint analyses that collaborate with C-2PO to augment the precision of the analysis.
- Chapter 3 introduces a concrete semantics that models C pointers, the memory content, and how program statements modify the values of pointers and the memory.
- Chapter 4 recalls the 2-Pointer Logic introduced in [Sei+24a] and extends it with block disequalities. A representation of the conjunctions is introduced, which automatically discovers the *closure* of the conjunction, i.e., the set of all propositions that are implied by the conjunction. For example the proposition $*(\&z + 2) \neq x - 2$ is implied by the conjunction (1.1). The implied conjunctions are computed using the *quantitative congruence closure* of all the equalities that are present in the conjunction and the closure of the disequalities, using the rule that when $*t_1 \neq *t_2$, this implies that the pointers are also different, i.e., $t_1 \neq t_2$.
- Chapter 5 describes the abstract domain used in the analysis and the corresponding operations that are needed to implement an abstract interpretation analysis, including the *equal*, *meet*, *join*, *widening* and *narrowing* operations. Two possible algorithms of the *equal* and the *join* operation are presented, which differ in their precision and efficiency. Transfer functions are described, which define the behavior of the analysis during an assignment and a function call.
- Chapter 6 evaluates the implementation of the C-2PO analysis in Goblint on a set of benchmarks, comparing the analysis with the existing non-relational pointer analysis in Goblint, and also evaluating the differences between the two algorithms for the *join* operation and the two possibilities for the *equal* operation.
- Chapter 7 gives an overview of the related work on congruence closure, pointer analysis, and weakly-relational analyses.

- Chapter 8 concludes the thesis and gives an outlook on future work.

The implementation of C-2PO is available under . Parts of this thesis have been submitted to [Ghi+24]. TODO

2 Background

The C-2PO analysis is based on the abstract interpretation method. Before going into the details of the structure of the C-2PO analysis, it is helpful to explain the fundamental concepts of abstract interpretation and how it can be used to design static program analyses. This chapter also presents the Goblint abstract analyzer, where C-2PO was implemented, as well as existing non-relational pointer analysis *MayPointTo* and the analysis of tainted variables *MayBeTainted*, which are implemented in Goblint. They will later be used in collaboration with the C-2PO analysis to increase precision.

2.1 Abstract Interpretation

Abstract interpretation is a theoretical framework used to infer the properties of programs that are proven to be correct for all possible executions of the program. It was first introduced by Patrick and Radhia Cousot in 1977 [CC77; Cou21], and it has since been widely used in the fields of static analysis [RY20] and compiler construction [HH12]. Static analysis by abstract interpretation consists in abstracting the concrete semantics of a program with an abstract over-approximation of the program's behavior. As the abstract semantics is an approximated version of the concrete semantics, it can be computed more efficiently.

2.1.1 Basic Principles

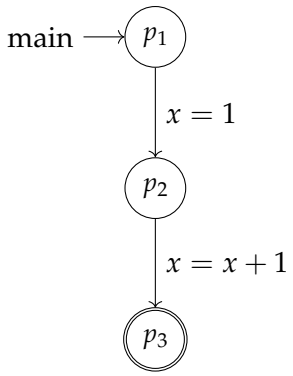
During the analysis, the program is represented by a control flow graph (CFG), where each node represents a program point, and each edge represents a statement of the program, such as assignments, conditional branches, and function calls. Each node p in the CFG is associated with a concrete state $C[p]$ that describes the values of the variables and the memory at that program point. Fig. 2.1a shows an example C program and Fig. 2.1b shows its corresponding CFG.

The concrete edge effect $[\![l]\!]$ describes the alteration of the concrete state after an edge with label l. For example in Fig. 2.1b, the concrete state at node p_2 is $C[p_2] = \{x \to 1\}$. The state of node p_3 is equal to $C[p_3] = [\![x = x + 1]\!]\, C[p_2] = \{x \to 2\}$.

The abstract interpretation method consists of computing an abstract over-approximation of the concrete semantics at each node in the CFG with an abstract description of the

```c
int main(){
  int x = 1;
  x = x + 1;
}
```

(a) A simple C program.



(b) A simple control flow graph.

Figure 2.1: An example of a C program and its corresponding CFG.

state. Each analysis chooses an abstract domain that can characterize specific properties of interest of the program. For a node p in the CFG, the abstract state is denoted as $X[p]$. For example, the abstract domain of the C-2PO analysis is a conjunction of equalities between pointers. One conjunction is computed at each program point, over-approximating all possible concrete states. Another simpler example would be an abstract domain that tracks whether each variable is even or odd. An abstract state that only contains the variable x is represented as $\{x \to \text{even}\}$, $\{x \to \text{odd}\}$, or $\{x \to \text{unknown}\}$ if the parity of x is not known.

The abstract edge effect $[\![l]\!]^\sharp$ describes the effect that an edge with label l has on the abstract state. It must be an over-approximation of the corresponding concrete effect $[\![l]\!]$. For example, in the CFG in Fig. 2.1b, the abstract state at node p_2 is $X[p_2] = \{x \to \text{odd}\}$. After the edge with label $x = x + 1$, the abstract state at node p_3 is $X[p_3] = [\![x = x + 1]\!]^\sharp\, X[p_2] = \{x \to \text{even}\}$.

The abstract domain is often required to be a complete lattice, meaning that there is a partial order denoted by $\sqsubseteq$ between the domain elements, and for every subset of elements, there must be a least upper bound (or join, denoted by $\sqcup$) and a greatest lower bound (or meet, denoted by $\sqcap$). However, some domains do not have a *least* upper bound for each subset. In this case, an over-approximation in the form of an upper bound can be used instead without losing the correctness of the analysis. The same holds for the greatest lower bound. Complete lattices also have a top element ($\top$) and a bottom element ($\bot$), representing the lattice's least and greatest elements, respectively. During the analysis, the value $\top$ represents an abstract state with no information about any variable, while $\bot$ represents an unreachable program point.

The concrete and the abstract semantics are related by the concretization function γ and the abstraction function α. The concretization function maps an abstract element

to the set of all concrete elements represented by the abstract element. For example, the concretization of $\{x \to \text{even}\}$ is the set of all concrete states where x is an even number:

$$\gamma(\{x \to \text{even}\}) = \{\{x \to 0\}, \{x \to 2\}, \{x \to 4\}, \ldots\}$$

The abstraction function maps a concrete element to the most precise abstract element that represents the concrete element. For example, $\alpha(\{x \to 0\}) = \{x \to \text{even}\}$.

The functions γ and α form a Galois connection, which ensures the following property for each abstract state a and each edge label l:

$$[\![l]\!]\, \gamma(a) \subseteq \gamma([\![l]\!]^\sharp\, a).$$

This property guarantees that the abstract edge effect $[\![l]\!]^\sharp$ is an over-approximation of the concrete edge effect $[\![l]\!]$. If this property can be proven for each l, then we know that computing the semantics of a program in the abstract domain is a sound over-approximation of the concrete semantics.

2.1.2 Abstract Effects of Program Statements

The analysis of a program via abstract interpretation is done by traversing the CFG and computing the abstract state at each node. We begin from the entry point of the main function in the CFG, which has the initial abstract state $\top$. Then, each edge is traversed, and the abstract effects of the edges are applied to the abstract state. The analysis must sometimes re-visit nodes multiple times when the abstract state of the predecessor of that node changes. The abstract effects of each type of edge are described in the following.

Assignment

The abstract effect of an assignment $[\![x = e]\!]^\sharp$ computes the change in the abstract state when assigning the expression e to the variable x. For example, if x and e are pointers, the C-2PO analysis removes all previous propositions involving x since the value of x might have changed. Then, it adds the equality $x = e$ to the abstract state.

Conditionals

When there is an if statement with the condition c, the CFG diverges into two paths, one for the *true* branch (denoted by $Pos(c)$) and one for the *false* branch (denoted by $Neg(c)$). The analysis can compute the abstract state a_c corresponding to the condition c and then meet the previous abstract state a with a_c, resulting in the state $a \sqcap a_c$. The analysis does the same for the false branch with the abstract state corresponding to $\neg c$.

For example, if the condition is $x = y$, the C-2PO analysis adds the equality $x = y$ in the positive branch and the disequality $x \neq y$ in the negative branch.

Loops

The CFG represents for-loops and while-loops as edges that lead from the loop's end back to its beginning. The abstract value of a program point with multiple incoming edges is the least upper bound of the set of all abstract states of the incoming edges. Formally, for each node p, the abstract state $X[p]$ is defined by the constraint

$$X[p] \sqsupseteq \bigsqcup \{ [\![l]\!]^\sharp\, X[p] \mid \text{there is an edge from } p \text{ to } p' \text{ in the CFG labeled with } l \}.$$

The values $X[p]$ are computed by solving the system of constraints generated at each program point by this definition. According to the Knaster-Tarski theorem, each monotonic function f on a complete lattice has a least fixpoint, equal to the least solution of the inequality $x \sqsubseteq f(x)$. Therefore, the system of inequalities can be solved by starting with the bottom element $\bot$ for each program point and iteratively applying the abstract semantics until the computation reaches a fixpoint. However, in some cases, this approach does not converge. This happens when the abstract domain has infinite ascending chains and the analysis computes a sequence of states where each element is strictly greater than the previous one, without reaching a fixpoint.

Widening and Narrowing

Sometimes, while traversing a loop, the analysis recomputes the abstract state of a program point infinitely many times without converging to a stable solution. A widening operator (∇) is used instead of the join operator in these cases. The widening operator over-approximates the join operator, and it is designed to ensure the termination of the analysis of loops at the cost of some precision loss.

Complementary to the process of widening is the narrowing operation (Δ), which is designed to refine the result obtained by widening in order to restore some of the precision lost by the widening operator. The narrowing operation over-approximates the meet operation to partially regain the lost information by ensuring the termination of the computation.

Interprocedural Analysis

Before diving into the abstract effects of a function call, we discuss a concrete semantic for function calls. The concrete semantics of a function depends not only on the function definition but also on the values of the parameters, global variables, and the

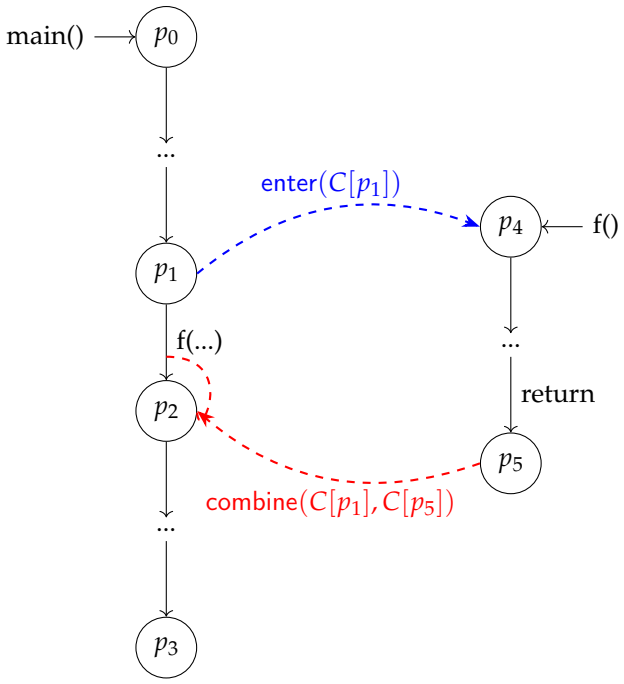


Figure 2.2: Visualization of the CFG of a program during a function call.

state of the memory at the time when the function is called. Therefore, a separate CFG is created for each possible value that the parameters and memory can have. An edge is added from the caller CFG to the function's entry point, labeled with the enter function, and one from the entry point of the function to the caller CFG, labeled with the combine function. These edges are visualized in Fig. 2.2, where p_4 is the first node of the function, and p_5 is the final node.

When traversing the edge from the function call to the function's entry point, the enter function transforms the caller's concrete state to the initial state for the function analysis. The concrete state returned by enter contains only global variables, the function parameters, and the function's local variables. The caller's local variables are removed by enter. Then, each node in the CFG of the function is analyzed as usual, starting from the initial state returned by enter.

At the end of the function, the combine function merges the previous concrete state of the caller and the function's final state. It is used when traversing the edge from the returning point of the function to the caller. It keeps the information about the caller's local variables and forgets the function's local variables. The values of the global variables and the memory content are equal to the state after the function call.

The abstract analysis of function calls is handled by the abstract enter function enter$^\sharp$

and the abstract combine function $\mathsf{combine}^\sharp$. They are an over-approximation of the corresponding concrete functions. As for the concrete semantics, the abstract CFG has an edge from the node before the function call to the entry point of the function, labeled $\mathsf{enter}^\sharp$, and an edge from the exit point of the function to the node after the function call, labeled $\mathsf{combine}^\sharp$.

Let s be the abstract state before the function call. The start state of the function is $\mathsf{enter}^\sharp\, s$. A separate CFG is generated for each possible abstract value at the beginning of the function. Let the final state of the function be s'. After the function call, the state is $\mathsf{combine}^\sharp\, (s, s')$.

2.2 Goblint

The Goblint tool is an abstract interpretation framework for analyzing C programs. It allows the definition of custom abstract domains by specifying the lattice operations and all abstract edge effects. Then, Goblint can analyze C programs by transforming them into a set of CFGs, generating the system of inequalities arising from each edge, and solving these constraints using appropriate fixpoint algorithms. There are already several abstract domains implemented in Goblint. These include relational and non-relational value analyses, concurrency and mutex analyses, as well as out-of-bounds access analyses.

The different analyses can communicate through *queries*, which allows the combination of the results of different analyses, leading to an increase in the precision of each analysis. The C-2PO analysis utilizes information from two existing analyses of Goblint: the pointer analysis and the analysis of tainted variables.

Pointer Analysis

The pointer analysis in Goblint is a non-relational value analysis that tracks for each expression the set of possible addresses it can take. The query MayPointTo determines the set of possible addresses an expression can refer to. It is a may-analysis, meaning that the result is an over-approximation of the possible addresses, and it is impossible for the expression to point to an address that is not in the result of MayPointTo. On the other hand, the C-2PO analysis is a must-analysis, meaning that each proposition is definitely true for the current state. However, there might be even more true propositions that the analysis did not find. The MayPointTo analysis complements the C-2PO analysis by determining additional disequalities between terms. Two terms are not equal if the MayPointTo sets of the two terms have an empty intersection.

Tainted Variables Analysis

An additional analysis used in the C-2PO analysis is the tainted variables analysis. Here, a *tainted* variable is defined as a variable that was overwritten with a new value during the execution of a function. The query MayBeTainted returns an over-approximation of the set of tainted variables in the current function. C-2PO uses this information to remove the tainted variables from the abstract state of the caller when returning from the function. This analysis is necessary to remove only the changed values from the caller state, thus allowing us to keep the information about all the caller variables that were not modified during the function call.

3 Concrete Semantics

This chapter will describe the model of concrete program states used in this thesis and its semantics during assignments and function calls.

The concrete state consists of the memory content and the values of the variables. The memory is modeled as a two-dimensional address space $\mathbb{Z} \times \mathbb{Z}$, where each address is a pair of a *block identifier* and an *offset* within the block. Each call to malloc returns an address with a fresh block identifier, and each variable is in a unique memory block of the memory, distinct from the block of any other variable.

We call *auxiliaries* the variables whose address is never taken in the current program, therefore we know of these variables that their address cannot be reached from other terms using address arithmetic or dereferencing. Let $\mathcal{X}$ and $\mathcal{A}$ be disjoint finite sets of non-auxiliary variables and auxiliaries. A concrete state is composed of the following three functions:

- $\rho : \mathcal{X} \to \mathbb{Z} \times \mathbb{Z}$ assigns an address to each variable,
- $\nu : \mathcal{A} \to \mathbb{Z} \times \mathbb{Z}$ assigns an address to each auxiliary, and
- $\mu : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ assigns a value to each address.

We are only interested in analyzing the values of pointers and addresses. Therefore, we interpret each value stored in the memory and the auxiliaries as an address.

In order to define the effects of program statements, we first need to describe the meaning of C expressions. C-2PO considers only expressions whose C type is a pointer type and which are equivalent to a term used in the 2-Pointer Logic [Sei+24a]. This logic is the basis of the C-2PO analysis and will be described in Chapter 4. The terms we consider are formed from auxiliaries and addresses of program variables, using constant offsets and dereferencing. They are defined by the following grammar:

$$t ::= A \mid \&x \mid *(z + t),$$

where $A \in \mathcal{A}$ is an auxiliary, $x \in \mathcal{X}$ is a variable with address $\&x$ and $z \in \mathbb{Z}$ is an integer. We call terms $\&x$ or A *atoms*.

A term t in the C code is interpreted as the value $[\![t]\!](\rho, \nu, \mu)$ defined by:

$$\begin{array}{lcl} [\![\&x]\!](\rho,\nu,\mu) & = & \rho\, x, \\ [\![A]\!](\rho,\nu,\mu) & = & \nu\, A, \\ [\![*(z+t)]\!](\rho,\nu,\mu) & = & \mu\,(z + [\![t]\!](\rho,\nu,\mu)), \end{array}$$

where the operator $(+) : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ on addresses is defined as $z + (a, b) = (a, z + b)$. Here, z is added to the address offset, thus modeling that it is impossible to leave a memory block using address arithmetic.

3.1 Assignment

We consider assignments of the form $s_1 := s_2$ where s_1 is either an auxiliary $V \in \mathcal{A}$ or a pointer term $*(z + t)$, while s_2 is either the symbol ?, which represents an unknown value, or is of the form $z_1 + t$ for some term t, or malloc, which assigns an address from a fresh block to s_1. A *fresh* address block a is an address block identifier that does not occur as a mapping in any of the functions ρ, ν, or μ, i.e., for each result (a', b') of the functions ρ, ν, or μ, we have $a \neq a'$.

For a set H of states (ρ, ν, μ), we define the concrete semantics of the assignment $s_1 := s_2$ as the transformation $[\![s_1 := s_2]\!]\, H$, where:

$$\begin{array}{l} [\![V := ?]\!]\, H = \\ \quad \{(\rho, \nu \oplus \{V \mapsto a\}, \mu) \mid (\rho, \nu, \mu) \in H, a \in \mathbb{Z} \times \mathbb{Z}\} \\ [\![*(z+t) := ?]\!]\, H = \\ \quad \{(\rho, \nu, \mu \oplus \{(z + [\![t]\!](\rho, \nu, \mu)) \mapsto a\}) \mid (\rho, \nu, \mu) \in H, a \in \mathbb{Z} \times \mathbb{Z}\} \\ [\![V := z_1 + s]\!]\, H = \\ \quad \{(\rho, \nu \oplus \{V \mapsto z_1 + [\![s]\!](\rho, \nu, \mu)\}, \mu) \mid (\rho, \nu, \mu) \in H\} \\ [\![*(z+t) := z_1 + s]\!]\, H = \\ \quad \{(\rho, \nu, \mu \oplus \{(z + [\![t]\!](\rho, \nu, \mu)) \mapsto z_1 + [\![s]\!](\rho, \nu, \mu))\} \mid (\rho, \nu, \mu) \in H\} \\ [\![V := \mathsf{malloc}]\!]\, H = \\ \quad \{(\rho, \nu \oplus \{V \mapsto (a, 0)\}, \mu) \mid (\rho, \nu, \mu) \in H, a \in \mathbb{Z}, a \text{ is a } \textit{fresh}\} \\ [\![*(z+t) := \mathsf{malloc}]\!]\, H = \\ \quad \{(\rho, \nu, \mu \oplus \{(z + [\![t]\!](\rho, \nu, \mu)) \mapsto (a, 0))\} \mid (\rho, \nu, \mu) \in H, a \in \mathbb{Z}, a \text{ is a } \textit{fresh}\}. \end{array}$$

Here, $f \oplus \{m \mapsto z\}$ is the mapping obtained from f by setting the value of f for m to z.

3.2 Interprocedural Semantics

Let f be a function with parameters $p_1, \ldots, p_n$. Assume that there is an edge of the CFG labeled $f(e_1, \ldots, e_n)$, and the concrete state before the function call is (ρ, ν, μ). The

initial state of the function call is defined by the function $\mathsf{enter}\,(\rho, \nu, \mu)$, which returns the state (ρ', ν', μ'), defined as follows:

- $\rho'\,x = \rho\,x$ for all global variables $x \in \mathcal{X}$. For the caller's local variables, ρ' is not defined. Additionally, for each parameter $p_i \in \mathcal{X}$ of the function, $\rho'\,p_i = [\![e_i]\!]$.
- ν' is defined correspondingly: it contains the entries of $\nu'\,A = \nu\,A$ for all global variables $A \in \mathcal{A}$, and it is not defined for the caller's local variables. For each auxiliary parameter $p_i \in \mathcal{A}$ of the function, $\nu'\,p_i = [\![e_i]\!]$.
- $\mu'\,(a, b) = \mu\,(a, b)$ for each address (a, b).

At the end of the function, let the resulting state be (ρ', ν', μ') and the caller state before the function call (ρ, ν, μ). The concrete semantics of the edge returning from the function to the caller is defined by $\mathsf{combine}\,(\rho, \nu, \mu)\,(\rho', \nu', \mu')$, which returns the state (ρ'', ν'', μ''):

- $\rho''\,x = \rho'\,x$ for all global variables $x \in \mathcal{X}$ and $\rho''\,x = \rho\,x$ for the local variables of the caller. The local variables of the function are not defined in ρ''.
- Correspondingly, $\nu''\,A = \nu'\,A$ for all global auxiliary variables $A \in \mathcal{X}$ and $\nu''\,A = \nu\,A$ for the local variables of the caller, and ν'' is not defined for the function's local variables.
- $\mu''\,(a, b) = \mu'\,(a, b)$ for each address (a, b).

4 2-Pointer Logic as an Abstract Domain

The abstract domain of the C-2PO analysis consists of finite conjunctions of propositions from the 2-Pointer Logic. In the following section, we describe this logic and how to represent the propositions to infer all deriving propositions. Then, we discuss the extension of the domain to include additional terms and the restriction of the domain to a subset of terms, which will be needed later for the analysis.

Seidl et al. [Sei+24a] introduced the 2-Pointer Logic, which consists of equalities and disequalities between the terms described in Chapter 3. Here, we extend this logic with an additional type of proposition, the *block disequalities*. These disequalities express that two terms do not belong to the same address block. The propositions p are defined by the following grammar:

$$p ::= t_1 = z + t_2 \mid t_1 \neq z + t_2 \mid bl(t_1) \neq bl(t_2),$$

where $*$ denotes the dereferencing operator and $bl(t)$ represents the address block of t. We use $*t$ as an abbreviation for $*(0 + t)$ and x as an abbreviation for $*(0 + \&x)$.

Moreover, we define the function $bl : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ that returns the address block identifier of an address, where $bl(a, b) = a$. The validity of a proposition under a concrete state (ρ, ν, μ) is defined as:

$$\begin{aligned}
(\rho, \nu, \mu) &\models t_1 = z + t_2 \quad \textbf{iff} \\
&\quad [\![t_1]\!](\rho, \nu, \mu) = z + [\![t_2]\!](\rho, \nu, \mu) \\
(\rho, \nu, \mu) &\models t_1 \neq z + t_2 \quad \textbf{iff} \\
&\quad [\![t_1]\!](\rho, \nu, \mu) \neq z + [\![t_2]\!](\rho, \nu, \mu) \\
(\rho, \nu, \mu) &\models bl(t_1) \neq bl(t_2) \quad \textbf{iff} \\
&\quad bl([\![t_1]\!](\rho, \nu, \mu)) \neq bl([\![t_2]\!](\rho, \nu, \mu))
\end{aligned}$$

If $(\rho, \nu, \mu) \models p$ for each proposition p in Ψ, then we say that $(\rho, \nu, \mu) \models \Psi$.

4.1 Representation of C Expressions as 2-Pointer Logic Terms

Since the C-2PO analysis only considers C expressions with pointer types, they have a uniform size. For instance, on an x86-64 architecture, pointers occupy 64 bits in the memory.

However, the offsets added to pointers vary depending on the pointer's type. For example, if we have a pointer p of type $int32_t * p$, the expression $p + z$ represents an offset of $32 \cdot z$ bits. If the same pointer p is cast to a pointer of type $int64_t * p$, the offset z is of $64 \cdot z$ bits. It is essential to consider this to know the relative positioning of different expressions in the memory and to discover possible overlapping data. Therefore, in our implementation, the offsets in the abstract terms representing C expressions are expressed in bits rather than the value present in the code. For instance, if p is a pointer to a 32-bit integer and q is a pointer to a 64-bit integer, then the analysis represents the C expression $*(p + 1)$ as $*(p + 32)$, and $*(q + 1)$ as $*(q + 64)$.

In some cases, it is impossible to determine the offset of a pointer expression, e.g., if the offset is an unknown variable instead of a constant. Here, the Goblint constant propagation analysis is used to infer the offset value. If the value remains indeterminable, the expression is treated as unknown.

Furthermore, arrays and structs are handled similarly to pointers in C, allowing us to express properties not only about pointers but also about arrays and structs. For example, in C, the expression $a[1]$ is equivalent to the expression $*(a + 1)$. Hence, the analysis treats arrays as pointers to their first element. A two-dimensional array in C with dimensions n and m can be viewed as a one-dimensional array with $n \cdot m$ elements. The expression $a[i][j]$ is treated as $*(a + (i \cdot m + j) \cdot s)$, where s is the size in bits of the array elements. Similarly, arrays with more dimensions are treated as pointers to a one-dimensional array.

For a struct s, the analysis treats $\&s$ as a pointer to the first field of the struct. For a field f of the struct s with an offset of z bits, the expression s.f is treated as $*(\&s + z)$. In C, the offset of a struct element may not always be a multiple of 8 bits. Thus, offsets are represented in bits rather than bytes. The analyzer Goblint can determine the bit offsets of fields within a struct in most cases. C-2PO leverages Goblint to find these offsets, and in the rare instances where Goblint cannot determine the exact offset, the expression is treated as unknown.

Example 4.1. Consider this C struct:

```
struct S {
    int32_t first;
    int32_t arr[5];
};
```

Let x be a variable of type struct S. The expression x.arr[4] is treated as $*(\&x + 160)$. Inside the struct, the offset of arr is 32 and the offset of $arr[4]$ is $32 \cdot 4 = 128$. In total, the offset is $32 + 128 = 160$.

4.2 Quantitative Congruence Closure

Given a conjunction Ψ of propositions, we want to find all equalities logically implied by Ψ. As infinitely many such equalities exist, we only consider the ones containing terms exclusively from a specific set $\mathcal{T}$. The set $\mathcal{T}$ must be subterm-closed and contain at least all the terms in Ψ. The equalities implied by Ψ containing terms from $\mathcal{T}$ are described by the quantitative equivalence relation $\equiv_{\Psi,\mathcal{T}}$, which is the smallest equivalence relation $\equiv_{\Psi,\mathcal{T}}$ satisfying these rules:

[E0] If $t_1 = z + t_2$ occurs in Ψ, then $t_1 \equiv_{\Psi,\mathcal{T}} z + t_2$;

[E1] $t \equiv_{\Psi,\mathcal{T}} 0 + t$ for all $t \in \mathcal{T}$ (*quantitative reflexivity*);

[E2] If $t_1 \equiv_{\Psi,\mathcal{T}} z + t_2$, then $t_2 \equiv_{\Psi,\mathcal{T}} -z + t_1$ (*quantitative symmetry*);

[E3] If $t_1 \equiv_{\Psi,\mathcal{T}} z_1 + t_2$ and $t_2 \equiv_{\Psi,\mathcal{T}} z_2 + t_3$, then $t_1 \equiv_{\Psi,\mathcal{T}} (z_1 + z_2) + t_3$ (*quantitative transitivity*);

[E4] If $t_1 \equiv_{\Psi,\mathcal{T}} (z_2 - z_1) + t_2$, then $*(z_1 + t_1) \equiv_{\Psi,\mathcal{T}} *(z_2 + t_2)$ holds as well, whenever $*(z_1 + t_1)$, $*(z_2 + t_2)$ are in $\mathcal{T}$ (*dereferencing*).

4.2.1 Quantitative Partition

We can represent the quantitative equivalence relation $\equiv_{\Psi,\mathcal{T}}$ as a partition. The *quantitative partition* $P = (\mathcal{T}, \tau, \omega)$ consists of a set of terms $\mathcal{T}$, a function $\tau : \mathcal{T} \to \mathcal{T}$ that assigns a *representative* to each term $t \in \mathcal{T}$ and a mapping $\omega : \mathcal{T} \to \mathbb{Z}$, which assigns to each term its offset from the representative of its equivalence class. These two functions model equalities of the form $t \equiv_{\Psi,\mathcal{T}} \omega\, t + \tau\, t$ for each $t \in \mathcal{T}$. All terms with the same representative belong to the same equivalence class, i.e., they are equivalent up to an integer offset.

We denote as $P[\Psi]$ the quantitative partition that model the conjunction Ψ. The set of representative terms $rep(P[\Psi])$ is the set of terms $t \in \mathcal{T}$ that occur on the right-hand side of the function τ, i.e., there exists a $t' \in \mathcal{T}$ such that $\tau\, t' = t$. For each $t \in rep(P[\Psi])$, it holds that $\tau\, t = t$ and $\omega\, t = 0$.

The quantitative partition $P = (\mathcal{T}, \tau, \omega)$ is useful to define a normal form for propositions. Given an equality $p \equiv t_1 = z + t_2$ and a partition $P = (\Pi, \tau, \omega)$, the normalized equality $norm_P(p)$ is an equivalent equality that only contains representatives of P, i.e., $\tau\, t_1 = z - \omega\, t_1 + \omega\, t_2 + \tau\, t_2$. Equivalently, the normalized disequality $norm_P(p)$ for $p \equiv t_1 \neq z + t_2$ is the disequality $\tau\, t_1 = z - \omega\, t_1 + \omega\, t_2 + \tau\, t_2$. The normalized block disequality $norm_P(p)$ for $p \equiv bl(t_1) \neq bl(t_2)$ is the disequality $bl(\tau\, t_1) \neq bl(\tau\, t_2)$.

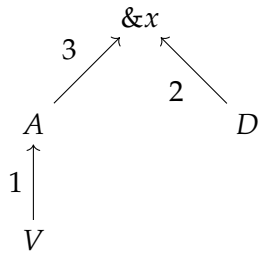


Figure 4.1: Union-find tree corresponding to the conjunction $(V = 1 + A) \wedge (D = 2 + \&x) \wedge (A = 3 + \&x)$

Implementation

The quantitative partition is represented using a data structure similar to union-find but extended with integer offsets. The union-find data structure is represented as a forest of trees, each representing an equivalence class. Each node in the tree corresponds to a term $t \in \mathcal{T}$, and the roots are representative elements $\tau\, t$.[Tar79] All the edges in the tree are directed towards the root. In order to represent the integer offset, each edge is labeled with an integer weight.

The forest is represented as a map $\mathit{parent} : \mathcal{T} \to \mathbb{Z} \times \mathcal{T}$ that maps each term t to an integer offset and its parent node in the tree. The nodes that have themselves as a parent are the representative terms. The mapping $\mathit{parent}(t_1) = (z, t_2)$ defines an edge from t_1 to t_2 with weight z in the graph, and it implies the equality $t_1 \equiv_{\Psi,\mathcal{T}} z + t_2$.

In order to compute the representative of a term in the tree, there is an operation $\mathsf{find}(t)$. It returns the pair $(\omega\, t, \tau\, t)$. The offset ωt can be computed by following the path from t to the tree's root and summing up the weights of the encountered edges. The representative $\tau\, t$ is the root of the tree containing t.

The union-find tree is built incrementally by adding consecutively equalities. Initially, each term $t \in \mathcal{T}$ is its own parent, and equalities $t_1 = z + t_2$ between terms can be added by performing the $\mathsf{union}(t_1, z, t_2)$ operation. This operation merges two trees by adding an edge from $\tau\, t_1$ to $\tau\, t_2$ with weight $z - \omega\, t_1 + \omega\, t_2$, thus merging the two equivalence classes of t_1 and t_2 with the correct offset. If the two terms are already equivalent with a different offset, i.e., $\tau\, t_1 = \tau\, t_2$, but $z - \omega\, t_1 + \omega\, t_2 \neq 0$, then the conjunction is unsatisfiable.

Example 4.2. Consider the conjunction $\Psi \equiv (V = 1 + A) \wedge (D = 2 + \&x) \wedge (A = 3 + \&x)$. The tree computed by the union-find algorithm is illustrated in Fig. 4.1. From this representation, we can, for example, derive that $V = 4 + \&x$. The tree can look different depending on the choice of the representatives.

In a practical implementation, the representatives are chosen such that the trees have the lowest possible height, thus making the $\mathsf{find}(t)$ operation more efficient. This optimization is called *union by rank* and consists of adding the edge from the smallest to the largest equivalence class during the $\mathsf{union}(t_1, z, t_2)$ operation. Another possible optimization is, during the $\mathsf{find}(t)$ operation, to set the parent node of each traversed node to its representative. This way, the tree is flattened, which makes the $\mathsf{find}(t)$ operation more efficient if we call it again for the same term t or a term in the same equivalence class. This is the so-called *path compression*. [Tar79]

4.2.2 Quantitative Finite Automaton

A partition $P = (\mathcal{T}, \tau, \omega)$ groups all terms that are equal up to an integer offset. The partition takes into account the reflexivity, symmetry, and transitivity rules. Further, it is necessary to ensure that the equalities deriving from [E4] are always considered. Therefore, for each term t, we need to keep track of the equivalence classes that contain terms of the form $*(z + t)$ and merge two equivalence classes if they contain terms that are equal according to the rule [E4]. For example, if $\mathcal{T} = \{A, V, *A, *V\}$ and we perform a union of A and V, the closure rule [E4] tells us that also the equivalence classes of $*A$ and $*V$ should be merged. Thus, we introduce an alternative representation of the partition P as a *quantitative finite automaton* (QFA) M.

The QFA is defined by a triple $M = (S, \overline{\tau}, \eta, \delta)$. S is a finite set of states, each representing an equivalence class of the partition. In order to know which state represents which equivalence class of the partition, the mapping $\overline{\tau} : S \to \mathcal{T}$ assigns a representative term to each state. If $\overline{\tau}\, s = t$ for a state $s \in S$, then s represents the equivalence class of all terms whose representative in P is t. The partial mapping $\eta : (\mathcal{A} \cup \{\&x | x \in \mathcal{X}\} \to \mathbb{Z} \times S)$ provides initial offsets and states for atoms, meaning that if $\eta\, a = (z, s)$, then a is in the equivalence class represented by s. $\delta : \mathbb{Z} \to S \to \mathbb{Z} \times S$ is the partial transition function. If $\delta\, z_1\, s_1 = (z_2, s_2)$, this means that it holds that $*(z_1 + \overline{\tau}\, s_1) \equiv_{\Psi,\mathcal{T}} z_2 + \overline{\tau}\, s_2$.

The QFA $M = (S, \overline{\tau}, \eta, \delta)$ is constructed from a partition $P = (\mathcal{T}, \tau, \omega)$ as follows:

- S contains a state s for each representative term t of P and we set $\overline{\tau}\, s = t$.
- $\eta\, a = (\omega\, a, s)$ if $s \in S$ and $\overline{\tau}\, s = \tau\, a$
- $\delta\, z\, s_1 = (\omega\, t', s_2)$ if there is a $\tau\, t' = \overline{\tau}\, s_2$, such that $t' \equiv *(z_1 + t_1)$ with $\tau\, t_1 = \overline{\tau}\, s_1$ and $z = z_1 + \omega\, t_1$.

We remark that following this definition, a transition $\delta\, z\, s_1$ could derive not only from a single term t' but also from a second term $*(z_2 + t_2) \in \mathcal{T}$ with $\tau\, t_2 = \overline{\tau}\, s_1$ and $z = z_2 + \omega\, t_2$. However, we show that the resulting mapping $\delta\, z\, s_1$ would have

the same value in this case. Since $t_i \equiv_{\Psi,\mathcal{T}} (\omega t_i) + (\tau Q_i)$ for $i = 1, 2$, it follows that $t_1 \equiv_{\Psi,\mathcal{T}} (z_2 - z_1) + t_2$. Therefore $\tau * (z_2 + t_2) = \overline{\tau}\, s_2$ with offset $\omega(*(z_2 + t_2)) = \omega\, t'$. This shows that the result of $\delta\, z\, s_1$ is well defined and does not depend on which term t' we choose for deriving a transition. Moreover, we note that δ is defined only for a finite amount of values, given that each term $*(z + t) \in \mathcal{T}$ defines at most one mapping for δ.

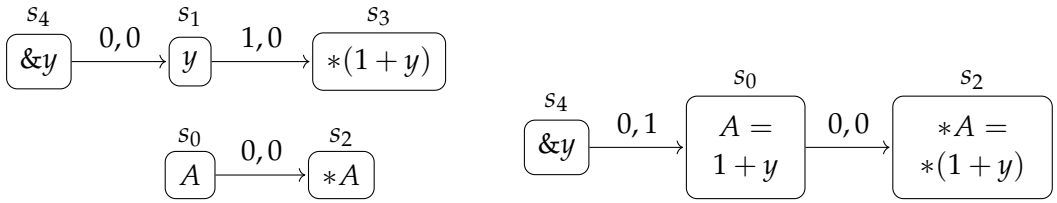
We use $\overline{\mathcal{T}}$ to denote the set of *all* terms with variable names from $\mathcal{X}$ and auxiliaries from $\mathcal{A}$. We can extend the mappings η and δ to a partial mapping $M : \overline{\mathcal{T}} \to \mathbb{Z} \times S$ where $M[a] = \eta(a)$ for atoms a, and $M[*(z + t_1)] = \delta(z + z_1, s)$ for terms t_1 if $M[t_1] = (z_1, s)$.

Furthermore, we define the set $\mathcal{L}(M)$ of terms $t \in \overline{\mathcal{T}}$ for which M is defined. For each state $s \in S$, the set $\mathcal{L}_M(s)$ is the set of terms $t \in \overline{\mathcal{T}}$ for which it holds that $M[t] = (z, s)$ for some $z \in \mathbb{Z}$.

Using the automaton, we define the $\mathsf{closure}(P, t_1 = z + t_2)$ operation that modifies the partition $P = (\mathcal{T}, \tau, \omega)$ and the automaton $M = (S, \overline{\tau}, \eta, \delta)$ to include the equality $t_1 = z + t_2$ and then computes the closure of the rule [E4]. The resulting partition $P' = (\mathcal{T}, \tau', \omega')$ and automaton $M' = (S', \overline{\tau}', \eta', \delta')$ are computed as follows:

- Case 1: $\tau\, t_1 = \tau\, t_2$. If $\omega\, t_1 = z + \omega\, t_2$, then $P' = P$ and $M = M'$. If $\omega\, t_1 \neq z + \omega\, t_2$, then the conjunction is unsatisfiable.
- Case 2: $\tau\, t_1 \neq \tau\, t_2$. We call $\mathsf{union}(t_1, z, t_2)$. Afterwards, either t_1 or t_2 has a new representative, so the transitions of the QFA are updated accordingly. W.l.o.g., let $\tau\, t_1$ be the new representative of t_1 and t_2:
  - Let $\overline{\tau}\, s_1 = \tau\, t_1$ and $\overline{\tau}\, s_2 = \tau\, t_2$. After the union, the state s_2 will not be in S', as the equivalence classes of s_1 and s_2 are merged. Therefore, we need to move all outgoing transitions of s_2 to be transitions of s_2. For each transition $\delta(z_2, s_2) = (z_3, s_3)$, we set $\delta'(z_1, s_1) = (z_3, s_3)$, where $z_1 = z_2 - \omega\, t_1 + \omega\, t_2 - z$, if $\delta(z_1, s_1)$ is not already defined.
  - If $\delta(z_1, s_1) = (z_4, s_4)$ is defined, then the equivalence classes of s_3 and s_4 need to be merged later. We add $(\overline{\tau}\, s_3 = \omega\, (\overline{\tau}\, s_4) - \omega\, (\overline{\tau}\, s_3) + \overline{\tau}\, s_4)$ to a queue Q of pending equalities.
  - For each incoming transition $\delta(z_5, s_5) = \delta(z_2', s_2)$, we set $\delta'(z_5, s_5) = (z_2' - z_1, s_2)$. The remaining transitions of δ are added to δ' without modifications.
  - Then we call $\mathsf{closure}(P', t_1 = z + t_2)$ for each equality $t_1 = z + t_2 \in Q$.

This algorithm is similar to congruence closure [DST80; Sho06], but it is restricted to a unary uninterpreted function symbol $*$ and extended with integer offsets.



(a) A QFA representing the empty conjunction. (b) The result of closure$(P, A = 1 + y)$.

Figure 4.2: An example of the closure$(P, A = 1 + y)$ operation on a QFA over the set of terms $\mathcal{T} = \{\&y, A, y, *A, *(1 + y)\}$. The representative of each equivalence class is the first mentioned term.

Example 4.3. A QFA can be visualized by labeling the states with the terms of $\mathcal{T}$ that are in the corresponding equivalence class. For example, consider the set of terms $\mathcal{T} = \{\&y, A, y, *A, *(1 + y)\}$. The edges are labeled with the offsets: for an edge $\delta(s_1, z_1) = (s_2, z_2)$, the edge between s_1 and s_2 is labeled with z_1, z_2.

The initial automaton, that represents the empty conjunction, is the one in Fig. 4.2a. If the operation closure$(P, A = 1 + y)$ is called, first the union$(A, 1, y)$ is performed and the new representative of y becomes A, with offset 1. It would be equivalent to choose y as the new representative, but we choose A here as an example. Then, we want to move the outgoing transitions from s_1 to s_0, as the state s_1 is removed. However, there already exists a transition $\delta(s_0, 0) = (s_2, 0)$. Therefore, we add the equality $*(1 + y) = *A$ to a queue of pending equalities. Then we recursively call closure$(P', *(1 + y) = 0 + *A)$ and thus perform the union of $*(1 + y)$ and $*A$. As before, we choose $*A$ as the new representative, but $*(1 + y)$ would also be a possible choice. The resulting automaton is visualized in Fig. 4.2b.

Theorem 4.4. *Assume that Ψ is a satisfiable conjunction of equalities, let $\mathcal{T}$ be a subterm-closed set of terms which contains all terms occuring in Ψ. The corresponding quantitative partition $P = (\mathcal{T}, \tau, \omega)$ and a corresponding QFA $M = (S, \overline{\tau}, \eta, \delta)$ are constructed by applying the closure operation with all equalities in Ψ. Then $\mathcal{T} \subseteq \mathcal{L}(M)$ and for every $t_1, t_2 \in \mathcal{L}(M)$ the following statements are equivalent:*

1. $M[t_1] = z + M[t_2]$,
2. $\tau\, t_1 = \tau\, t_2$ *and* $\omega\, t_1 = z + \omega\, t_2$,
3. Ψ *implies* $(t_1 = z + t_2)$.

This theorem was proven in [Sei+24a] for the case of a one-dimensional memory model. The proof can easily be adapted for the two-dimensional memory model.

We remark that the only equalities that can be derived from Ψ are derived from the equalities in Ψ. We cannot derive equalities from any of the disequalities.

4.3 Block Disequalities

The 2-Pointer Logic considers not only equalities but also block and quantitative disequalities between terms. These are necessary to infer which terms are not modified when an assignment occurs. Given a conjunction Ψ of propositions and the corresponding equivalence relation $\equiv_{\Psi,\mathcal{T}}$, the block disequalities implied by Ψ can be derived by the following rules:[Sei+24a; Ghi+24]

[B0] If $bl(t_1) \neq bl(t_2)$ then $bl(t_1) \not\equiv_{\Psi,\mathcal{T}} bl(t_2)$

[B1] If $bl(t_1) \not\equiv_{\Psi,\mathcal{T}} bl(t_2)$ then $bl(t_2) \not\equiv_{\Psi,\mathcal{T}} bl(t_1)$ (*symmetry*);

[B2] If $bl(t_1) \not\equiv_{\Psi,\mathcal{T}} bl(t_2)$, $t_1' \equiv_{\Psi,\mathcal{T}} z_1 + t_1$ and $t_2' \equiv_{\Psi,\mathcal{T}} z_2 + t_2$, then also $bl(t_1') \not\equiv_{\Psi,\mathcal{T}} bl(t_2')$ (*closure under quantitative equalities*);

[B3] $bl(\&x) \not\equiv_{\Psi,\mathcal{T}} bl(\&y)$ for all distinct $x, y \in \mathcal{X}$.

The block disequalities are stored as an undirected graph B with edges $\{t_1, t_2\}$ corresponding to conjunctions $bl(t_1) \neq bl(t_2)$, where t_1 and t_2 are representative terms in the partition $P[\Psi]$. It is sufficient to only store the disequalities between representatives, i.e., the normalized disequalities $norm_{P[\Psi]}(p)$ because the block disequalities between other terms are derived from the congruence closure $P[\Psi]$. The disequalities deriving from rule [B3] are considered as *implicit* block disequalities and are therefore not stored in the set $B[\Psi]$.

If Ψ contains a block disequality $bl(t_1) \neq bl(t_2)$ and for $P[\Psi] = (\mathcal{T}, \tau, \omega)$ it holds that $\tau\, t_1 \equiv \tau\, t_2$, then the conjunction Ψ is unsatisfiable.

Example 4.5. Consider the conjunction $\Psi \equiv A = 1 + y \wedge bl(y) \neq bl(*(1 + y))$. The QFA deriving from Ψ is shown in Fig. 4.2b. As $\tau\, y = A$ and $\tau\,(*(1 + y)) = *A$, the set $B[\Psi]$ would contain the entry $\{A, *A\}$.

4.4 Quantitative Disequalities

Let Ψ be a conjunction of 2-Pointer Logic terms. We define the binary relation $\not\equiv_{\Psi,\mathcal{T}}$ which represents all quantitative disequalities that can be derived from Ψ, given the relations $\equiv_{\Psi,\mathcal{T}}$ and $bl(\cdot) \not\equiv_{\Psi,\mathcal{T}} bl(\cdot)$.[Ghi+24]

[D0] If $t_1 \neq z + t_2$ occurs in Ψ, and $t_1 \equiv_{\Psi,\mathcal{T}} z'_1 + t'_1$ and $t_2 \equiv_{\Psi,\mathcal{T}} z'_2 + t'_2$, then $t'_1 \not\equiv_{\Psi,\mathcal{T}} (z'_2 - z'_1 + z) + t'_2$ (*closure under quantitative equalities*);

[D1] If $t_1 \not\equiv_{\Psi,\mathcal{T}} z + t_2$, then $t_2 \not\equiv_{\Psi,\mathcal{T}} -z + t_1$ (*quantitative symmetry*);

[D2] If $*(z_1 + t_1) \not\equiv_{\Psi,\mathcal{T}} *(z_2 + t_2)$, then $t_1 \not\equiv_{\Psi,\mathcal{T}} (z_2 - z_1) + t_2$ (*inverse dereferencing*);

[D3] If $t_1 \equiv_{\Psi,\mathcal{T}} z + t_2$, then $t_1 \not\equiv_{\Psi,\mathcal{T}} z' + t_2$ for all $z' \neq z$;

[D4] If $bl(t_1) \not\equiv_{\Psi,\mathcal{T}} bl(t_2) \in \Psi$, then $t_1 \not\equiv_{\Psi,\mathcal{T}} z + t_2$ for all $z \in \mathbb{Z}$.

As for the block disequalities, we store a set $D[\Psi]$ of quantitative disequalities between representative terms. $D[\Psi]$ is a finite weighted directed graph D with edges (t_1, z, t_2) corresponding to all implied quantitative disequalities $t_1 \not\equiv_{\Psi,\mathcal{T}} z + t_2$, where t_1 and t_2 are representative terms in the partition $P[\Psi]$ and $\{t_1, t_2\} \notin B$ and at least one t_i is not an address expression. The set of disequalities following from rule [D3] and rule [D4] can be infinite, but we do not need to explicitly store them, as they implicitly derive from the relations $\equiv_{\Psi,\mathcal{T}}$ and $bl(\cdot) \not\equiv_{\Psi,\mathcal{T}} bl(\cdot)$, which are stored in the partition $P[\Psi]$ and the set $B[\Psi]$.

The list $D[\Psi]$ consists of all quantitative and block disequalities deriving from [D0] to [D4], but only those that contain representative terms. This set is finite because we only store equalities between different representatives and not the implicit equalities following from [D3] and [D4].

Given a set D, a disequality $t_1 \not\equiv_{\Psi,\mathcal{T}} z + t_2$ can be added by:

- Let $norm_{P[\Psi]}(t_1 \neq z + t_2) \equiv t'_1 = z' + t'_2$.
- If $t'_1 \equiv t'_2$ and $z \neq 0$, then it is unsatisfiable.
- If t'_1 and t'_2 are both address constants, then the disequality is implicit and not stored in D.
- Otherwise, add (t'_1, z', t'_2) and $(t'_2, -z', t'_1)$ to D.

The disequalities $D = D[\Psi]$ can be computed in polynomial time in the size of the formula Ψ in the following way:

- For each equality $*(z_1 + t_1) \equiv_{\Psi,\mathcal{T}} z + *(z_2 + t_2)$ for $z \neq 0$ that follows from Ψ, add $t_1 \neq (z_2 - z_1) + t_2$ to D, as described before.
- For each disequality $bl(*(z_1 + t_1)) \not\equiv_{\Psi,\mathcal{T}} bl(*(z_2 + t_2))$ that follows from Ψ, add $t_1 \neq (z_2 - z_1) + t_2$ to D, as described before. These two first rules only add a finite amount of disequalities to the set, given that each equality and each block disequality implies at most one disequality of the type $*(z_1 + t_1) \not\equiv_{\Psi,\mathcal{T}} *(z_2 + t_2)$.

- Compute the closure of the quantitative disequalities in D using rule [D2], and add the normalized disequalities to D.
- For each disequality $p \in \Psi$, add $norm_{P[\Psi]}(p)$ to D.
- Again, compute the closure of the disequalities in D using rule [D2].

Given a conjunction Ψ, a set $\mathcal{T}$ and the corresponding partition $P = P[\Psi]$, let $p \equiv t_1 \neq z + t_2$ and $norm_{P[\Psi]}(p) = t_1' \neq z' + t_2'$. Then it holds that Ψ implies p if and only if $t_1 \not\equiv_{\Psi,\mathcal{T}} z + t_2$ and if and only if one of the following holds:

1. $(t_1', z', t_2') \in D[\Psi]$;
2. $\{t_1, t_2\} \in B[\Psi]$;
3. or $t_1 \equiv_{\Psi,\mathcal{T}} z' + t_2$ for any $z' \neq z$.

We call the disequalities that follow from 1. the *explicit* disequalities and those following from 2. and 3. the *implicit* disequalities.

Example 4.6. We consider the conjunction from Example 4.5 and add a quantitative disequality. Let $\Psi \equiv (A = 1 + y) \wedge (bl(y) \neq bl(*(1 + y))) \wedge (\&y \neq 3 + *(1 + y))$. The QFA deriving from Ψ is again the one in Fig. 4.2b. We can derive the disequality between representatives $\&y \neq 3 + *A$. From the block disequality $bl(y) \neq bl(*(1 + y))$ we derive the disequality between representatives $\&y \neq A$ using the rules [D4] and [D2]. Therefore, $D[\Psi] = (\&y \neq 3 + *A) \wedge (\&y \neq A)$.

4.5 Kernel Representation of 2-Pointer Logic Conjunctions

We introduce the *kernel* representation, which groups the quantitative partition, the quantitative automaton, and the two sets of disequalities and block disequalities. The kernel is the abstract state of the analysis, and it represents the conjunction of propositions that hold in a specific program point, as well as all propositions that they imply.

A *kernel* consists of a tuple $k = \langle P, M, B, D \rangle$, where $P = (\mathcal{T}, \tau, \omega)$ is a quantitative partition, as described in Section 4.2.1. Each state of the QFA M represents an equivalence class of P, as outlined in Section 4.2.2. B and D are sets representing the explicit block and quantitative disequalities, respectively, as detailed in Sections 4.3 and 4.4. Given a satisfiable conjunction Ψ of 2-Pointer Logic propositions, the kernel representation of Ψ is denoted as $k[\Psi] = \langle P[\Psi], M[\Psi], B[\Psi], D[\Psi] \rangle$. If Ψ is unsatisfiable, the abstract state is represented as $\bot$.

It is possible to convert a kernel representation $k = \langle P, M, B, D \rangle$ with $M = (S, \overline{\tau}, \eta, \delta)$ to a formula $\mathcal{F}[k]$, that is given by

$$\begin{aligned} \mathcal{F}[k] \quad \equiv \quad & \textstyle\bigwedge_{\eta\, a=(z,s)} (a = z + \overline{\tau}\, s) \wedge \\ & \textstyle\bigwedge_{\delta\, z\, s=(z',s')} (*(z + \overline{\tau}\, s) = z' + \overline{\tau}\, s') \wedge \\ & \textstyle\bigwedge_{\{t,t'\} \in B} (bl(t) \neq bl(t')) \wedge \\ & \textstyle\bigwedge_{(t,z,t') \in D} (t \neq z + t') \end{aligned}$$

The trivial propositions of the form $t = 0 + t$ are removed from the formula and the repeated equalities.

Given a conjunction Ψ of 2-Pointer Logic propositions, the formula $\mathcal{F}[k[\Psi]]$ is equivalent to Ψ, even though it may not be syntactically identical. The formula $\mathcal{F}[k[\Psi]]$ is not unique for each Ψ, as it depends on the choice of representative terms in the quantitative partition and of the set $\mathcal{T}$ of terms.

The domain of C-2PO consists of all kernels that can be derived from a conjunction of 2-Pointer Logic propositions up to semantic equivalence, as well as the bottom element $\bot$. The top element of the domain is the kernel $k[true]$. This element has multiple possible representations, depending on the set $\mathcal{T}$ of chosen terms. The concretization $\gamma(k)$ of k is the set of all (ρ, ν, μ) with $(\rho, \nu, \mu) \models F[k]$.

4.6 Extension to a Superset of Terms

When new terms appear during the analysis, the partition P and the automaton M need to be updated to express properties about the new terms. It is not sufficient to simply add a fresh equivalence class for the new terms, as they sometimes need to be added to existing equivalence classes.

Example 4.7. Let $\Psi \equiv \&x = \&y$ and $\mathcal{T} = \{\&x, \&y, x\}$. The equivalence classes of $P[\Psi]$ are $\{\&x, \&y\}$ and $\{x\}$. If we want to add the term y to $\mathcal{T}$, we need to add it to the same equivalence class as x, as the two terms are equivalent according to the rules defined in Section 4.2.

Let $P = (\mathcal{T}, \tau, \omega)$ be a partition with the corresponding QFA $M = (S, \overline{\tau}, \eta, \delta)$ and let $\mathcal{T}'$ be a superset of $\mathcal{T}$ and closed under subterms. We define the operation $\mathrm{ext}\, k\, \mathcal{T}'$ that extends the kernel $k = \langle P, M, B, D \rangle$ to the set of terms $\mathcal{T}'$, without altering the semantics of the representation. Only P and M need to be modified, while B and D remain unchanged.

The extension is built inductively based on the structure of the terms in $\mathcal{T}'$. We first describe how to add a new atom and then consider terms of the form $*(z + t)$, where t is already a part of $\mathcal{T}$. This is repeated for all terms in $\mathcal{T}'$.

In the case that $\mathcal{T}' = \mathcal{T} \cup \{a\}$ for an atom $a \notin \mathcal{T}$, we simply add a new equivalence class with representative a to P, by including a new state s to the states S of M and defining $\tau\, a = a$, $\omega\, a = 0$, $\eta(a) = (0, s)$, $\overline{\tau}\, s = a$.

If $\mathcal{T}' = \mathcal{T} \cup \{*(z + t)\}$ for a term $t \in \mathcal{T}$, but $*(z + t) \notin \mathcal{T}$, we add a new equivalence class $\{*(z + t)\}$ to P and then determine if it needs to be merged with an existing equivalence class. We define $\tau\,(*(z + t)) = *(z + t)$ and $\omega\,(*(z + t)) = 0$. Let the resulting partition be P' and let $s \in S$ be the state of the subterm t, i.e., for which it holds that $\overline{\tau}\, s = \tau\, t$. If $\delta(z + \omega\, t, s)$ is not defined, then we add a new state s' to S and define $\delta(z + \omega\, t, s) = (0, s')$ and $\overline{\tau}\, s' = *(z + t)$. Otherwise, let $\delta(z + \omega\, t, s) = (z', s')$. Then we perform the operation $\mathsf{closure}(P', *(z + t) = z' + \overline{\tau}\, s')$. In this case, we do not need to add a new state to the automaton, as the new term is added to the existing equivalence class of s'.

4.7 Restriction to a Subset of Terms

During an assignment, it is necessary to forget information about a specific set of terms. For instance, when assigning a value to a term, all terms in the domain that this assignment might modify—those that may alias with this term—must be removed. This section explains how to restrict a kernel $k = \langle P, M, B, D \rangle$ to keep only the propositions containing terms from a specified set $\mathcal{S}$ of allowed terms.

Let $P = (\mathcal{T}, \tau, \omega)$ be a partition and its corresponding automaton $M = (S, \overline{\tau}, \eta, \delta)$ and $\mathcal{S} \subseteq \mathcal{L}(M)$ a subterm-closed set of terms. This section describes how to compute the restricted automaton $M|_{\mathcal{S}} = (S', \overline{\tau}', \eta', \delta')$ such that $\mathcal{L}(M|_{\mathcal{S}})$ only includes terms from $\mathcal{S}$. For some states $s \in S$, the representative $\overline{\tau}\, s$ might not be in $\mathcal{S}$. If this is the case, we must find a new representative $\overline{\tau}'\, s$ from the set $\mathcal{L}_M(s) \cap \mathcal{S}$. If this set is empty, the state s must be removed from the automaton.

From $M|_{\mathcal{S}}$, we will see how to derive the restricted partition $P|_{\mathcal{S}} = (\mathcal{T}', \tau', \omega')$, where $\mathcal{T}'$ contains the terms in $\mathcal{T} \cap \mathcal{S}$ and additionally all terms that appear in the codomain of the function $\overline{\tau}'$. We do not define $\mathcal{T}' = \mathcal{S}$, as $\mathcal{S}$ may be an infinite set. However, defining $\mathcal{T}' = \mathcal{T} \cap \mathcal{S}$ would not be sufficient, as there may be some states $s \in S'$ for which $\mathcal{T} \cap \mathcal{S}$ does not contain any term of $\mathcal{L}_M(s)$, i.e., $\mathcal{L}_M(s) \cap \mathcal{T} \cap \mathcal{S} = \emptyset$. If such a state s exists, it would mean that s does not have a corresponding equivalence class in $P|_{\mathcal{S}}$. To avoid this, we add for each $s \in S'$ the term $\overline{\tau}'\, s$ to $\mathcal{T}'$, ensuring that there is at least one element in the equivalence class of s.

Example 4.8. Consider the conjunction $\Psi \equiv (\&x = \&z) \wedge (*x = \&y)$. We choose $\mathcal{T} = \{\&x, \&y, \&z, x, *x\}$. The corresponding automaton M is visualized in Fig. 4.3. Let $\mathcal{S}$ be the (infinite) set of terms that do not contain the variable x. We want to compute $M|_{\mathcal{S}}$. It holds that $\mathcal{T} \cap \mathcal{S} = \{\&y, \&z\}$. The set $\mathcal{T} \cap \mathcal{S}$ does not contain any terms from

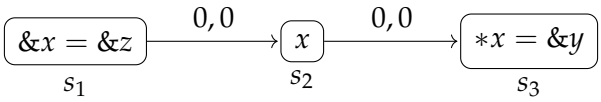


Figure 4.3: The QFA for the conjunction $(\&x = \&z) \wedge (*x = \&y)$.

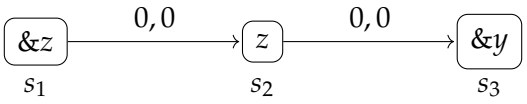


Figure 4.4: The automaton M after the restriction.

$\mathcal{L}_M(s_2) = \{x, z\}$. However, $\mathcal{L}_M(s_2)$ contains z, which is in $\mathcal{S}$, and is therefore allowed to appear in $M|_{\mathcal{S}}$. Therefore, we can choose z as the new representative $\tau' s_2$. The new set of terms after the restriction is $\mathcal{T}' = (\mathcal{T} \cap \mathcal{S}) \cup \{z\} = \{\&y, \&z, z\}$. The resulting automaton is visualized in Theorem 4.9. It represents the formula $*z = \&y$.

The restriction $M|_{\mathcal{S}} = (S', \overline{\tau}', \eta', \delta')$ is computed using a breadth-first search (BFS) on the automaton M to determine which paths remain reachable when keeping only the terms in $\mathcal{S}$. The algorithm proceeds as follows, with $\mathcal{V}$ representing the set of visited states and Q being a queue of states to be processed:

1. Start with the initial states $\eta(a) = (z, s)$ for each atom $a \in \mathcal{T} \cap \mathcal{S}$:
   - If s has not been visited, i.e., $s \notin \mathcal{V}$, set a as the representative for s: $\overline{\tau}'(s) = a$ and $\eta'(a) = (0, s)$. Add s to $\mathcal{V}$ and to Q.
   - If s has already been visited, i.e., $s \in \mathcal{V}$, there exists a t such that $\overline{\tau}'(s) = t$. Let $M[t] = (z', s)$. Define $\eta'(a) = (z - z', s)$.
2. Remove a state s_1 from the queue Q. Consider all outgoing transitions $\delta(z_1, s_1) = (z_2, s_2)$.
   - If a term $*(z + t) \in \mathcal{L}(M) \cap \mathcal{S}$ exists such that $M[t] = (z_1', s_1)$ and $z = z_1 - z_1'$, then the transition remains valid.
     - If $s_2 \in \mathcal{V}$, then $\overline{\tau}'(s_2)$ is already defined. Otherwise, set $\overline{\tau}'(s_2) = *(z + t)$ and add s_2 to $\mathcal{V}$ and to Q.
     - Now let $M[\overline{\tau}'(s_1)] = (z_3, s_1)$ and $M[\overline{\tau}'(s_2)] = (z_4, s_2)$. Define the transition $\delta'(z_1 - z_3, s_1) = (z_2 - z_4, s_2)$.
   - If no such term exists, the transition does not appear in $M|_{\mathcal{S}}$, and s_2 is not added to the set of visited states.

3. If Q is not empty, return to step 2. and continue with the next state in Q.

Essentially, the algorithm retains the transitions between the states that remain reachable using only terms from $\mathcal{S}$ and adjusts the weights of these transitions to the new representatives. The set S' equals the set of visited states $\mathcal{V}$.

The new partition $P|_{\mathcal{S}} = (\mathcal{T}', \tau', \omega')$ is defined as follows:

- $\mathcal{T}' = (\mathcal{T} \cap \mathcal{S}) \cup \{t \mid \exists s \in S'. \overline{\tau}'\, s = t\}$,
- For each $t \in \mathcal{T}'$, if $M|_{\mathcal{S}}\,[t] = (z, s)$, then $\tau'\, t = \overline{\tau}'\, s$ and $\omega'\, t = z$.

Theorem 4.9. *Let $M = (S, \overline{\tau}, \eta, \delta)$ be a QFA and let $M|_{\mathcal{S}} = (S', \overline{\tau}', \eta', \delta')$ be the automaton that is obtained by restricting M as described above. Then,*

1. *for each term $t \in \mathcal{L}(M|_{\mathcal{S}})$, it holds that $M[t] = (z, s)$ and $M[\overline{\tau}'\, s] = (z', s')$ iff $M|_{\mathcal{S}}\,[t] = (z - z', s)$ and $s = s'$,*
2. *for each term $t_1, t_2 \in \mathcal{L}(M|_{\mathcal{S}})$, it holds that $M[t_1] = z + M[t_2]$ iff $M|_{\mathcal{S}}\,[t_1] = z + M|_{\mathcal{S}}\,[t_2]$.*

Proof. The first point can be proven by induction over the structure of the term t. Point 2 follows directly from point 1. □

(Block-)Disequalities

Given a kernel $k = \langle P, M, B, D\rangle$, we can restrict it to a set of terms $\mathcal{S}$ by setting $k|_{\mathcal{S}} = \langle P|_{\mathcal{S}}, M|_{\mathcal{S}}, B|_{\mathcal{S}}, D|_{\mathcal{S}}\rangle$. $B|_{\mathcal{S}}$ and $D|_{\mathcal{S}}$ represent the block disequalities and quantitative disequalities that are still valid when considering only the terms in $\mathcal{T}'$. We keep only the propositions between representatives of the equivalence classes that still have a corresponding state in the automaton $M|_{\mathcal{S}}$, but we need to update the terms in the disequalities to the new representatives. Disequalities containing terms that are not in $\mathcal{T}'$ are removed.

For $k = \bot$, the restriction $k|_{\mathcal{S}}$ is $\bot$.

5 The C-2PO Analysis

This chapter presents the fundamental operations of the new domain. This includes a partial order and the operations *meet*, *join*, *widening* and *narrowing*. Two different versions of the *join* and of the *equal* operation are introduced, differing in precision and efficiency. Finally, the abstract effect of the assignment and the interprocedural analysis of the C-2PO analysis are discussed.

5.1 Equality

Given two kernels, k_1 and k_2, we want to decide whether they are semantically equivalent. This section presents two methods for computing the *equal* operation. Their main difference is in how the equivalence of two partitions P_1 and P_2 is decided. The first method compares the equivalence classes of the partitions P_1 and P_2. The second method is based on computing a normal form of the conjunctions represented by the kernels k_1 and k_2. It then suffices to compare the normal forms syntactically. This approach has the advantage that the normal form can be computed once for every kernel, and there is no need to recompute it for every comparison.

5.1.1 Comparing Equivalence Classes

Let $P_1 = (\mathcal{T}_1, \tau_1, \omega_1)$ and $P_2 = (\mathcal{T}_2, \tau_2, \omega_2)$ be two partitions. First, we extend both partitions to the set $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$. Then we compare the equivalence classes of the resulting partitions $\mathsf{ext}\, P_1\, \mathcal{T} = (\mathcal{T}, \tau_1', \omega_1')$ and $\mathsf{ext}\, P_2\, \mathcal{T} = (\mathcal{T}, \tau_2', \omega_2')$.

For this, we need to check for each element $t \in \mathcal{T}$ if all equalities implied by P_1 are also implied by P_2. Let $t' \equiv \tau_1' t$ and $z = \omega_1' t$. It follows that P_1 implies the equality $t = z + t'$. If $\omega_2' t + \tau_2' t = z + \omega_2' t' + \tau_2' t'$ holds, then the same equality is also implied by P_2.

Afterward, the disequalities and block disequalities are compared. For two kernels $k_1 = \langle P_1, M_1, B_1, D_1 \rangle$ and $k_2 = \langle P_2, M_2, B_2, D_2 \rangle$, we rewrite the disequalities B_2 and D_2 to be about the representatives of P_1. For each block disequality $\{t_1, t_2\} \in B_2$, we convert it to the disequality $norm_{P_1}(bl(t_1) = bl(t_2))$ and for each disequality in D_2 of the form $(t_1, z, t_2) \in D$, we convert it to the disequality $norm_{P_1}(t_1 = z + t_2)$. The normalization is possible because we previously extended the two partitions to the

same set $\mathcal{T}$, i.e., $\tau_1' t$ is defined for each term t occurring in B_2 and D_2. Then, we check if the resulting set of block disequalities is equal to the set B_1 and if the resulting set of disequalities equals the set D_1.

5.1.2 Compute Normal Form

Seidl et al. [Sei+24a] presents a different approach to deciding the equivalence of two partitions. This is done by computing a normal form $\mathsf{nf}(\Psi)$ of a conjunction Ψ, such that Ψ is semantically equivalent to $\mathsf{nf}(\Psi)$ and such that two conjunctions Ψ_1 and Ψ_2 are semantically equivalent iff $\mathsf{nf}(\Psi_1)$ and $\mathsf{nf}(\Psi_2)$ are syntactically equivalent.

The main idea is to find *minimal representatives* for each state s in the QFA M, which corresponds to the smallest term in $\mathcal{L}_M(s)$, according to the order defined in Chapter 4. Then, the automaton is transformed to a formula that utilizes only the minimal representatives, thus obtaining a normal form that is independent of the chosen set $\mathcal{T}$ and on the chosen representatives.

The minimal element of a set of terms is defined by using the following order on terms: Given a linear order $<$ on atoms, we define:

- $a < *(z + t)$ whenever a is an atom and
- $*(z_1 + t_1) < *(z_2 + t_2)$ whenever either $t_1 < t_2$ or $t_1 \equiv t_2$ and $z_1 < z_2$.

Let $M = (S, \overline{\tau}, \eta, \delta)$ be a QFA. For each state $s \in S$, we compute the minimal term m_s and the corresponding offset z_s such that $M[m_s] = (z_s, s)$. This can be computed using a variation of Dijktra's shortest path algorithm, which is described in the following.

Let Q be an initially empty FIFO queue, which we use to store all states s for which (m_s, z_s) is already computed but where the outgoing edges still have to be processed.

1. First, we consider all atoms a for which η is defined in ascending order. For each $\eta\, a = (z, s)$, if (m_s, z_s) is not defined yet, we set

   $$(m_s, z_s) = (a, z)$$

   and add s to Q.

2. Then, we process the queue Q in a FIFO manner until it is empty. For each state s in Q, we consider all outgoing edges $\delta\, z\, s = (z', s')$ in order of ascending z. For each such edge where $(m_{s'}, z_{s'})$ is not defined yet, we set

   $$(m_{s'}, z_{s'}) = (*((z - z_s) + m_s), z')$$

   and add s' to Q.

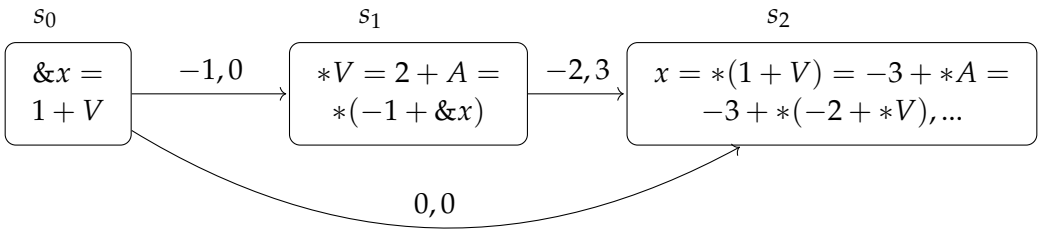


Figure 5.1: Visualization of the QFA corresponding to the conjunction $(V = -1 + \&x) \wedge (*V = 2 + A) \wedge (*V = 3 + x)$.

The correctness is given by the fact that longer paths always result in larger terms than shorter paths. Thus, we do not need to update a pair once it was computed.

Example 5.1. Let $\Psi \equiv (V = -1 + \&x) \wedge (*V = 2 + A) \wedge (*V = 3 + x)$. The corresponding automaton M is shown in Fig. 5.1.

The minimal representatives are computed by first considering the atoms in ascending order. Assuming that the linear order on atoms is $\&x < A < V$, we define first $m_{s_0} = \&x$ and $z_{s_0} = 0$. Then we define $m_{s_1} = A$ and $z_{s_1} = -2$. There is nothing to do for V because we already defined m_{s_0}. The queue Q is now $\{s_0, s_1\}$. We consider the outgoing edge of s_0 with $z = -1$. There is nothing to do for s_1, as m_{s_1} is already defined. Then we consider the outgoing edge of s_0 with $z = 0$ and set $m_{s_2} = x$ and $z_{s_2} = 0$. Thus, we have found the minimal representative for each state.

Using the minimal representatives, we can now transform the kernel $k = \langle P, M, B, D \rangle \neq \bot$ with $M = (S, \overline{\tau}, \eta, \delta)$ to a normal form $\mathcal{F}_{normal}[k]$, defined as follows:

$$\begin{aligned}\mathcal{F}_{normal}[k] \quad \equiv \quad & \textstyle\bigwedge_{\eta\, a=(z,s)} (a = (z - z_s) + m_s) \wedge \\ & \textstyle\bigwedge_{\delta\, z\, s=(z',s')} (*((z - z_s) + m_s) = ((z' - z_{s'}) + m_{s'}) \wedge \\ & \textstyle\bigwedge_{s,s' \in S \wedge \{\overline{\tau}\, s, \overline{\tau}\, s'\} \in B} (bl(m_s) \neq bl(m_{s'})) \wedge \\ & \textstyle\bigwedge_{s,s' \in S \wedge (\overline{\tau}\, s, z, \overline{\tau}\, s') \in D} (m_s \neq (z + z_s - z_{s'}) + m_{s'})\end{aligned}$$

The trivial propositions of the form $t = 0 + t$ are removed from the formula, as well as the repeated equalities and the implicit disequalities of the form $\&x \neq z + \&y$ and $bl(\&x) \neq bl(\&y)$.

The definition of $\mathcal{F}_{normal}[k]$ is analogous to the formula representation $\mathcal{F}[k]$, but it expresses the proposition using the minimal representatives instead of the union-find representatives. The advantage of this normal form is that the formula representation depends neither on the chosen representatives nor the chosen set $\mathcal{T}$. Thus, there is a unique normal form for each semantic equivalence class of kernels, even though

multiple kernel representations of the same conjunction exist. Therefore, in order to decide the equivalence of two kernels k_1 and k_2, it suffices to compare the normal forms $\mathcal{F}_{normal}[k_1]$ and $\mathcal{F}_{normal}[k_2]$ syntactically.

Example 5.2. Consider the conjunction Ψ from Example 5.1. The normal form of the kernel k representing Ψ is

$$(V = -1 + \&x) \wedge (*(-1 + \&x) = 2 + A) \wedge (*A = 3 + x)$$

where $V = -1 + \&x$ originates from $\eta\, V = (-1, s_0)$. The equalities originating from $\eta\, \&x = (0, s_0)$ and $\eta\, A = (0, s_1)$ are removed, because they are trivial. The edge $\delta\,(-1, s_0) = (0, s_0)$ gives rise to the equality $*(-1 + \&x) = 2 + A$ and $\delta\,(-2, s_1) = (3, s_2)$ generates the equality $*A = 3 + x$. The last edge $\delta\,(0, s_0) = (0, s_2)$ generates the trivial equality $x = 0 + x$.

The advantage of computing the normal form instead of comparing the equivalence classes is that it only needs to be computed once per kernel, and the resulting normal form can be reused for each equality check. In the implementation, it is possible to configure which of the two algorithms for equality is used by the analysis. If the normal form algorithm is chosen, then the kernel contains an additional field containing the normal form. This normal form is *lazily* computed, i.e., it will only be calculated when needed, and it is not computed if it is never used. Once calculated, the result is stored directly in the kernel and can be reused.

5.2 Partial Order

The natural partial ordering between conjunctions is the semantic implication, i.e., $\Psi_1 \to \Psi_2$ if and only if $(\rho, \nu, \mu) \models \Psi_1$ whenever $(\rho, \nu, \mu) \models \Psi_2$. This is the case if and only if $\Psi_1 \wedge \Psi_2$ is semantically equal to Ψ_1. Therefore, it is possible to reduce the *less equal* operation to a *meet* and an *equal* operation. A kernel k_1 that represents a conjunction Ψ_1 is *less or equal* to a kernel k_2 representing Ψ_2 if and only if $k_1 \sqcap k_2$ is semantically equal to k_1. We have already described how semantic equality between two kernels can be decided. In the next section, the *meet* operation ($\sqcap$) is presented.

5.3 Meet

Semantically, the *meet* of two conjunctions consists in taking the conjunction of the conjunctions: $\Psi_1 \sqcap \Psi_2 \equiv \Psi_1 \wedge \Psi_2$.

Given two kernels k_1 of Ψ_1 and k_2 of Ψ_2, we want to compute the kernel $k_1 \sqcap k_2$ that represents the conjunction $\Psi_1 \wedge \Psi_2$. If either $k_1 = \bot$ or $k_2 = \bot$, then $k_1 \sqcap k_2 = \bot$.

Otherwise, we consider the formula representation $\mathcal{F}[k_2]$ of k_2 and add each proposition of $\mathcal{F}[k_2]$ to the kernel k_1. First, all quantitative equalities are added, then all block disequalities, and finally, all disequalities. Let $k = \langle P, M, B, D\rangle$ and $P = (\mathcal{T}, \tau, \omega)$ and $M = (S, \overline{\tau}, \eta, \delta)$. Assuming that all the terms occurring in p have already been added to $\mathcal{T}$ by extending the partition P, the conjunction $p \sqcap k$ of a kernel k and a proposition p is defined as follows:

- Let $p \equiv (t_1 = z + t_2)$ and let $t_1' = \tau\, t_1$ and $t_2' = \tau\, t_2$.
  - If $t_1' \equiv t_2'$ and $\omega\, t_1 = z + \omega\, t_2$, then the equality is already implied by P and we set $p \sqcap k = k$.
  - If $t_1' \equiv t_2'$, but $\omega\, t_1 \neq z + \omega\, t_2$, or if $\{t_1', t_2'\} \in B$, or if $t_1' \neq (z + \omega\, t_2 - \omega\, t_1) + t_2'$ is in D, then there is a contradiction and we set $p \sqcap k = \bot$.
  - Otherwise, we compute the $\mathsf{closure}(P, t_1 = z + t_2)$. Let $P' = (\mathcal{T}, \tau', \omega')$ and M' be the partition and QFA resulting from the closure operation. Now, we need to update the block and quantitative disequalities to be about the representatives of P'. For each block disequality $\{t_1, t_2\}$ in B, we convert it to the disequality $bl(\tau'\, t_1) \neq bl(\tau'\, t_2)$. B' is the result of updating all terms of B to the new representatives:

    $$B' = \{\{\tau'\, t_1, \tau'\, t_2\} \mid \{t_1, t_2\} \in B\}.$$

    We set $D' = D[\mathcal{F}[k_1] \wedge p]$. If a contradiction occurs during the construction of D', we set $p \sqcap k = \bot$. We set $p \sqcap k = \langle P', M', B', D'\rangle$.
- Let $p \equiv (bl(t_1) \neq bl(t_2))$. Again, let $t_1' = \tau\, t_1$ and $t_2' = \tau\, t_2$.
  - If $t_1' \equiv t_2'$, then $p \sqcap k = \bot$. Otherwise, let $t_1' \not\equiv t_2'$. We define $p \sqcap k = \langle P, M, B', D'\rangle$, where B' and D' are described in the following:
  - If $t_1{}'$ and t_2' are address expressions, then $bl(t_1') \neq bl(t_2')$ is trivially true. Therefore, we set $B' = B$ and $D' = D$.
  - Otherwise, we add the disequality $\{t_1', t_2'\}$ to B. We set $D' = D[\mathcal{F}[k_1] \wedge p]$.
- Let $p \equiv (t_1 \neq z + t_2)$, $t_1' = \tau\, t_1$ and $t_2' = \tau\, t_2$.
  - If t_1' and t_2' are both address expressions, then the disequality is trivially true. The same holds if $t_1' \equiv t_2'$ and $\omega\, t_1 = z + \omega\, t_2$ or if $\{t_1', t_2'\} \in B$. In these cases, we set $p \sqcap k = k$.
  - If $t_1' \equiv t_2'$, but $\omega\, t_1 \neq z + \omega\, t_2$, then there is a contradiction and we set $p \sqcap k = \bot$.
  - We set $p \sqcap k = \langle P, M, B, D'\rangle$, where $D' = D[\mathcal{F}[k_1] \wedge p]$.

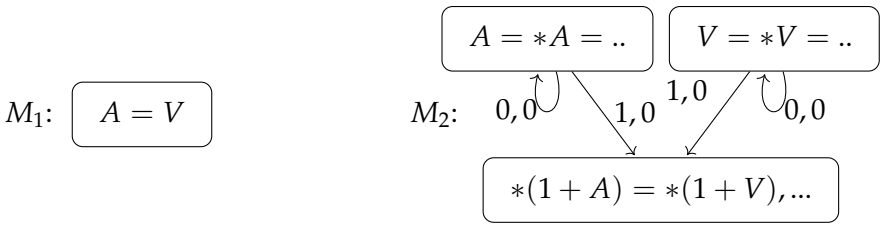


Figure 5.2: QFAs for $\Psi_1 \equiv (A = V)$ and $\Psi_2 \equiv (*A = A) \wedge (*V = V) \wedge (*(1 + A) = *(1 + V))$

As an optimization, the quantitative disequalities can be computed only at the end, and not after each added proposition, by setting $D' = D[\mathcal{F}[k_1] \wedge \mathcal{F}[k_2]]$

5.4 Join

Given two conjunctions Ψ_1 and Ψ_2, the least upper bound is a conjunction implied by both Ψ_1 and Ψ_2 that is smallest with respect to the partial order. Intuitively, the least upper bound is $\Psi_1 \vee \Psi_2$. However, disjunctions are not expressible in the 2-Pointer Logic, so we are looking for the most precise conjunction that is implied by $\Psi_1 \vee \Psi_2$. The conjunction of all propositions implied by a disjunction is not always representable as a conjunction of finite size [GTN04; Sei+24a]. Therefore, it is impossible to compute the exact least upper bound of two conjunctions.

Example 5.3. Adapting the original example by Gulwani et al. [GTN04] to C-2PO, consider the conjunctions $\Psi_1 \equiv (A = V)$ and $\Psi_2 \equiv (*A = A) \wedge (*V = V) \wedge (*(1 + A) = *(1 + V))$, represented by the automata M_1 and M_2 shown in Fig. 5.2. Then for each $n \geq 0$, the equality $*(1 + *^n A) = *(1 + *^n V)$ is implied both by Ψ_1 and Ψ_2, where $*^m$ denotes m-fold dereference. It can be shown that the set of quantitative equalities implied both by Ψ_1 and Ψ_2 cannot be represented by a finite conjunction [GTN04; Sei+24a].

This example shows that it is not possible to compute the exact least upper bound of two domain elements because of the limitations in the expressiveness of the congruence closure representation. However, it is possible to define a sound over-approximation of the join operation.

Given that equalities only depend on the automaton and cannot follow from the disequalities, we first consider how to construct an automaton that represents the conjunction of two automata. Later, we will consider the (block-)disequalities that follow from Ψ_1 and Ψ_2.

As we only compute an approximated join operation, multiple possibilities exist to define the join. A compromise must be made between the precision of the join and the computational complexity. Two different approaches will be presented in the following sections. The first one computes the join based on the automata representation of the equalities, while the second only considers the partition P of the terms $\mathcal{T}$. This first approach is less efficient but more precise, as all terms of the set $\mathcal{L}(M)$ are considered, while in the second approach, only the terms of the set $\mathcal{T}$ are considered. In particular, circular dependencies between terms are lost in the second approach. However, these circular dependencies do not often occur in practice. Therefore, we expect that the difference in precision is not significant.

5.4.1 Join Using the Automaton

We propose an implementation of the join algorithm of two quantitative finite automata. It is based on the work of Gulwani et al. [GTN04], where they discuss join algorithms for congruence closure data structures.

Given two partitions $P_1 = (\mathcal{T}_1, \tau_1, \omega_1)$ and $P_2 = (\mathcal{T}_2, \tau_2, \omega_2)$ and the corresponding automata M_1 and M_2, we want to compute the partition P_3 and the automaton M_3 that represent the conjunction of the two automata. The first step is to extend the set of terms of P_1 and P_2 to $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$. Henceforth, we assume that both partitions are defined over the same set of terms $\mathcal{T}$, i.e., $P_1 = (\mathcal{T}, \tau_1, \omega_1)$ and $P_2 = (\mathcal{T}, \tau_2, \omega_2)$. Let M_1 and M_2 be the corresponding automata.

The join is defined as the product automaton of M_1 and M_2.

Definition 5.4. Let $M_1 = (S_1, \overline{\tau}_1, \eta_1, \delta_1)$ and $M_2 = (S_2, \overline{\tau}_2, \eta_2, \delta_2)$ be two quantitative finite automata. The *quantitative product automaton* $M_3 = (S_3, \overline{\tau}_3, \eta_3, \delta_3)$ is defined as:

- If $\eta_1(a) = (z_1, s_1)$ and $\eta_2(a) = (z_2, s_2)$, then $\eta_3(a) = (z_1, \langle s_1, s_2, z_2 - z_1 \rangle)$.
- If $\delta_1(z_1, s_1) = (z_1', s_1')$ and $\delta_2(z_2, s_2) = (z_2', s_2')$, then $\delta_3(z_1, \langle s_1, s_2, z_2 - z_1 \rangle) = (z_1', \langle s_1', s_2', z_2' - z_1' \rangle)$.
- $S_3 \subseteq S_1 \times S_2 \times \mathbb{Z}$. S_3 contains all states that are reachable in M_3.
- For each $s \in S_3$, $\overline{\tau}_3(s)$ is any term in $\mathcal{L}_{M_3}(s)$.

The product automaton is finite, as there are only a finite amount of reachable states in $S_1 \times S_2 \times \mathbb{Z}$. The remaining states are unreachable, i.e., they neither occur in a return value of η_3 nor δ_3. This follows from the fact that η_i and δ_i for $i = 1, 2$ are defined only for a finite amount of values, as discussed in Section 4.2.2.

The product can be computed in polynomial time, as we need to add an initial state for all pairs of initial states of M_1 and M_2 and then add a transition for each pair of

transitions of M_1 and M_2. Therefore, the runtime is equal to the size of M_1 times that of M_2.

Lemma 5.5. *Let M_3 be the product automaton of two quantitative automata M_1 and M_2. Then, for each term $t \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ it holds that $M_1[t] = (z_1, s_1)$ and $M_2[t] = (z_2, s_2)$ iff $M_3[t] = (z_1, \langle s_1, s_2, z_2 - z_1 \rangle)$.*

Proof. We prove it by induction over the structure of the term t. □

As shown in Example 5.3, it is not possible to define a join algorithm that is complete. In fact, we can only show for our definition of the join algorithm that it is at least complete over the terms in $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. The following proposition shows that our definition of join is complete over the set of terms $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ and that it is sound.

Proposition 5.6. *Let M_3 be the product automaton of two quantitative automata M_1 and M_2. Then, for all terms $t_1, t_2 \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ it holds that $M_3[t_1] = z + M_3[t_2]$ iff $M_1[t_1] = z + M_1[t_2]$ and $M_2[t_1] = z + M_2[t_2]$.*

Proof. This proposition follows from Lemma 5.5. □

As we mentioned before, we added all terms of $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ to both automata before computing the product automaton, and after having added these terms, it holds that $\mathcal{T} \subseteq \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. Therefore, the join of two automata is complete at least over the terms in $\mathcal{T}$.

Using the automaton M_3, we can compute the corresponding partition P_3. Let $\mathcal{T}' = \{t \mid \exists s \in S_3. \overline{\tau}_3(s) = t\}$ be the set of terms chosen as representatives for the states of M_3. The partition P_3 is defined as $P_3 = (\mathcal{T}' \cup \mathcal{T}, \tau_3, \omega_3)$, where for each $t \in \mathcal{T}' \cup \mathcal{T}$, $\tau_3\, t = \overline{\tau}_3\,(s)$ and $\omega_3\, t = z$ if $M_3[t] = (z, s)$.

Example 5.7. We consider again the two automata of Example 5.3. First, we add the terms of the set $\mathcal{T} = \{A, V, *A, *V, *(1 + A), *(1 + V)\}$ to the automaton M_1, resulting in the automata M_1'. Let M_3 be the product automata of M_1' and M_2. The automata M_1' and M_3 are shown in Fig. 5.3 From M_3 we can infer the equality $*(1 + A) = *(1 + V)$, while the equalities $*(1 + *^n A) = *(1 + *^n V)$ for $n \geq 1$ are implied by M_1' and M_2, but not by M_3.

5.4.2 Join Using the Partition

We propose a second approach to compute the join of two quantitative automata. This approach is less precise than the previous one, as it only considers the partition of the terms $\mathcal{T}$ and not the automaton.

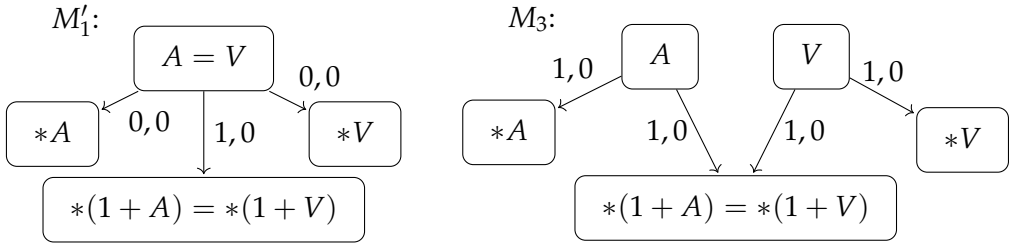
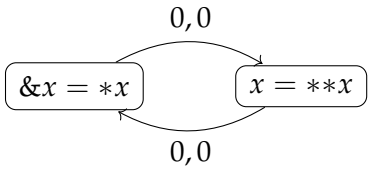
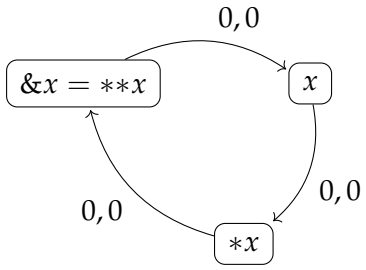


Figure 5.3: Visualization of the automata M_1' and M_3.



(a) The QFA of the conjunction $\&x = *x$.



(b) The QFA of the conjunction $\&x = **x$.

Figure 5.4: QFAs for the conjunctions $\&x = *x$ and $\&x = **x$.

As before, we first extend the two partitions to the same set of terms $\mathcal{T}$ that contains all terms of both partitions.

Let $P_1 = (\mathcal{T}, \tau_1, \omega_1)$ and $P_2 = (\mathcal{T}, \tau_2, \omega_2)$ be two partitions. We define the partition $P_3 = (\mathcal{T}, \tau_3, \omega_3)$ that represents the join of P_1 and P_2. The terms of $\mathcal{T}$ are partitioned in sets Q_i such that for each two terms t_1 and t_2 in the same set, it holds that $\tau_1\, t_1 = \tau_1\, t_2$, $\tau_2\, t_1 = \tau_2\, t_2$ and $\omega_2\, t_1 - \omega_1\, t_1 = \omega_2\, t_2 - \omega_1\, t_2$. Then for each equivalence class Q_i we choose a representative term $t_i \in Q_i$ and for each $t \in Q_i$ we define $\tau_3\, t = t_i$ and $\omega_3\, t = \omega_1\, t - \omega_1\, t_i$.

The corresponding automaton M_3 can be computed starting from the partition P_3, as described in Section 4.2.2.

Example 5.8. The main difference between the two join algorithms is the behavior when the automata contain cycles. For example, let $\Psi_1 \equiv \&x = *x$ and $\Psi_2 \equiv \&x = **x$. Assume that $\mathcal{T} = \{\&x, x, *x, **x\}$. The automaton $M[\Psi_1]$ is a cycle of length 2 (Fig. 5.4a) and the automaton $M[\Psi_2]$ is a cycle of length 3 (Fig. 5.4b). If we use the join of the automata, the resulting automaton is a cycle of length 6. However, the join of the partitions only considers the terms in $\mathcal{T}$, which are all in distinct equivalence classes in the join. Therefore, all information about any equalities are lost with the partitions-join.

5.4.3 Join of Two Kernels

Given two kernels $k_1 = (P_1, M_1, B_1, D_1)$ and $k_2 = (P_2, M_2, B_2, D_2)$, we want to compute the kernel $k_3 = (P_3, M_3, B_3, D_3)$ that represents the join of the two kernels. We already discussed two different methods of computing the partition P_3 and the automaton M_3 that represents the join of the two automata. Now we consider the (block-)disequalities that follow from both kernels, assuming that P_3 and M_3 are the join of P_1 and P_2 and M_1 and M_2, respectively, using either of the two methods described above.

Block Disequalities

In order to infer the block disequalities that follow from the conjunction of two kernels k_1 and k_2, we rewrite the block disequalities of k_1 and k_2 to use the new representatives of the new equivalence classes of $P_3 = (\mathcal{T}, \tau_3, \omega_3)$. Then, we take the intersection of these two sets of block disequalities.

Thus, B_3 is defined as

$$\begin{aligned} B_3 \quad &= \{\{\tau_3\, t_1, \tau_3\, t_2\} \mid \\ &\quad (\{\tau_1\, t_1, \tau_1\, t_2\} \in B_1 \wedge \{\tau_2\, t_1, \tau_2\, t_2\} \in B_2)\}. \end{aligned}$$

B_3 is computed by rewriting each equality $\{t, t'\} \in B_1$ to use the new representatives of the equivalence classes in P_3. We consider the disequality $\{t_1, t_2\}$ for each terms t_1 and t_2 that are representatives in P_3 and for which t_1 is in the same equivalence class of t and t_2 is in the same equivalence class of t' in both partition P_1. There could be multiple such sets $\{t_1, t_2\}$, as the equivalence classes of P_1 might be divided into multiple equivalence classes in P_3. We add each of these disequalities to B_3 if $\{\tau_2\, t_1, \tau_2\, t_2\} \in B_2$.

Disequalities

In Section 4.4, we defined the two types of disequalities that follow from a conjunction: the set of disequalities that follow from the explicitly stored disequalities, and the set of implicit disequalities that follow from the equalities or the block disequalities. As before for the equalities, it is also impossible to define a join operation that computes all disequalities that follow from two conjunctions, as is shown in the following example.

Example 5.9. Let $\Psi_1 \equiv A = V + 1$ and $\Psi_2 \equiv A = V + 2$. There are infinitely many equalities implied by Ψ_1 and Ψ_2, i.e., $A \neq V + z$ for all $z \in \mathbb{Z} \setminus \{1, 2\}$. There is no way to implicitly represent these equalities via equalities or block disequalities, and we can only store a finite amount of explicit disequalities. Therefore, it is not always possible to compute the exact set of disequalities implied by two conjunctions.

We have seen that the intersection between the set of implicit disequalities that follow from two conjunctions Ψ_1 and Ψ_2 could be infinite. Therefore, we define the join to keep only those disequalities that follow from an explicit disequality in Ψ_1 and from an explicit or implicit disequality in Ψ_2 or vice versa.

Let $P_3 = (\mathcal{T}, \tau_3, \omega_3)$. We define

$$\begin{aligned} D_3 \quad &= \{(\tau_3\, t_1, z + \omega_3\, t_2 - \omega_3\, t_1, \tau_3\, t_2) \mid \\ &\quad ((\tau_1\, t_1, \tau_1\, t_2, z + \omega_1\, t_2 - \omega_1\, t_1) \in D_1 \vee (\tau_2\, t_1, \tau_2\, t_2, z + \omega_2\, t_2 - \omega_2\, t_1) \in D_2) \\ &\quad \wedge t_1 \not\equiv_{k_1} t_2 + z \wedge t_1 \not\equiv_{k_2} t_2 + z\}, \end{aligned}$$

where $\not\equiv_k$ symbolizes the disequalities that are (implicitly or explicitly) implied by k, i.e., it is equal to $\not\equiv_{\mathcal{F}[k],\mathcal{T}}$. They can be determined by following the rules described in Section 4.4.

D_3 can be computed in polynomial time in the number of explicitly stored disequalities by first constructing the set containing all equalities of D_1 by computing their closure with respect to the *closure under quantitative equalities* rule [B2]. This is necessary because each equivalence class of P_1 and P_2 might now be divided into multiple equivalence classes in P_3. Then, for each resulting disequality $t_1 \neq z + t_2$, we need to rewrite it to use the new representatives of the equivalence classes in P_3, so we consider the disequality $\tau_3\, t_1 \neq z' + \tau_3\, t_2$, where $z' = z + \omega_3\, t_2 - \omega_3\, t_1$. If this equality is implied (implicitly or explicitly) by k_2, then it is added to D_3.

The same is done for the disequalities D_2.

The only disequalities that are lost by this definition of join are those that are implied only by the implicit disequalities of Ψ_1 and the implicit disequalities of Ψ_2.

Thus, we can define the join of two kernels $k_1 = (P_1, M_1, B_1, D_1)$ and $k_2 = (P_2, M_2, B_2, D_2)$ as $k_1 \sqcup k_2 \equiv (P_3, M_3, B_3, D_3)$.

5.5 Widening

The join operation that we defined does not compute the exact least upper bound of two automata, but only an approximation. However, the loss of information is not enough to guarantee that the join operation always reaches a fixpoint after a finite number of iterations.

Example 5.10. Consider the sequence of kernels corresponding to the conjunctions

$$\Psi_n \equiv (*^{2^n} \&x = \&x) \quad (n \geq 0).$$

This could for example occur in a list structure, where each element points to its successor, and the last element points again to the first element of the list. Then we

have for each $n \geq 0$, $\Psi_n \Longrightarrow \Psi_{n+1}$ and in particular the product automaton between $M[\Psi]$ and $M[\Psi_{n+1}]$ is $M[\Psi_{n+1}]$. If we use the second algorithm for join and compute the join of the partitions $P[\Psi_n]$ and $P[\Psi_{n+1}]$, we get the partition $P[\Psi_{n+1}]$.

Therefore, we need to define a widening operator [CC92] that finds an even less precise upper bound of two automata than the join, such that it reaches a fixpoint after a finite number of iterations.

We decide to define the widening operator in a way that the result cannot contain terms of arbitrary size, such that there cannot be ascending chains of arbitrary size. Therefore, we define the widening operator as first computing the join and then restricting the set of terms of the result to the set of terms that occur in the first input conjunction.

Definition 5.11. For two kernels k_1 and k_2 over the sets of terms $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, we define the widening operation as $k_1 \nabla k_2 = (k_1 \sqcup k_2)|_{\mathcal{T}_1}$.

The widening operator depends on which join operator was chosen, and it works for both possibilities. However, if we choose to use the join operator that uses the partition, the one described in Section 5.4.2, then it can be implemented in a more efficient way. Instead of adding all the terms of the two kernels to both kernels at the beginning of the join, we simply add the terms of the first kernel to the second kernel, and we build the resulting partition starting from the set of terms of the first kernel. This computes the same result as the above defined widening operator, but it avoids computing a restriction.

For a given set $\mathcal{T}$ of terms, there is only a finite number of possible automata and sets of disequalities that contain only terms of $\mathcal{T}$. The set of terms does not change after having applied the widening operation, therefore the widening operator will always reach a fixpoint after a finite amount of iterations.

Example 5.12. If we consider the sequence of conjunctions Ψ_n from Example 5.10, the automaton representation $M[\Psi_n]$ is a cycle of length 2^n. For example, Fig. 5.4a shows the automaton for Ψ_1 If we apply widening to Ψ_n and Ψ_{n+1}, the resulting automaton is restricted to only contain the terms in Ψ_n. The term $*^{2^{(n+1)}}\&x$ is therefore removed, together with the corresponding state in the automaton, thus removing the cycle. Afterward, each term is in its own equivalence class, and we have lost all information about equalities. Therefore, the widening terminates in the next iteration, as we already reached the top element of our domain.

5.6 Narrowing

As shown in [Sei+24a], the domain of 2-Pointer propositions also has infinite strictly descending chains.

Example 5.13. Consider the sequence of conjunctions

$$\Phi_n \equiv \bigwedge_{i=1}^{n} (\&y = *(i + \&x)) \qquad (n \geq 0)$$

Then $\Phi_{n+1} \implies \Phi_n$ for all $n \geq 0$ and all are inequivalent. Thus, the conjunctions Φ_n constitute an infinite strictly descending chain in C-2PO.

Similarly to the widening operation, we define the narrowing as the meet operation with a limited set of terms. However, it is not sufficient to limit the terms of the result, we also need to limit the possible offsets appearing in disequalities.[Ghi+24]

Example 5.14. Consider the sequence of kernels corresponding to the propositions

$$\Psi_n \equiv (y \neq n + \&x) \qquad (n \geq 0)$$

Let k_n denote the $k[\Psi_n]$. Then all kernels consider the set $\mathcal{T} = \{\&x, \&y, y\}$ of terms. The sets D_n of disequalities, are given by $D_n = \{(y, n, \&x)\}$. Consider the sequence $a_n, n \geq 0$ defined by $a_0 = k_0$ and $a_n = a_{n-1} \sqcap k_n$ for $n > 0$. Each element a_m, is a kernel with set D'_m of quantitative disequalities given by

$$D'_m = \{(y, j, \&x) \mid 0 \leq j \leq m\}.$$

To guarantee that the kernel does not grow in each iteration and that a fixpoint is always reached, we only consider the propositions that contain terms of the first input conjunction and disequalities that contain offsets that are in the range of offsets of the first input conjunction.

Let $k = \langle P, M, B, D \rangle$ be a kernel. We define $max\_offset(D)$ to be the biggest absolute value of an offset that appears in the disequalities of D:

$$max\_offset(D) = \max\{|z| \mid (t_1, z, t_2) \in D\}.$$

Let k_1 and k_2 be the kernels. As for the meet, if either $k_1 = \bot$ or $k_2 = \bot$, then $k_1 \sqcap k_2 = \bot$. Otherwise, we consider the formula representations $\mathcal{F}[k_1]$, and we keep only the propositions that contain terms of $\mathcal{T}_1$. We also filter out the disequalities in k_2 that contain offsets z where $|z| > max\_offset(D_1)$. Then we add each remaining proposition of $\mathcal{F}[k_1]$ to the kernel k_2, as is done for the meet operation in Section 5.3.

5.7 Assignment

In the following, we will describe the abstract semantics $[\![t_1 := t_2]\!]^\sharp k$ of the assignment $t_1 := t_2$ over the kernel k, for each of the three possibilities of t_2. If $k \equiv \bot$, then $[\![t_1 := t_2]\!]^\sharp k \equiv \bot$. In the following, assume that $k \neq \bot$.

5.7.1 Indefinite Assignment

We consider an assignment of the form $t_1 := ?$. After this assignment, the value of t_1 changes to an unknown value, so we must forget all the equalities and disequalities that contain the term t_1, but also all other terms that may be modified by overwriting t_1. Only the terms where we know that each subterm is definitely not equal to the address of t_1 and also definitely not overlapping with t_1 are not modified by overwriting t_1.

Example 5.15. Consider the following C program:

```
1  int main(){
2          long *lpt = (long *)malloc(sizeof(long));
3          long lo = 0;
4
5          *lpt = lo;// *lpt: 0; l: 0
6          assert(*lpt == lo);
7          *(lpt + 1) = 'a';// *lpt: 0; l: 0
8          assert(*lpt == lo);
9          *((char *)lpt + 1) = 'a';// *lpt: 24832; l: 0
10         assert(*lpt == lo); // this is false
11 }
```

In line 6, the assertion `*lpt == lo` holds, because the value of `*lpt` and the value of `lo` are both 0. In line 7, the expression `*(lpt + 1)` is overwritten, which is represented as $*(64 + lpt)$ in our analysis. We now want to determine if the term $*lpt$ is modified by this assignment. The address of $*lpt$ is lpt and the address of $*(64 + lpt)$ is $64 + lpt$. Given that $lpt \not\equiv_k 64 + lpt$, we know that the assignment does not write at the same address as $*lpt$, therefore we do not need to forget the information we had about $*lpt$.

However, in line 9, the expression `*((char *)lpt + 1)` is overwritten, which is represented as $*(8 + lpt)$ in our analysis. Even though we know that $lpt \not\equiv_k 8 + lpt$, it does not hold that the value of $*lpt$ does not change when overwriting $*(8 + lpt)$, because the two values overlap. Therefore, when a disequality between two terms follows from an equality with an offset, we additionally need to make sure that the terms do not overlap in order to be sure that assigning to one of them does not modify the other one.

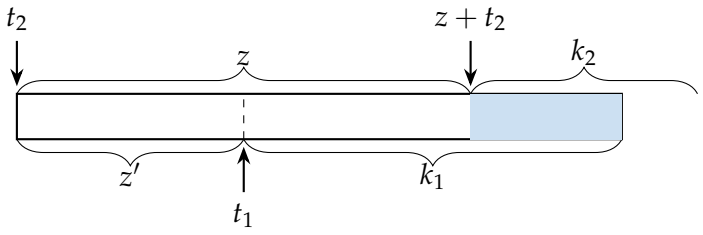


Figure 5.5: Visualization of two pointers $z + t_2$ and t_1 that have an overlapping region. It holds that $t_1 \equiv_k z' + t_2$. The pointer t_1 points to a value of size k_1 and $z + t_2$ points to a region inside this value. The highlighted region is the overlap.

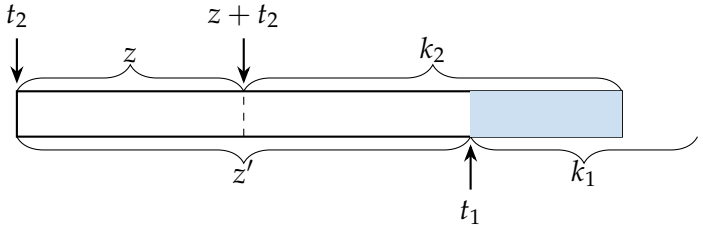


Figure 5.6: Visualization of two pointers t_1 and $z + t_2$ that have an overlapping region. It holds that $t_1 \equiv_k z' + t_2$. The pointer $z + t_2$ points to a value of size k_2 and t_1 points to a region inside this value. The highlighted region is the overlap.

Let t_1 be a pointer to a value that is k_1 bits large and t_2 be a pointer to a value of k_2 bits. The two terms t_1 and $z + t_2$ overlap iff

- $t_1 \equiv_k z' + t_2$ and
- $(z - z') \geq 0$ and $(z - z') < k_1$ (as in Fig. 5.5), or $(z - z') < 0$ and $(z' - z) < k_2$ (see Fig. 5.6),

where $\equiv_k$ symbolizes the equalities that are implied by k, i.e., it is equal to $\equiv_{\mathcal{F}[k],\mathcal{T}}$.

In the following, we will describe how to find the set of terms that are not modified by an assignment. For this, we use different information sources: the disequalities implied by the abstract domain, the C-type of the terms to find potential overlaps, and the *MayPointTo* analysis that is present in GOBLINT and which was introduced in Section 2.2.

Given a kernel $k = \langle P, M, B, D \rangle$, a partition $P = (\mathcal{T}, \tau, \omega)$ and a term t, we want to compute the set $\mathcal{T}|_{\neg t}$ that contains all terms $t' \in \mathcal{T}$ such that each subterm of t' definitely does not alias or overlap with t. This is exactly the set of terms that are not

modified when the value of t changes. We differentiate between two cases: the case where t is an atom and the case where t is a dereferenced term.

If t is an atom, then it is impossible to reach the address of t by dereferencing. Therefore, $\mathcal{T}|_{\neg t}$ is the set of all terms that do not contain t as a subterm.

If $t \equiv *(z + t')$ is a dereferenced term, then $\mathcal{T}|_{\neg t}$ contains all the terms where for each subterm of the form $*(z' + v)$ it holds that:

- The disequality $v \not\equiv_k (z - z') + t'$ is implied by D or B;
- or $v \equiv_k z'' + t$ for any $z'' \neq (z - z')$ and the terms v and $(z' + z'' - z) + t'$ do not overlap. The C-type of the terms is used to determine the size in bits of the value that is stored at the address of the term;
- or the intersection of the sets of addresses that v may point to and the addresses that $(z - z') + t'$ may point to is empty. These sets are computed using the *MayPointTo* analysis of Goblint.

To simplify the notation, the restricted kernel $k|_{\mathcal{T}|_{\neg t}}$ is denoted $k|_{\neg t}$. It represents all propositions that follow from k and that are still valid after overwriting the value of t. The assignment is defined as follows:

$$[\![t_1 := ?]\!]^\sharp\, k \equiv k|_{\neg t_1}\,.$$

Proposition 5.16. *For every kernel k and assignment $t_1 := ?$,*

$$[\![t_1 := ?]\!]\,(\gamma\, k) \subseteq \gamma\,([\![t_1 := ?]\!]^\sharp\, k)$$

The proof of this proposition for the case of a one-dimensional memory model can be found in [Sei+24a]. This proof can easily be adapted to the two-dimensional model.

5.7.2 Definite Assignment

We consider an assignment of the form $t_1 := z_1 + t$. As before, we need to forget all the equalities and disequalities about terms that may be modified by overwriting t_1. After this, is not sufficient to simply add the equality $t_1 = z_1 + t$ to the set of equalities, because the term $z_1 + t$ may also be modified by the assignment. For example, when we assign $V := 1 + V$, the equality $V = 1 + V$ would incorrectly lead to an unsatisfiable conjunction. Therefore, we introduce a fresh auxiliary A and first assign A to the right-hand side of the assignment, then restrict the automaton, and then assign t_1 to A. This way we can remember the original value of $z_1 + t$ trough the auxiliary A and reconstruct the equalities that still hold after the assignment.

Thus, the abstract effect of the definite assignment is defined as follows:

$$[\![t_1 := z_1 + t]\!]^\sharp\, k \;\equiv\; ((t_1 = A) \wedge ((A = z_1 + t) \wedge \Psi)|_{\neg t_1})\Big|_{\neg A}$$

Proposition 5.17. *For every kernel k and any assignment of the form $t_1 := z_1 + t$,*

$$[\![t_1 := z_1 + t]\!]\,(\gamma\, k) \subseteq \gamma\,([\![t_1 := z_1 + t]\!]^\sharp\, k)$$

As before, the proof of this proposition for the case of a one-dimensional memory model can be found in [Sei+24a].

5.7.3 Memory Allocation

We consider an assignment of the form $t_1 := \mathsf{malloc}$. The term t_1 is assigned to a fresh address, therefore we know that after the assignment, the block of the term t_1 is different from the block of any other term in the set $\mathcal{T}$ of the partition, but also different than any other term $t \in \overline{\mathcal{T}}$ that is not (yet) in the partition. However, we cannot remember an infinite number of equalities that contain an infinite number of possible terms, therefore we choose to only remember that this new address is different than the address of any other term in $\mathcal{T}$. Let $k = \langle P, M, B, D\rangle$, and $k|_{\neg t_1} = \langle P', M', B', D'\rangle$. For each equivalence class in the partition P', we add a disequality that states that the block of the term t_1 is different from the block of the representative of the equivalence class.

$$[\![t_1 := \mathsf{malloc}]\!]^\sharp\, k \equiv k|_{\neg t_1} \sqcap \bigwedge_{t \in \mathcal{T}, t \not\equiv t_1} (bl(t_1) \neq bl(\tau\, t))$$

Proposition 5.18. *For every kernel k and any assignment of the form $t_1 :=$ malloc,*

$$[\![t_1 := \mathit{malloc}]\!]\,(\gamma\, k) \subseteq \gamma\,([\![t_1 := \mathit{malloc}]\!]^\sharp\, k)$$

Proof. We only consider the case where t_1 equals the expression $*(z + t)$. The other case where $t_1 = V$ is analogous. Let $(\rho', \nu', \mu') \in [\![t_1 := \mathsf{malloc}]\!]\,(\gamma\, k)$, then we know that there exists $(\rho, \nu, \mu) \in \gamma\, k$ such that $\rho' = \rho$, $\nu' = \nu$ and $\mu' = \mu \oplus \{(z + [\![t]\!]\,(\rho, \nu, \mu)) \mapsto (a, 0))\}$ and a is *fresh*. From Proposition 5.16 it follows that $(\rho', \nu', \mu') \models k|_{\neg t_1}$. Given that a is *fresh*, it follows that all addresses stored in any term which is not t_1 do not have the same address block as a, therefore $(\rho', \nu', \mu') \models \bigwedge_{t \in \mathcal{T}, t \not\equiv t_1} (bl(t_1) \neq bl(\tau\, t))$. It directly follows that $(\rho', \nu', \mu') \in \gamma\,([\![t_1 := \mathsf{malloc}]\!]^\sharp\, k)$ □

5.8 Interprocedural Analysis

When entering a function, we need to forget the local variables of the caller, but keep the information about global variables and the content of the memory. After performing

the analysis of a function and then return to the caller, it is necessary to restore the propositions between the return value/global variables and the parameters given at the beginning of the function. While the parameters may have been overwritten during the function execution, the values of local variables that were handed over as parameters to the function are still unchanged when we return to the caller. Therefore, it is necessary to keep track of the initial value of the parameters in a function. This will be illustrated in the following example.

Example 5.19. Consider the C program:

```
1    int* f(int* a, int* b) {
2        int *g = a;
3        a = NULL;
4        return g;
5    }
6    int main(){
7        int* c;
8        int* d;
9        int* e = f(c,d);
10       assert(e == c);
11   }
```

If we simply remove all local variables of the caller from the abstract state, then after the abstract enter function ($\mathsf{enter}^\sharp$), the state represents an empty conjunction at the beginning of the function in line 1. After line 2, the abstract state is $g = a$. We lose all information after line 3. When returning to main, we cannot retrieve any information from the function, and we lose the information that $e = c$.

To solve this problem, we add additional shadow variables for each parameter of the function, that represent the initial values of the parameter at the beginning of the function. This way, in line 1, we add the variables a' and b' and the equalities $a = a'$ and $b = b'$ to the abstract state. After line 2, we add the equality $a = g$. After line 3, we lose information about a, but we still know that $g = a'$. When returning to main, we know that the parameters with which we called f were c and d, so we can add the equalities $c = a'$ and $d = b'$. Then we still need to remove the local variables of f from the abstract state, as well as the shadow variables a' and b'. But the equality $e = c$ is still preserved.

More in detail, the abstract enter function $\mathsf{enter}^\sharp$ performs the following steps:

1. Remove all local variables of the caller from the abstract state.

2. For each parameter p of the function, add a shadow variable p' to the abstract state, as well as the equality $p = p'$.

The abstract combine function $\mathsf{combine}^\sharp$ performs the following steps:

1. Remove all tainted variables from the caller state. The set of tainted variables is determined by query MayBeTainted, described in Section 2.2.
2. Meet the caller state with the state of the function.
3. For each parameter p, if the function was called with the expression e as a value for the parameter p, and if this expression is expressible in 2-Pointer Logic, add the equality $p' = e$ to the resulting state.
4. If the return value is assigned to the variable x in the caller, and the function returns an expression e' and this expression is expressible in 2-Pointer Logic, then the abstract effect of the assignment $[\![x = e']\!]^\sharp$ is applied to the resulting state.
5. Remove all local variables of the function from the abstract state, as well as the shadow variables corresponding to the current function call.

6 Evaluation

The C-2PO analysis was implemented in the Goblint static analyzer for C programs, which is based on abstract interpretation. Goblint already contains a set of basic abstract domains and analyses, including interval analysis, inter-procedural analyses, and a non-relational pointer analysis. The group of these basic analyzes is denoted as *base*. An additional Goblint analysis, called *var_eq*, tracks must-equalities between general expressions and thus can handle a subset of the behavior of C-2PO. We evaluated the precision gain and the performance loss of the C-2PO analysis and compared it to *base* and *var_eq*. Moreover, we analyzed the difference in precision and performance between four different configurations, which differ in the choice of the join algorithm and the choice of the equal function. The four configurations are denoted as:

1. *c*-2po_1 which uses the precise join with the automaton and computes a canonical normal form for each domain element in order to compare them.
2. *c*-2po_2 which uses the join that considers the partition and computes *equal* again by using the normal form.
3. *c*-2po_3 which uses the precise join with the automaton and compares the equivalence classes of the partitions.
4. *c*-2po_4 which uses the join that considers the partition and computes *equal* by comparing the equivalence classes.

All benchmarks were conducted on a machine with two Intel Xeon Platinum 8260 CPUs and 512 GB of RAM.

6.1 Precision

The precision of the C-2PO analysis does not depend on the choice of the algorithm for *equal*, as the two possible implementations are functionally equivalent. Therefore, in the following, we will only compare the precision of the *c*-2po_1, *c*-2po_2, *base*, and *var_eq* analyses. They were evaluated on two test suites.

The first suite consists of 29 *litmus* tests inspired by real-world programming patterns, designed to incorporate complex manipulations of pointers. Each test was annotated

Table 6.1: Summary of precision experiments. For each group of programs, the number of programs, the lines of code, and the total number of invariants generated by c-2po_1 are given. ✓ indicates that all assertions are proven. Otherwise, the number of proven assertions is given.

Group	#	Σ LLoC	Σ Inv.	*base*	*var_eq*	c-2po_1	c-2po_2
litmus	29	419	109	17	23	✓	108
coreutils	9	97664	19488	4636	8188	✓	19386

with assertions to be verified. c-2po_1 could prove all assertions, while only 16% of the assertions could be proven by *base* and 23% by *var_eq*. c-2po_2 proved all assertions except one, which was the Example 5.8. These programs may serve as benchmarks for future analyzers.

We further evaluated the precision of C-2PO on 11 GNU core utilities programs (Coreutils). We used c-2po_1 to identify program invariants and automatically generate assertions. For one of these programs, the generation of invariants timed out after one hour. For the remaining nine programs, the *base* analysis could prove only 24% of the generated invariants and *var_eq* 42%. In contrast, c-2po_1 was able to prove all invariants. c-2po_2 could prove more than 99% of the invariants.

Table 6.1 summarizes these results.

We can conclude that the C-2PO analysis can prove many assertions about relations between pointers that are out of reach for the *base* analysis, and it is also more precise than the *var_eq* analysis. As expected, the c-2po_1 analysis is more precise than the c-2po_2 analysis, as its join operation maintains more equalities in case the quantitative automaton contains cycles. However, the difference in precision is minimal, indicating that in practice, the cases where there is a cycle in the automaton do not occur frequently. Therefore, the choice of the join operation does not significantly affect the precision of the analysis.

6.2 Efficiency

For evaluating the performance, we executed C-2PO on the reachability set of the SV-COMP 2024 benchmarks [Bey24] and compared the performance of the four possible configurations against *base*. Each task was given a timeout of 900 seconds and a memory limit of 15 GB. We used the configuration of the analyzer for SV-COMP but removed the *var_eq* analysis. We ran the analyzer with each of the four configurations and once without C-2PO.

Table 6.2: Summary of efficiency experiments on the reachability set of the SV-COMP 2024 benchmarks. All data is measured with respect to the *base* analysis. The total amount of tasks was 15015. The table shows the median and 95% slowdown of each analysis with respect to *base*. Additionally, the number of tasks that are unreachable for *base*, but are proven correct by the analysis is given, as well as the number of tasks that base was able to prove, but which ran into a timeout or out-of-memory error for C-2PO.

	c-2po_1	c-2po_2	c-2po_3	c-2po_4
median slowdown	1,08	1,07	1,07	1,07
95-percentile slowdown	2,80	2,76	2,70	2,68
additional correct tasks	20	20	20	20
additional timeouts/out-of-memory	79	79	74	73

We observed that the analysis with C-2PO has a 95th percentile slowdown of at most a factor 2,8 compared to the base setting, indicating that 95% of tests in the benchmark take at most around three times as long as the base setup. The median slowdown is 1.07-1.08.

The runtime difference of the two configurations varied significantly between the benchmarks. In some cases, *base* timed out while C-2PO proved the property in less than 30 seconds. However, in most cases, C-2PO incurred a noticeable performance overhead compared to *base*. Given the wide variance in performance, the mean slowdown is not a significant indicator.

The results suggest that the more precise join operation (used by c-2po_1 and c-2po_3) is slightly less efficient than the partition-join, as predicted. Moreover, the normal form algorithm (used by c-2po_1 and c-2po_2) is not more efficient than the other algorithm for *equal*, contrary to our expectations. However, the difference in performance between the four configurations is not significantly large. In total, the results demonstrate that C-2PO scales reasonably well and that the four possible configurations are mostly equivalent in terms of efficiency.

The reachability set of the SV-COMP benchmarks contains over 20,000 tests where the reachability of an error function is to be decided. The analysis with C-2PO infers extra properties, which do not necessarily improve the verdict, as finding the correct verdict may require intricate arithmetic reasoning, which is orthogonal to our goal here. For this reason, our analysis only succeeds in proving the verdict for 20 SV-COMP tests that *base* could not prove. They are the same 20 passed tests for each of the four configurations. As expected, those 20 tests contain complicated pointer or struct manipulations. Due to the computational overhead, approximately 80 of the tests that

could be proven by *base* timed out with C-2PO out of more than 2000 passed verdicts.

The results are summarized in Table 6.2.

Given the negligible differences observed across the four configurations, it is reasonable to conclude that these configurations are equivalent in practice in terms of performance and precision. Therefore, the most practical choice may be the most straightforward configuration to implement—specifically, the one that avoids computing the normal form and employs the less precise join operation (c-2po_4).

The implementation is publicly available under...

put on zenodo?

7 Related Work

This chapter gives an overview of the related work on congruence closure, pointer analyses, and weakly-relational analyses.

Existing congruence closure algorithms are discussed and compared to the quantitative congruence closure introduced by Seidl et al. [Sei+24a], which is used by the C-2PO analysis. Moreover, multiple approaches to pointer analysis are presented, as well as weakly-relational analyses employing domains different from pointers.

The 2-Pointer Logic [Sei+24a] is based on these foundational concepts, offering a weakly-relational pointer analysis that utilizes a quantitative congruence closure of pointer expressions. Building upon this approach, the C-2PO analysis was developed by defining the necessary operations for abstract interpretation.

7.1 Congruence Closure

The congruence closure algorithm is widely used in formal verification for reasoning about the equivalence of terms. Different variations of congruence closure were introduced in the early 1980s. Downey et al. [DST80] described a congruence closure algorithm where each element is directly mapped to its representative, thus not requiring the union-find data structure. The methods of Nelson et al. [NO80] and Shostak [Sho06] are based on a union-find data structure [Tar79]. They define the procedure directly on graphs instead of using a finite automaton, thus allowing the construction of the congruence closure of terms that contain arbitrary uninterpreted function symbols of any arity. On the other hand, our quantitative congruence closure is restricted to a single uninterpreted function $*$ and extended to handle simple arithmetic operations. This quantitative congruence closure for 2-Pointer Logic was introduced by Seidl et al. [Sei+24a].

The congruence closure algorithm can also be viewed as a rewrite system for finding canonical normal forms of terms by replacing subterms with their representatives, as in [Kap97; BTV03]. There exists an extension with integer offsets for this approach, as described by Nieuwenhuis et al. [NO03]. The extension does not use union-find, and it considers a binary function $\cdot(f, a)$, where the offset is added to the second argument, for example, $\cdot(f, a + 2)$. Arbitrary terms with uninterpreted functions of any arity can be rewritten using only the function $\cdot$ by *curryfying* the terms. Our quantitative

congruence closure is less expressive, as we only consider the unary uninterpreted function $*$.

Join algorithms for congruence closure are described by Gulwani et al. [GTN04], where the join operation is defined over the congruence closure described as a rewrite system. Here, this join was adapted for the congruence closure that uses a quantitative finite automaton. We also introduced an alternative join that does not consider the information deriving from the quantitative automaton but only from the partition of the union-find data structure. It is less precise, but it can find almost the same number of invariants as the more precise join in practical examples.

7.2 Pointer Analysis

Numerous pointer analyses exist for C programs, which are used by static analysis tools. As C is widely used in crucial software, such as operating systems, device drivers, and embedded systems, it is essential to be able to automatically prove the desired properties of the pointers used in these programs. For a comprehensive overview of pointer analyses, see Smaragdakis et al. [SB15].

The recent work on pointer analysis can be roughly divided into three categories: fast flow- and context-insensitive analyses, non-relational context-sensitive analyses, and very precise shape analyses.

Andersen [And94] and Steensgaard [Ste96] proposed highly efficient but relatively unprecise analyses for large programs. The analysis remembers a set of possible addresses for each pointer without distinguishing between different offsets inside a memory block. These analyses are flow-insensitive and context-insensitive, meaning they do not take into account a specific program point or call context but summarize the possible values of each variable in one set of addresses for the whole program. The main advantage of this analysis is its efficiency, even for extremely large programs. However, in order to prove intricate properties, more precise analyses are needed.

The same idea can be applied in a context- and flow-sensitive analysis by tracking a set of possible addresses for each pointer at each program point and call context. This approach is used in multiple practical C analyzers, such as Mopsa [MOM21], Frama-C [Bau+21; BMP24] and Goblint [Voj+16]. This non-relational analysis is more precise than the flow-insensitive analysis, but when the sets are not singletons, it can no longer infer relational properties about the pointers. Our relational analysis is, therefore, an excellent complement to these analyses, as it can infer additional properties about the pointers, i.e., must-equalities and disequalities between terms made up of pointers, dereferencing, and pointer arithmetic. C-2PO uses the non-relational pointer analysis by Vojdani et al. [Voj+16] in combination with the newly implemented relational analysis

to augment the precision. The non-relational analysis infers disequalities between two pointer variables whose sets of possible values do not intersect. These disequalities are then used by the C-2PO analysis.

Shape analyses are a third category of pointer analyses, which are extremely precise and can infer complex properties about pointers and about the shape and content of data structures. Some examples are methods that use separation logic [OHe19; ILR21], or graph-based representations [DPV13] to abstract the shapes of data structures in the heap. Kreiker et al. [KSV10] introduced a shape analysis that considers overlapping data structures. These analyses are precise but very expensive in terms of time and memory consumption. C-2PO is a compromise between the precision of shape analysis and the efficiency of less precise analyses by only considering relations between pairs of pointers.

Moreover, the C-2PO domain is comparable to the domain introduced by Miné [Min06], which, similarly to C-2PO, considers dereferencing and pointer arithmetic. Pointer values are interpreted as a tuple of a variable and an offset. However, in [Min06], the abstraction tracks a set of possible pointer cells that the variables may point to and a set of possible values for the offset. In contrast, we track must-equalities and disequalities without considering concrete values, which makes it possible to express different properties than the domain in [Min06]. The domain in [Min06] assumes that malloc is not used in the program, while our analysis explicitly handles the malloc operator. This other domain also explicitly handles union types, which are not explicitly considered in our analysis.

7.2.1 Weakly-Relational Analyses

The idea of restricting the number of variables considered in relational analyses has been precedently used for non-pointer domains, such as numerical domains and strings. They are so-called *weakly-relational* analyses, as they only consider relations between pairs of variables. For example, for numerical analyses, the octagon domain [Min01] is a well-known weakly-relational domain, which tracks relationships between pairs of variables and a constant. It is a simplified version of the polyhedra domain [SPV17], which tracks relationships between arbitrary sets of variables. The same idea has been applied to non-numerical domains by Seidl et al. [Sei+24b].

Here, we use this concept for pointer analysis by tracking equalities and disequalities between pairs of pointer expressions. The advantage of weakly-relational analyses is their simplicity: by focusing on simple propositions, we have been able to provide polynomial-time algorithms for the implementation of our analysis in an abstract interpreter. Therefore, our analysis complements the computationally expensive full-fledged shape analyses.

8 Conclusion

We designed and implemented a weakly-relational pointer analysis for C programs, called C-2PO. It is a static analysis based on abstract interpretation for C programs, capable of inferring relational properties between pairs of pointer expressions that contain dereferences and pointer arithmetic.

The basic principles of abstract interpretation are presented, and then a concrete semantics is introduced that is the foundation for our abstract analysis. Then, the 2-Pointer Logic [Sei+24a] is described, as well as its representation and operations for computing the closure of a conjunction of propositions and for changing the set of terms that are considered by the analysis. The 2-Pointer Logic can express equalities and disequalities between pairs pointer expressions that contain the addition of integer constants, as well as dereferences. Here, it is extended with block disequalities, which model that pointer arithmetic is only valid within the same memory block.

The abstract domain of C-2PO is represented by a representation of 2-Pointer Logic conjunctions, including a congruence closure of the equalities and representations of quantitative disequalities and block disequalities.

Operations for the analysis by abstract interpretation are described, including the *equal*, *meet*, *join*, *widening*, and *narrowing* operations. Two algorithms for the *equal* and *join* operations are presented, which differ in their precision and efficiency. Transfer functions for assignments and dynamic memory allocation are defined. For interprocedural analysis, the *enter* and *combine* functions are described.

The analysis was integrated into the static analyzer Goblint [Voj+16]. The precision of the analysis was evaluated on 29 litmus tests and 11 real-world programs taken from the code for the GNU Core Utilities. In order to evaluate the performance of the analysis, we ran the analysis on the SV-COMP 2024 benchmarks and compared the performance to the performance of Goblint without the C-2PO analysis.

Our experiments demonstrate that our analysis scales reasonably for large programs and that it infers some properties that the Goblint analyzer without C-2PO was not able to infer. The difference in precision and performance between the two algorithms for the *join* operation and the two possibilities for the *equal* operation is evaluated, but no significant difference was found. Therefore, the simpler algorithms are preferred for their easier implementation.

In order to improve the precision of the analysis and be able to infer more types of

program behavior and possible undefined behavior, some future work is proposed in the next section.

8.1 Future Work

An interesting property that could be inferred by the analysis is the correct usage of mutexes. For example, if a mutex is locked in one thread and it is unlocked in a different thread, then it is important to be sure that the unlocked mutex is still *equal* to the locked mutex. In order to prove this, the analysis must be extended to handle multi-threaded programs, by defining the behavior of the analysis when a thread is created and when shared memory is accessed.

Another possible extension is the explicit handling of undefined pointers. A separate equivalence class could be added to the congruence closure that represents the set of all undefined pointers, i.e., those that are uninitialized or have been freed. This extension would allow the analysis to detect invalid pointer accesses and double frees. Additional closure rules must be added to express that an undefined pointer with a certain offset is still undefined, and that an undefined pointer is different from any other pointer, even other pointers in the same equivalence class.

Similarly, it could be investigated if it is useful to track null pointers. As for the undefined pointers, a separate equivalence class for null pointers can be added. This would allow the analysis to detect undefined behaviour relating to null-pointer dereferences.

Abbreviations

List of Figures

List of Tables

Bibliography

[And94] L. O. Andersen. "Program Analysis and Specialization for the C Programming Language." PhD thesis. Copenhagen, Denmark: University of Copenhagen, May 1994.

[Bau+21] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. "The dogged pursuit of bug-free C programs: the Frama-C software analysis platform." In: *Commun. ACM* 64.8 (2021), pp. 56–68. DOI: 10.1145/3470569.

[Bey24] D. Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024." In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*. Ed. by B. Finkbeiner and L. Kovács. Vol. 14572. Lecture Notes in Computer Science. Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15.

[BMP24] D. Bühler, A. Maroneze, and V. Perrelle. "Abstract Interpretation with the Eva Plug-in." In: *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*. Ed. by N. Kosmatov, V. Prevosto, and J. Signoles. Cham: Springer International Publishing, 2024, pp. 131–186. ISBN: 978-3-031-55608-1. DOI: 10.1007/978-3-031-55608-1_3.

[BTV03] L. Bachmair, A. Tiwari, and L. Vigneron. "Abstract Congruence Closure." In: *J. Autom. Reasoning* 31 (Oct. 2003), pp. 129–168. DOI: 10.1023/B:JARS.0000009518.26415.49.

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by R. M. Graham, M. A. Harrison, and R. Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CC92] P. Cousot and R. Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation." In: *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*. Ed. by M. Bruynooghe and M. Wirsing. Vol. 631. Lecture Notes in Computer Science. Springer, 1992, pp. 269–295. DOI: 10.1007/3-540-55844-6_142.

[Cou21] P. Cousot. *Principles of Abstract Interpretation*. The MIT Press, 2021. ISBN: 9780262044905.

[DPV13] K. Dudka, P. Peringer, and T. Vojnar. "Byte-Precise Verification of Low-Level List Manipulation." In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by F. Logozzo and M. Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 215–237. DOI: 10.1007/978-3-642-38856-9_13.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. "Variations on the Common Subexpression Problem." In: *J. ACM* 27.4 (Oct. 1980), pp. 758–771. ISSN: 0004-5411. DOI: 10.1145/322217.322228.

[Ghi+24] R. Ghidini, J. Erhard, M. Schwarz, and H. Seidl. "C-2PO: A Weakly Relational Pointer Domain: "These Are Not the Memory Cells You Are Looking For"." In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '24), October 22, 2024, Pasadena, CA, USA*. To appear. ACM, 2024. DOI: 10.1145/3689609.3689994.

[GTN04] S. Gulwani, A. Tiwari, and G. C. Necula. "Join Algorithms for the Theory of Uninterpreted Functions." In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*. Ed. by K. Lodaya and M. Mahajan. Vol. 3328. Lecture Notes in Computer Science. Springer, 2004, pp. 311–323. DOI: 10.1007/978-3-540-30538-5_26.

[HH12] R. W. Helmut Seidl and S. Hack. *Compiler Design: Analysis and Transformation*. Springer Berlin, Heidelberg, 2012. ISBN: 978-3-642-17547-3.

[ILR21] H. Illous, M. Lemerre, and X. Rival. "A relational shape abstract domain." In: *Formal Methods Syst. Des.* 57.3 (2021), pp. 343–400. DOI: 10.1007/S10703-021-00366-4.

[Kap97] D. Kapur. "Shostak's Congruence Closure as Completion." In: *Rewriting Techniques and Applications, 8th International Conference, RTA-97, Sitges, Spain, June 2-5, 1997, Proceedings*. Ed. by H. Comon. Vol. 1232. Lecture Notes in Computer Science. Springer, 1997, pp. 23–37. DOI: 10.1007/3-540-62950-5_59.

[KSV10] J. Kreiker, H. Seidl, and V. Vojdani. "Shape Analysis of Low-Level C with Overlapping Structures." In: *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*. Ed. by G. Barthe and M. V. Hermenegildo. Vol. 5944. Lecture Notes in Computer Science. Springer, 2010, pp. 214–230. DOI: 10.1007/978-3-642-11319-2_17.

[Min01] A. Miné. "The Octagon Abstract Domain." In: *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*. Ed. by E. Burd, P. Aiken, and R. Koschke. IEEE Computer Society, 2001, p. 310. DOI: 10.1109/WCRE.2001.957836.

[Min06] A. Miné. "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics." In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*. Ed. by M. J. Irwin and K. D. Bosschere. ACM, 2006, pp. 54–63. DOI: 10.1145/1134650.1134659.

[MOM21] R. Monat, A. Ouadjaout, and A. Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions." In: *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*. Ed. by C. Dragoi, S. Mukherjee, and K. S. Namjoshi. Vol. 12913. Lecture Notes in Computer Science. Springer, 2021, pp. 323–345. DOI: 10.1007/978-3-030-88806-0_16.

[NO03] R. Nieuwenhuis and A. Oliveras. "Congruence Closure with Integer Offsets." In: *Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings*. Ed. by M. Y. Vardi and A. Voronkov. Vol. 2850. Lecture Notes in Computer Science. Springer, 2003, pp. 78–90. DOI: 10.1007/978-3-540-39813-4_5.

[NO80] G. Nelson and D. C. Oppen. "Fast Decision Procedures Based on Congruence Closure." In: *J. ACM* 27.2 (Apr. 1980), pp. 356–364. ISSN: 0004-5411. DOI: 10.1145/322186.322198.

[OHe19] P. W. O'Hearn. "Separation logic." In: *Commun. ACM* 62.2 (2019), pp. 86–95. DOI: 10.1145/3211968.

[RY20] X. Rival and K. Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, 2020. ISBN: 9780262043410.

[SB15] Y. Smaragdakis and G. Balatsouras. "Pointer Analysis." In: *Found. Trends Program. Lang.* 2.1 (2015), pp. 1–69. DOI: 10.1561/2500000014.

[Sei+24a] H. Seidl, J. Erhard, M. Schwarz, and S. Tilscher. "2-Pointer Logic." In: *Taming the Infinities of Concurrency - Essays Dedicated to Javier Esparza on the Occasion of His 60th Birthday*. Ed. by S. Kiefer, J. Křetínský, and A. Kučera. Vol. 14660. Lecture Notes in Computer Science. Springer, 2024, pp. 281–307. DOI: 10.1007/978-3-031-56222-8_16.

[Sei+24b] H. Seidl, J. Erhard, S. Tilscher, and M. Schwarz. "Non-numerical weakly relational domains." In: *International Journal on Software Tools for Technology Transfer* (June 2024), pp. 1–16. DOI: 10.1007/s10009-024-00755-0.

[Sho06] R. Shostak. "Deciding Combinations of Theories." In: vol. 31. Jan. 2006, pp. 209–222. ISBN: 3-540-11558-7. DOI: 10.1007/BFb0000061.

[SPV17] G. Singh, M. Püschel, and M. T. Vechev. "Fast polyhedra abstract domain." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM, 2017, pp. 46–59. DOI: 10.1145/3009837.3009885.

[Ste96] B. Steensgaard. "Points-to Analysis in Almost Linear Time." In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by H. Boehm and G. L. S. Jr. ACM Press, 1996, pp. 32–41. DOI: 10.1145/237721.237727.

[Tar79] R. E. Tarjan. "A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets." In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 110–127. DOI: 10.1016/0022-0000(79)90042-4.

[Voj+16] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. "Static race detection for device drivers: the Goblint approach." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by D. Lo, S. Apel, and S. Khurshid. ACM, 2016, pp. 391–402. DOI: 10.1145/2970276.2970337.