Conjunctive Abstract Interpretation using Paramodulation*

Or Ozeri, Oded Padon, Noam Rinetzky, and Mooly Sagiv

School of Computer Science, Tel Aviv University, Israel

Abstract. Scaling static analysis is one of the main challenges for program verification in general and for abstract interpretation in particular. One way to compactly represent a set of states is using a formula in *conjunctive normal form* (*CNF*). This can sometimes save exponential factors. Therefore, CNF formulae are commonly used in manual program verification and symbolic reasoning. However, it is not used in abstract interpretation, due to the complexity of reasoning about the effect of program statements when the states are represented this way.

We present algorithms for performing abstract interpretation on CNF formulae recording equality and inequalities of ground terms. Here, terms correspond to the values of variables and of addresses and contents of dynamically allocated memory locations, and thus, a formula can represent pointer equalities and inequalities. The main idea is the use of the rules of paramodulation as a basis for an algorithm that computes logical consequences of CNF formulae, and the application of the algorithm to perform joins and transformers.

The algorithm was implemented and used for reasoning about low level programs. We also show that our technique can be used to implement best transformers for a variant of *Connection Analysis* via a non-standard interpretation of equality.

1 Introduction

Arguably, the greatest challenge in abstract interpretation [6] is managing the trade-off between the cost and the precision of the abstraction. This trade-off appears in an especially stark light in the handling of disjunction. An abstract domain that is closed under logical disjunction allows the analysis to enumerate cases, greatly enhancing its precision. However, this can quickly lead to an explosion of cases and intractability. An alternative is to use a Cartesian/conjunctive abstract domain. In this case, we are limited to logical conjunctions of facts drawn from rather simple abstract domains. This approach has the disadvantage that it cannot capture correlations between the components of the abstraction, thus quickly loses precision.

In this paper, we consider points between these two extremes. Our abstract domain will consist of logical formulae in *conjunctive normal form* (*CNF*). For atomic formulae, we support pointer equalities where ground terms correspond to pointer expressions. This approach is consistent with the way in which manually constructed program invariants are written. That is, they tend to consist of a large conjunction of relatively simple formulae containing a few disjunctions. Note that in contrast to standard may- and must-

* This work is supported by EU FP7 ERC grant agreement n^o [321174].

points-to analysis (see, e.g., [19]), CNF formulae permit conditional information. For example, consider the following code fragment:

$$\texttt{if } (\texttt{x} \neq \texttt{0}) \texttt{ then y} := [\texttt{x}];\ \texttt{if } (\texttt{y} \neq \texttt{0}) \texttt{ then t} := [\texttt{y}]$$

which loads into y the contents of the memory location pointed to by x, provided the value of the latter is not *null* (0), and then, if the value assigned to y is not null, loads into t the contents of the memory location that y now points to. The conjunction of the following clauses, where $a \implies b$ is used as syntactic sugar for $\neg a \vee b$, records the conditional aliasing generated by the above code fragment:

$$x \neq 0 \implies y = [x] \text{ and } y \neq 0 \implies t = [y]\ .$$

A key advantage of using CNF formulae is that they can be exponentially more succinct then DNF formulae used, e.g., in predicate abstraction. For example, to express the information recorded by the above clauses, we need four disjuncts:

$$\begin{array}{l} x = 0 \wedge y = 0 \qquad\qquad\ , \quad x \neq 0 \wedge y = [x] \wedge y = 0\ , \\ x = 0 \wedge y \neq 0 \wedge t = [y]\ , \text{ or } \ x \neq 0 \wedge y = [x] \wedge y \neq 0 \wedge t = [y]\ . \end{array}$$

The greatest challenge in our domain is reasoning about the effect of program statements and program joins. The reason is that even a simple statement, e.g., variable copy, can have a complicated effect on a CNF formula. Moreover, it may require rather complicated reasoning about the interactions between different clauses. For example, computing the effect of the statement $\texttt{y} := \texttt{NULL}$ on the set of states described by the above CNF formula requires considering both conjuncts to observe that the set of post-states can be described by the following conjuncts:

$$y = 0 \text{ and } x \neq 0 \wedge [x] \neq 0 \implies t = [[x]]\ .$$

This is in contrast to disjunctive domains where the effect of transformers is computed distributively by applying them to each disjunct. Indeed, disjunctive domains are incremental in nature.

We solve the last challenge by adapting a technique from theorem proving called *paramodulation* (see, e.g., [15]). Paramodulation is a refinement of the *resolution* [16] procedure for reasoning about equalities. The main idea is to implicitly represent equalities avoiding the costs of transitive inferences. However, utilizing paramodulation in abstract interpretation is not a panacea. First, paramodulation is usually applied in a context of checking validity, and we need to compute logical consequences. Second, in the context of abstract interpretation, the transformers are repeatedly computed many times, which can be expensive. Third, it is not clear how to ensure that the reasoning does not lead to plethora of terms leading to non-scalability and divergence of the static analysis.

Main Results The main results of this paper can be summarized as follows:

1. We develop a parametric abstract domain comprised of CNF formulae. Particular abstract domains can be instantiated by setting different bounds on the sizes of clauses and terms. This allows to modify the precision/performance tradeoff in our analyses simply by varying these bounds. (Section 3)

2. We describe a technique for computing join operators (Section 3) and applying abstract transformers (Section 4) based on a novel algorithm for performing semantic reduction [6] (Section 5). The algorithm is based on finding the logical consequences of CNF formulae. In general, consequence finding is a hard problem and requires many calls to expensive SAT solver. However, for reasoning about equalities, the problem is somewhat easier since we can compactly represent equivalence classes and leverage the properties of equivalence relations and of function congruence [14][1] to detect important consequences using paramodulation [15].
3. We apply our technique to the problem of *Connection Analysis* [9] (Section 6). Connection analysis treats the program's heap (dynamically allocated memory) as an undirected graph, and infers whether two pointer variables may never point to the same weakly connected heap-component. We obtain a new abstract interpretation algorithm for connection analysis using heap-connectivity as a non-standard interpretation of equality, and adapting the abstract meaning of statements accordingly. Furthermore, we prove that the resulting transformers are the *best* (most precise conservative) *abstract transformers* [6], and demonstrate that our analysis can infer properties that existing connection analyses [3, 9] cannot.
4. We implemented our analysis algorithms in a proof-of-concept analyzer. We report on an initial empirical evaluation using a few small benchmarks, and discuss certain heuristics, based on *ordered paramodulation* (see, e.g., [15]) that can improve the performance of the analysis but may, in principal, hurt its precision. (Section 7).

2 Programming Language and Running Example

We formalize our results using a simple imperative programming language for sequential procedure-less programs. We assume the reader is familiar with the notion of *control-flow graphs* (see, e.g., [1]), and only define the necessary terminology.

Syntax. A program $P = (V, E, v_{entry}, v_{exit}, c)$ is a *control-flow graph (CFG)*: Nodes $v \in V$ correspond to *program points* and edges $e \in E$ indicate possible transfer of control. Every edge e is associated with a primitive command $c(e)$. In this paper, with the exception of Section 6, we use the primitive commands listed in Figure 1, which are typical for pointer-manipulating programs. They include the ubiquitous `skip` command, conditionals involving variables, assignments to variables, dynamic memory allocation, (possibly destructive) accesses to memory locations via pointers, and assignment commands $v := \mathtt{f}(v_1, \ldots, v_k)$ involving any function (operator) `f` coming from an arbitrary set F of *pure* functions over values (`f` is a pure function if its evaluation does not change the state. For example, in the running example, $F = \{+\}$, where $+$ is the arithmetic addition operator). Our technique is independent of the choice of the functions included in F, as, unless stated otherwise, our analysis treats them as uninterpreted functions.

Formally, our language allows for only two forms of conditionals: $v = v'$ and $v \neq v'$. However, we can encode arbitrary conditionals by using a function which returns, e.g., 1 if the condition holds and 0 otherwise. For example, We can encode the *less-than* relation $v < v'$ as a function $\mathtt{f}_<(v, v') =$ if $v < v'$ then 1 else 0.

[1] Function congruence means that if x and y are equal then, for any function $f(\cdot)$, so are $f(x)$ and $f(y)$. Congruence naturally generalizes to functions with multiple arguments.

command	Comment
`skip`	skip.
`assume`$(v \bowtie v')$	Acts as `skip` if $v \bowtie v'$ holds, and blocks the execution otherwise.
$v := \mathtt{c}$	Assigns the constant value c to a local variable v.
$v := v'$	Copies the value of v' to v. We require that v is different from v'.
$v :=$ `malloc`(v')	Dynamically allocates a block of memory comprised of v' locations.
$v := [v']$	Loads the value stored in (the dynamically allocated) memory location v' to a local variable v. We require that v is different from v'.
$[v] := v'$	Stores the value of v' at (the dynamically allocated) memory location v.
$v :=$ `f`$(v_1, \ldots, v_k)$	Assigns the result of applying a pure function (operator) `f` to v. We require that v is different from $v_1, \ldots, v_k$ and expect `f` not to be $[\,\cdot\,]$.

Fig. 1. Primitive commands. We assume the programing language allows to apply an arbitrary, but finite, set of pure functions $\mathtt{f} \in F$. We leave the set F unspecified. $\bowtie$ is either $=$ or $\neq$.

Operational semantics. Let $\mathit{Val} \supseteq \mathbb{N}$ be the semantic domains of *values*. A concrete state $\sigma = \langle \rho, h \rangle \in \Sigma$ is comprised of an *environment* $\rho : \mathsf{Var} \to \mathit{Val}$ mapping a fixed, but arbitrary, finite set of variable names to their values, and a *heap* $h : \mathbb{N}^+ \rightharpoonup \mathsf{Val}$ mapping allocated memory locations (addresses), represented as non zero natural numbers, to values. The semantics of programs is defined using a *concrete transition relation* $T \subseteq \Sigma \times \Sigma$ (See Appendix A). For simplicity, we assume that accessing an unallocated address blocks the execution.[2] With the exception of this, the semantics is standard.

Running example. Figure 2 shows `find_last`, a program which we use as our running example. For clarity, the figure also provides an implementation of the program in C. `find_last` gets as an input a pointer `s` to a nil-terminated string, i.e., a '`\0`'-terminated array of characters, and a character `c`. It returns a pointer to the last occurrence of `c` in `s`, or `NULL` if `c` does not appear in `s`. Our analysis successfully verifies that the `find_last` either returns `NULL` or a pointer to a memory location containing the same value as `c`.

3 Equality-based Parametric Abstract Domain

We define a parametric family of abstract domains $(\Phi^{k,d}_{\mathcal{F}}, \sqsubseteq)$ comprised of ground quantifier-free logical formulae in first order logic (FOL) in conjunctive-normal form (CNF).[3] The only allowed predicate symbol is equality, applied to terms constructed using function and constant symbols coming from a fixed, but arbitrary, finite vocabulary $\mathcal{F}$. Technically, we represent formulae as sets of *clauses*, where each clause C is a set of *literals*. A literal L is either an *atom* or its negation. An atom $s \simeq t$ is an instance of the

[2] Our analysis does not prove memory safety. Thus, our results can be adapted to the case where such an operation leads to an error state in the following way: The properties we infer hold unless the program performs a memory error, e.g., dereferencing a null-valued pointer or accessing an unallocated memory address.

[3] A ground formula is a formula which does not contain free variables.

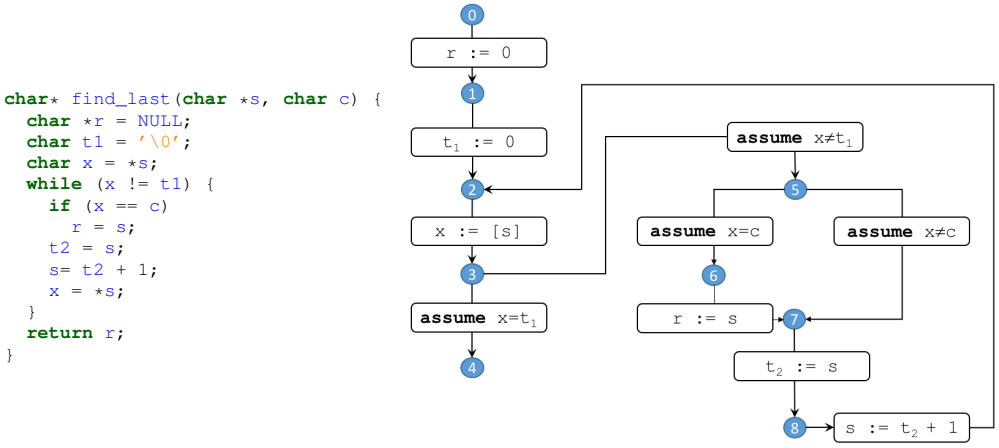

Fig. 2. The running example. Program points are depicted as solid numbered circles. Edges are decorated with rectangles labeled with primitive commands.

equality predicate $\simeq$ applied to two terms, s and t. The abstract domain is parameterized by the vocabulary $\mathcal{F}$, and by two natural numbers: k (dubbed *max-clause*) is the maximal number of literals in a clause, and d (dubbed *rank*) is the maximal number of nested function applications that may be used to construct a term.

Figure 3 defines the abstract states in $\Phi_{\mathcal{F}}^{k,d}$ for arbitrary $\mathcal{F}$, k, and d. We denote the set of well-formed terms over vocabulary $\mathcal{F}$ by $\mathcal{T}_{\mathcal{F}}$ and the *rank* of a term by $\mathrm{rank}(t)$. For example, the ranks of the terms 0, $\mathrm{a}+1$, and $\mathrm{f}(\mathrm{a})+1$, are zero, one, and two, respectively.[4] We denote the subset of $\mathcal{T}_{\mathcal{F}}$, containing all terms with rank d or less by

$$\mathcal{T}_{\mathcal{F}}^{d} \stackrel{\text{def}}{=} \{t \in \mathcal{T}_{\mathcal{F}} \mid \mathrm{rank}(t) \leq d\} \,.$$

In the following, we often refer to an abstract state $\varphi \in \Phi_{\mathcal{F}}^{k,d}$ as a *formula*. Also, for clarity, we write atoms and literals as $s \simeq t$ and $s \not\simeq t$ instead of $\{s,t\}$ and $\neg\{s,t\}$, respectively. Note that, e.g., $s \simeq t$ and $t \simeq s$ denote the same atom. Also note that our restrictions ensure that $\Phi_{\mathcal{F}}^{k,d}$ is finite for any allowed choice of $\mathcal{F}$, k, and d. We denote the set of CNF formulae over vocabulary $\mathcal{F}$ involving terms of rank d or less by $\Phi_{\mathcal{F}}^{d}$ and the set of all CNF formulae over vocabulary $\mathcal{F}$ by $\Phi_{\mathcal{F}}$:

$$\Phi_{\mathcal{F}}^{d} = \bigcup_{0 \leq k} \Phi_{\mathcal{F}}^{k,d} \qquad\qquad \Phi_{\mathcal{F}} = \bigcup_{0 \leq d} \Phi_{\mathcal{F}}^{d} \,.$$

The Least-Upper Bound (Join) Operator. The abstract domain is ordered by *subsumption*.

$$\varphi_1 \sqsubseteq \varphi_2 \iff \forall C_2 \in \varphi_2 .\, \exists C_1 \in \varphi_1 .\, C_1 \subseteq C_2 \,.$$

[4] We assume the reader is familiar with the notions of well-formedness, ranking and meaning of terms in FOL, and do not formalize them here. For a formal definition, see, e.g., [15, Chap. 7].

$$\varphi \in \Phi_{\mathcal{F}}^{k,d} \stackrel{\text{def}}{=} \mathcal{P}\left(\mathcal{C}_{\mathcal{F}}^{k,d}\right) \quad \text{Formulae} \qquad L \in \mathcal{L}_{\mathcal{F}}^{d} \stackrel{\text{def}}{=} \{A, \neg A \mid A \in \mathcal{A}_{\mathcal{F}}^{d}\} \quad \text{Literals}$$
$$C \in \mathcal{C}_{\mathcal{F}}^{k,d} \stackrel{\text{def}}{=} \left\{C \subseteq \mathcal{L}_{\mathcal{F}}^{d} \,\middle|\, |C| \leq k\right\} \quad \text{Clauses} \qquad A \in \mathcal{A}_{\mathcal{F}}^{d} \stackrel{\text{def}}{=} \left\{\{s,t\} \mid s,t \in \mathcal{T}_{\mathcal{F}}^{d}\right\} \quad \text{Atoms}$$

Fig. 3. Abstract states (formulae). k and d are arbitrary natural numbers.

$$\sigma \models \varphi \iff \forall C \in \varphi.\, \sigma \models C \qquad \sigma \models s \simeq t \iff \sigma(s) = \sigma(t)$$
$$\sigma \models C \iff \exists L \in C.\, \sigma \models L \qquad \sigma \models s \not\simeq t \iff \sigma \not\models s \simeq t$$

Fig. 4. Forcing semantics

The least upper bound operator in $\Phi_{\mathcal{F}}^{k,d}$, denoted by $\cdot \sqcup^k \cdot$, is defined as follows:

$$\varphi_1 \sqcup^k \varphi_2 = \{C_1 \cup C_2 \mid C_1 \in \varphi_1\ ,\ C_2 \in \varphi_2\} \cap \mathcal{C}_{\mathcal{F}}^{k,d}\,.$$

It easy to see that under the subsumption order $\sqsubseteq$, the set $\emptyset$ is the top element $\top$ of the domain and that $\{\emptyset\}$ is its bottom element $\bot$.

Concretization function. A concrete state $\sigma \in \Sigma$ *models* a formula φ if it satisfies all of its clauses, where a clause C is satisfied if at least of one its literals, which is either an equality or inequality holds. Figure 4 defines the forcing semantics $\sigma \models \varphi$ which determines whether a state $\sigma \in \Sigma$ satisfies a formula $\varphi \in \Phi_{\mathcal{F}}$. The interpretation of a term $t \in \mathcal{T}_{\mathcal{F}}$ in a state σ, denoted by $\sigma(t)$, is defined inductively in a standard way.[4] A formula (abstract state) represents the set of concrete states which satisfies it:

$$\gamma \colon \Phi_{\mathcal{F}}^{k,d} \to \mathcal{P}(\Sigma) \stackrel{\text{def}}{=} \gamma(\varphi) = \{\sigma \in \Sigma \mid \sigma \models \varphi\}\,.$$

Note that $\gamma(\emptyset) = \Sigma$ and $\gamma(\{\emptyset\}) = \emptyset$.

Example 1. We use the function symbol $[\,\cdot\,]$ to represent the contents of the heap. For example, let $\sigma = \langle \rho, h \rangle$, where $\rho = [x \mapsto 1, y \mapsto 2]$ and $h = [1 \mapsto 2, 2 \mapsto 3]$, then

$$\sigma \models x \simeq 1 \quad , \quad \sigma \models [x] \simeq 2 \quad , \text{ and } \quad \sigma \models [[x]] \simeq 3.$$

Example 2. In our running example, our analysis proves that the formula resulting at program point 4, where `find_last` terminates, contains the clause $\{r \simeq 0, [r] \simeq c\}$. The models of this clause are all the states where `r`, the return value of `find_last`, is either `NULL` or points to a memory location containing the same value as `c`.

We say that a formula φ_1 is a *semantic consequence* of a formula φ_2, denoted by $\varphi_1 \models \varphi_2$, if for any concrete state $\sigma \in \Sigma$ such that $\sigma \models \varphi_1$, it holds that $\sigma \models \varphi_2$. The subsumption order is an under-approximation of the *implication order*, i.e, for any concrete state σ and any formulae φ_1, φ_2, if $\varphi_1 \sqsubseteq \varphi_2$ then $\varphi_1 \models \varphi_2$. The other direction is not true. Thus, as it is often the case in logical domains (see, e.g., [8]), our domain does not have an abstraction function which maps every set of concrete state to its most precise representation in the abstract domain. For example, $\varphi_a = \{\{x \simeq y, y \not\simeq z\}, \{y \simeq z\}\}$ and $\varphi_b = \{\{x \simeq z\}, \{y \simeq z\}\}$ represent the same set of concrete states but neither $\varphi_a \not\sqsubseteq \varphi_b$ nor $\varphi_b \not\sqsubseteq \varphi_a$. Furthermore, there might be formulae such that $\varphi_1 \sqsubset \varphi_2$ and yet $\varphi_2 \models \varphi_1$. For example, let $\varphi_c = \{\{x \simeq y\}, \{y \simeq z\}\}$. It is easy to see that $\varphi_c \sqsubset \varphi_a$ and that $\varphi_2 \models \varphi_1$. This suggests, that analyses in our domain can benefit from applying *semantic reduction*, as we discuss below.

3.1 Semantic Reduction

A *semantic reduction operator* SR over an abstraction domain $(A, \sqsubseteq)$ with respect to a given concretization function $\gamma : A \to \Sigma$ is an idempotent total function $SR : A \to A$ which allows to find a possibly more precise representation of the set of concrete states S represented by an abstract state $a \in A$. Technically, SR should have the following properties: $SR(a) \sqsubseteq a$, $SR(SR(a)) = SR(a)$, and, most importantly, $\gamma(SR(a)) = \gamma(a)$ [6]. Semantic reduction operators can help improve the precision of an abstract interpretation algorithm because both the abstract transformers and the join operator are monotonic. Thus, applying a semantic reduction operator before, e.g., applying the join operator, may result in an abstract state which represents less concrete states than the abstract state resulting form a naive join, i.e.,

$$\forall a_1, a_2 \in A. \, \gamma(SR(a_1) \sqcup SR(a_2)) \subseteq \gamma(a_1 \sqcup a_2) \, .$$

It is sound to use SR in this way because for any $a \in A$, it holds that $\gamma(SR(a)) = \gamma(a)$.

We leverage the fact that our domain tracks equality between terms to define a semantic reduction operator $SR^{k,d}$ which finds semantic consequences stemming from the reflexivity, symmetry and transitivity properties of the equality predicate and the insensitivity of functions to substitutions of equal arguments (i.e., if x and y are equal then $f(x) = f(y)$ for any function f), that can be expressed in the abstract domain $\Phi^{k,d}_{\mathcal{F}}$.

We improve the precision of the analysis by employing two semantic reduction steps before applying the join operator.

$$\varphi_1 \hat{\sqcup}^{k,d} \varphi_2 = SR^{k,d}(\varphi_1) \sqcup^k SR^{k,d}(\varphi_2) \, .$$

Before formally defining $SR^{k,d}(\cdot)$, we show it can indeed improve the analysis precision.

Example 3. Let $\varphi_1 = \{\{x \simeq z\}\}$ and $\varphi_2 = \{\{x \simeq y\}, \{y \simeq z\}\}$. Note that $\varphi_1, \varphi_2 \in \Phi^{1,0}_{\{x,y,z\}}$, and that $\varphi_1 \sqcup^1 \varphi_2 = \emptyset = \top$. However, by leveraging the transitivity of $\simeq$, we have $\gamma(\varphi_2) = \gamma(\varphi_3)$, for $\varphi_3 = \varphi_2 \cup \{\{x \simeq z\}\}$. Note that $\varphi_1 \sqcup^1 \varphi_3 = \{\{x \simeq z\}\} \sqsubset \top$.

Example 4. Consider the formulae $\varphi_1 = \{\{r \simeq s\}, \{x \simeq c\}, \{x \simeq [s]\}\}$, which contains clauses which appear in the formula propagated to program point 7 of our running example from the true branch, and $\varphi_2 = \{\{r \simeq 0, c \simeq [r]\}\}$, which is a loop invariant, and propagated to program point 7 from the false-branch. If max-clause is two and max-rank is one, we get $\varphi_1 \sqcup^1 \varphi_2 = \emptyset = \top$, whereas $SR^{2,1}(\varphi_1) \sqsubseteq \{\{c \simeq [r]\}\}$, $\varphi_1 \hat{\sqcup}^{2,1} \varphi_2 = \varphi_2$, which helps establish that φ_2 is indeed a loop invariant.

Saturation-based semantic reduction. Algorithm 1 shows a saturation-based implementation of our semantic reduction operator. The procedure invokes a function $\mathsf{PMStep}(C_1, C_2)$ which, given two clauses $C_1, C_2 \in \mathcal{C}^{k,d}_{\mathcal{F}}$, returns a set of clauses $S \subseteq \mathcal{C}^{2k-1,2d}_{\mathcal{F}}$ which are semantic consequences of the two, i.e., $\{C_1, C_2\} \models S$. Technically, $\mathsf{PMStep}(C_1, C_2)$ is implemented using paramodulation, which, intuitively, computes logical consequences that can be inferred from the aforementioned properties of the equality predicate $\simeq$ by *directly* applying the paramodulation rules to C_1 and C_2. We provide a short technical overview of paramodulation in Section 5, and here consider it as a black box.

Algorithm 1: Semantic reduction procedure based on saturation.

Procedure: $SR^{k,d}$
Input: $\varphi \in \Phi^{k,d}_{\mathcal{F}}$
Output: $\varphi' \in \Phi^{k,d}_{\mathcal{F}}$ such that $\varphi \subseteq \varphi'$ and $\gamma(\varphi) = \gamma(\varphi')$
$\varphi' \leftarrow \varphi$
while $\exists C_1, C_2 \in \varphi'.\ \mathsf{PMStep}(C_1, C_2) \cap \mathcal{C}^{k,d}_{\mathcal{F}} \not\subseteq \varphi'$ **do**
    take $C_1, C_2 \in \varphi'$ s.t. $\mathsf{PMStep}(C_1, C_2) \cap \mathcal{C}^{k,d}_{\mathcal{F}} \not\subseteq \varphi'$
    $\varphi' \leftarrow \varphi' \cup \left(\mathsf{PMStep}(C_1, C_2) \cap \mathcal{C}^{k,d}_{\mathcal{F}}\right)$

Algorithm 1 operates by saturating the intial set of clauses φ with respect to the consequence finding procedure PMStep. This is obtained by iteratively applying PMStep to clauses in φ', and to the resulting clauses, and so forth, until reaching a fixpoint, where we say the set of clauses φ' is saturated. Note that for any $C \in \varphi'$, we also apply $\mathsf{PMStep}(C, C)$. In practice, the saturation procedure is implemneted by keeping a worklist of new clauses to which PMStep should be applied. Algorithm 1 is guaranteed to terminate because $\mathcal{C}^{k,d}_{\mathcal{F}}$ is finite for any fixed $\mathcal{F}$, k, and d.

4 Abstract Transformers

Figure 5, defines the abstract semantics $[\![c]\!]^\sharp : \Phi^{k,d}_{\mathcal{F}} \to \Phi^{k,d}_{\mathcal{F}}$ over our equality-based abstract domain for arbitrary $\mathcal{F}$, k, and d. The transformers use the auxiliary function $\mathrm{remSym}(\varphi, s) = \varphi \cap \Phi^{k,d}_{\mathcal{F}\setminus\{s\}}$. The function takes as input a formula φ and a symbol s, which may be either a variable name or a function symbol, and returns a formula $\varphi' \subseteq \varphi$ which is comprised of all the clauses in φ that do not include symbol s. Intuitively, $\mathrm{remSym}(\varphi, s)$ acts as a *havoc(·)* operator [2] which forgets any information regarding s. For example, for $\varphi = \{\{r \simeq v\}, \{[u] \simeq [w]\}, \{[r] \not\simeq v, u \simeq w\}\}$,

$$\mathrm{remSym}(\varphi, r) = \{\{[u] \simeq [w]\}\} \qquad \text{and} \qquad \mathrm{remSym}(\varphi, [\,]) = \{\{r \simeq v\}\}\,.$$

Our programming language has, essentially, two types of primitive statements: assignments and assume commands. Thus, the transformers follow two different patterns:

The transformers for assignments of the form $v := e$ (where v is a variable and e is an expression) take an abstract state φ and first performs a semantic reduction, then remove from the resulting formula any clause which contains v, and, finally, add a clause which records the equality of v and e. This mimics the overwriting (destructive) nature of assignments. For example, the set of concrete states that arises at program point 6 in our running example, can be represented by the abstract state

$$\varphi = \{\{r \simeq 0, [r] \simeq c\}, \{x \simeq [s]\}, \{x \not\simeq t_1\}, \{x \simeq c\}\}$$

Applying the transformer for the instruction `r := s` to φ, we first apply a semantic reduction on φ, yielding:

$$SR^{2,1}(\varphi) = \varphi \cup \{\{c \not\simeq t_1\}, \{[s] \not\simeq t_1\}, \{[s] \simeq c\}\}$$

$$
\begin{aligned}
[\![v := c]\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \mathrm{remSym}(SR^{k,d}(\varphi), v) \cup \{\{v \simeq c\}\} \\
[\![v := v']\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \mathrm{remSym}(SR^{k,d}(\varphi), v) \cup \{\{v \simeq v'\}\} \\
[\![v := \mathtt{f}(v_1, \ldots, v_k)]\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \mathrm{remSym}(SR^{k,d}(\varphi), v) \cup \{\{v \simeq \mathtt{f}(v_1, \ldots, v_k)\}\} \\
[\![v := [v']]\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \mathrm{remSym}(SR^{k,d}(\varphi), v) \cup \{\{v \simeq [v']\}\} \\
[\![[v] := v']\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \mathrm{remSym}(SR^{k,d}(\varphi), [\,]) \cup \{\{[v] \simeq v'\}\} \\
[\![v := \mathtt{malloc}(v')]\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \mathrm{remSym}(SR^{k,d}(\varphi), v) \\
[\![\mathtt{assume}(v = v')]\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \varphi \cup \{\{v \simeq v'\}\} \\
[\![\mathtt{assume}(v \neq v')]\!]^\sharp(\varphi) &\stackrel{\text{def}}{=} \varphi \cup \{\{v \not\simeq v'\}\}
\end{aligned}
$$

Fig. 5. Abstract transformers.

Next, we apply remSym to forget all facts that mention the old value of `r`:

$$\mathrm{remSym}(SR^{2,1}(\varphi), r) = SR^{2,1}(\varphi) \setminus \{\{r \simeq 0, [r] \simeq c\}\}$$

And finally, we add an equality indicating the new value of `r` equals the value of `s`. Thus,

$$[\![r := s]\!]^\sharp(\varphi) = \{\{x \simeq [s]\}, \{x \not\simeq t_1\}, \{x \simeq c\}, \{c \not\simeq t_1\}, \{[s] \not\simeq t_1\}, \{[s] \simeq c\}, \{r \simeq s\}\}\,.$$

The transformers for `assume( )` commands, leverage our restriction that the only allowed conditions are equalities and inequalities, which can be readily translated into the domain. In fact, our restrictions ensure that $[\![\mathtt{assume}(\,)]\!]^\sharp$ is the *best abstract transformer* [6]. For example, consider the abstract state $\varphi = \{\{r \simeq 0, [r] \simeq c\}, \{x \simeq [s]\}, \{t_1 \simeq 0\}\}$ which represents all the states that can arise at program point 3 of our running example. Applying the transformer of $\mathtt{assume}(\mathtt{x} = \mathtt{t_1})$, we get:

$$[\![\mathtt{assume}(x = t_1)]\!]^\sharp(\varphi) = \{\{r \simeq 0, [r] \simeq c\}, \{x \simeq [s]\}, \{t_1 \simeq 0\}, \{x \simeq t_1\}\}\,,$$

which indicates that when the program terminates at program point 4, the value of variable `x` is (unsurprisingly) zero.

We note that in our analysis, we only apply $\mathrm{remSym}(\varphi, s)$ where s is either a variable or to the function $[\,\cdot\,]$. Also note that the interpretation of destructive updates (assignment to heap locations) in our analysis is, by design, extremely conservative: It leads to a loss of all the information we had regarding the contents of the heap in the abstract state prior to the assignment due to possible aliasing.

5 Consequence Finding by Paramodulation

In Section 3, we presented a semantic reduction operator which utilizes a function $\mathsf{PMStep}(C_1, C_2)$ that finds all the *direct* consequences that can be inferred from two clauses C_1, C_2 based on properties of the equality relation. In this section, we explain how to implement PMStep using *paramodulation* [15]. In Section 7, we describe an over-approximation of PMStep so it becomes more efficient, but less precise. In our experiment, we use the more efficient operator.

$$\frac{C_1 \cup \{l \simeq r\} \qquad C_2 \cup \{s \simeq t\}}{C_1 \cup C_2 \cup \{s[r]_p \simeq t\}}\ s|_p = l \quad \text{SUPERPOSITION RIGHT} \qquad \frac{C \cup \{s \not\simeq s\}}{C} \quad \text{EQUALITY RESOLUTION}$$

$$\frac{C_1 \cup \{l \simeq r\} \qquad C_2 \cup \{s \not\simeq t\}}{C_1 \cup C_2 \cup \{s[r]_p \not\simeq t\}}\ s|_p = l \quad \text{SUPERPOSITION LEFT} \qquad \frac{C \cup \{s \simeq t\} \cup \{s \simeq t'\}}{C \cup \{t \not\simeq t'\} \cup \{s \simeq t'\}} \quad \text{EQUALITY FACTORING}$$

Fig. 6. Ground rules of paramodulation.

5.1 A Bird's-Eye View of Paramodulation

Paramodulation is a reasoning technique which is used for automated theorem proving. It is a semi algorithm for determining the unsatisfiability of a formula φ in first order logic with equality. It is refutation-complete: given an unsatisfiable formula, it is guaranteed to terminate. Technically, it is a specialization of *resolution*-based reasoning [16] to first-order logic with equality. In the following, we provide an informal overview of paramodulation. For a formal description, see, e.g., [15].

Roughly speaking, paramodulation is based on the use of special inference rules which leverage the reflexivity, symmetry, and transitivity of the equality relation, as well as *function congruence* [14][1], i.e., the insensitivity of functions valuation with respect to substitution of equal arguments. More technically, paramodulation works by inferring logical consequences in an iterative manner: It keeps generating consequences, i.e. new clauses which are logically derived from the clauses of the input formula or from previously derived ones by applying the *paramodulation rules*. The rules are sound: any logical consequence $\{C_1, C_2\} \vdash C$ is also a semantic consequence $\{C_1, C_2\} \models C$. When used for satisfiability checking, paramodulation is often coupled with a search strategy, e.g., ordered paramodulation [15], which limits the applications of the rules while maintaining refutation completeness. In our setting, we do not aim to refute the satisfiability of a formula, but rather use the rules of paramodulation to infer logical consequences. Thus, we avoid discussing the search strategies often used together with paramodulation, and focus on describing the underlying rules.

Paramodulation uses four inference rules, dubbed the *paramodulation rules* [15]. These rules, simplified to the case where only ground clauses are considered, are shown in Figure 6. The *superposition right* rule is applied to a pair of clauses. One clause contains the literal $l \simeq r$, for some arbitrary terms l and r, and the other clause contains the literal $s \simeq t$, for some arbitrary terms s and t, such that l is a *sub-term* of s. Technically, l is a sub-term of s if either l is (syntacticly) equal to s, or s is of the form $f(s_1, \ldots, s_k)$, for some function symbol f, and l is a sub-term of s_i, for some $i = 1..k$. Figure 6 records this requirement as a side condition to the rule using the notation $s|_p = l$. The latter holds if l is a sub-term of s in a specific *position* p. Note that l can occur as a sub-term of l multiple times. For example, the term a appears twice in the term $f(g(a), a)$. Following [15], we use the notation $s|_p$ to identify a particular sub-term of s and the notation $s[r]_p$ to denote the term obtained by substituting the sub-term of s at position p with r. For example, if p_1 and p_2 identify, respectively, the first and second occurrences of a in $f(g(a), a)$ then $f(g(a), a)[b]_{p_1} = f(g(b), a)$ and $f(g(a), a)[b]_{p_2} = f(g(a), b)$. For brevity, we avoid formalizing the rather standard notions of sub-terms, positions, and substitutions, (see, e.g., [15]), and rely on the reader intuitive understanding, instead.

Algorithm 2: PMStep: A procedure for computing direct logical consequences.

```
Procedure: PMStep
Input: C_1, C_2 ∈ C_F^{k,d}
Output: φ ∈ Φ_F^{2k-1,2d} such that {C_1, C_2} ⊨ φ
if C_1 ≠ C_2 then
    φ ← SuperpositionRight(C_1, C_2) ∪ SuperpositionRight(C_2, C_1) ∪
        SuperpositionLeft(C_1, C_2) ∪ SuperpositionLeft(C_2, C_1)
else
    φ ← EqualityResolution(C_1) ∪ EqualityFactoring(C_1)
```

Example 5. We demonstrate the use of the superposition rules using clauses $C_1 = \{u + v \not\simeq 0, v \simeq u\}$ and $C_2 = \{[v] \simeq 0, v \not\simeq u\}$. The only relevant choice for term l in either of the rules is to be v. Using superposition-right and setting s to be $[v]$, we can infer the clause $\{u + v \not\simeq 0, v \not\simeq u, [u] \simeq 0\}$. Using superposition-left and setting s to be v, we infer the clause $\{u + v \not\simeq 0, [v] \simeq 0, u \not\simeq u\}$.

The *equality resolution* rule applies to a single clause containing a literal of the form $s \not\simeq s$. Such a literal is always false due to reflexivity of $\simeq$. The rule thus allows to remove the literal from the clause that contains it.

Example 6. Applying the equality-resolution rule to the clause $\{u + v \not\simeq 0, [v] \simeq 0, u \not\simeq u\}$ allows to infer the clause $\{u + v \not\simeq 0, [v] \simeq 0\}$.

The *equality factoring* rule can be applied to a clause containing two literals of the form $s \simeq t$ and $s \simeq t'$. The rule allows replacing one of the equalities by an inequality between t and t'.

Example 7. Let $C = \{[u] \simeq [v], [u] \simeq 0\}$. There are two possible inferences by the equality factoring rule: by taking s to be $[u]$ and t' to be 0 we get $\{[v] \not\simeq 0, [u] \simeq 0\}$, and by taking s to be $[u]$ and t' to be $[v]$ we get $\{0 \not\simeq [v], [u] \simeq [v]\}$.

Note that superposition rules operate on two clauses, while the equality rules operate on a single clause. Also note that it may be possible to apply the superposition rules to a single clause and itself (i.e., $C_1 = C_2$). However, this will yield a clause which is subsumed by the input clause, and we thus avoid it.

5.2 Direct Consequence Finding

Algorithm 2 presents an implementation of the PMStep procedure which was used as a key ingredient of our semantic reduction algorithm (see Section 3). By abuse of notation, we refer to each of the paramodulation rules shown Figure 6 as a function which returns the set of clauses that rule can infer.

PMStep uses the superposition rules when invoked with two different clauses, and the equality resolution and equality factoring rules when it is given two identical clauses as input. For example, let $C = \{s \not\simeq s, t \not\simeq t\}$, then $\mathsf{PMStep}(C, C) = \texttt{EqualityResolution}(C) = \{\{s \not\simeq s\}, \{t \not\simeq t\}\}$. Note that $\mathsf{PMStep}(C, C)$ does not

return the empty clause, although it is a consequence of C, because it is not a direct consequence: Obtaining it, requires applying the equality resolution rule twice. Since the paramodulation rules are sound, it follows that the returned set of clauses (formula) φ satisfies the requirement that $\{C_1, C_2\} \models \varphi$.

When applying a paramodulation rule to clauses in $\Phi_{\mathcal{F}}^{k,d}$, the maximal clause size of any inferred clause is $2k-1$ (the so-called *resolved literals* do not appear in the result clause). Under the same conditions, the maximal rank of any inferred clause is $2d$, as the only way the rules create new terms is via substitution. Thus, the formula φ returned by PMStep is in $\Phi_{\mathcal{F}}^{2k-1,2d}$. Termination of the PMStep procedure is self-evident.

We note that although the paramodulation rules are refutation complete, they are incomplete when used, as in our case, for consequence finding. That is to say that they may not be able to produce all semantic consequences, not even up to subsumption. For example, applying the paramodulation rules to the formula $\varphi = \{\{f(a) \not\simeq f(b)\}\}$ yields no consequences, as none of the rules can be applied. However, the clause $\{a \not\simeq b\}$ is a semantic consequence of φ. An unfortunate by product of this incompleteness is that the abstract transformers, defined in section 4, are incomplete, even if we place no bound on the maximal size or rank of clauses. For example, let $\varphi = \{\{f(a, v) \not\simeq f(b, v)\}\}$. Applying our transformers, we get that $[\![v := c]\!]^\sharp(\varphi) = \{\{v \simeq c\}\}$, while the best abstract transformer must also infer the clause $\{a \not\simeq b\}$. In spite of this incompleteness, our initial experiments (see Section 7) show that the paramodulation rules can be used to prove interesting properties of programs.

6 Connection Analysis

In the previous section, we developed an analysis which tracks equalities and inequalities between variables and memory locations, and supports limited disjunction via CNF. The atomic literals represented equalities and inequalities, and the analysis benefited from the fact that equality is an equivalence relation (i.e. it is reflexive, transitive, and symmetric) to employ paramodulation. In this section, we present an additional application of the techniques presented so far, that uses a different domain, where the atomic literals represent an equivalence relation other than equality, namely *heap connectivity*. The key idea is to reinterpret $s \simeq t$ to denote the fact that s is *heap-connected* to t. Because this relation is an equivalence relation, and because our domain here will not include any function symbols, the paramodulation rules can be used to obtain sound abstract transformers. As we shall see, the obtained transformers are the best abstract transformers [6] for this domain.

Static analysis of heap connectivity is known as *connection analysis* [9]. Connection analysis arose from the need to automatically parallelize sequential programs. It is a kind of pointer analysis that aims to prove that two pointer variables can never point to the same (undirected) *heap component*: We say that two heap objects are *heap-connected* in a state σ when it is possible to reach from one object to the other, following paths in the heap of σ, ignoring pointer direction. Two variables are *heap-connected* when they point to connected heap objects. Connection analysis soundly determines whether variables are *not* heap-connected. Despite its conceptual simplicity, connection analysis is flow-sensitive, and the effect of program statements is non-distributive.

```
(node*, node*) build_lists() {
   node *x = (node*) malloc(sizeof(node)), *y = (node*) malloc(sizeof(node));
   while (*) {
      node *z = (node*) malloc(sizeof(node)); z->next = null;
      if (*) { z->next = x; x = z
      } else { z->next = y; y = z;  } }
   return (x, y);}
```

Fig. 7. Program `build_lists()`. `node` is a record with a single field of type `node`$*$ named `next`.

Example 8. Program `build_lists`, shown in Figure 7, constructs two disjoint singly-linked lists. For clarity, the program is written in a C-like language. The program first allocates the tails of the lists, and then, iteratively, prepends a newly allocated object to one of the lists. The program returns pointers to the heads of the two lists it created. Our analysis proves that the two lists are disjoint, by proving the loop invariant $x \not\simeq y$. We note that existing connection analyses [3, 9] cannot prove this property.

Concrete semantics. In the context of this section, a concrete state $\sigma = \langle \rho, h \rangle \in \Sigma$ is comprised of an environment ρ and a heap h. The heap $h = \langle V, E \rangle$ is a finite labeled graph, where V is a finite set of objects and $E \subseteq V \times \mathsf{Fields} \times V$ is set of labeled edges that represent pointer fields of objects. The environment $\rho : \mathsf{Var} \rightharpoonup V$ maps some of the variables to objects in the heap. Variables not mapped by ρ, or fields of an object for which no outgoing edge exists, are said to be equal to $null$. The heap-connected relation in state $\sigma = \langle \rho, h \rangle$ is determined by ρ and by the weakly connected components of h. The programming language allows read and write operations to both variables and object fields, and operations to allocate new objects. Its operational semantics, described by a transition relation $\mathrm{TR}_c \subseteq \Sigma \times \Sigma$ for every command c, is standard, and omitted.

Abstract semantics. The abstract domain of existing connection analyses [3, 9] is based on a partitioning of the program variables into disjoint sets. Two variables are in the same set, often referred to as a *connection set*, when the two *may* be heap-connected. Note that at the loop header, it is possible that `x` and `z` are heap-connected and that `y` and `z` are heap-connected. However, it is never the case that all three variables are heap-connected. Existing connection analyses maintain a single partitioning of the variables, and hence have to conservatively determine that `x` and `y` might be heap-connected.

In a way, the abstract domains of [3, 9] maintain a conjunction of clauses of the form $x \not\simeq y$, which record that `x` and `y` are definitely not heap-connected (In [9], clauses of the form $x \not\simeq null$ which records that variable `x` is definitely not equal to $null$ are also tracked). In contrast, our analysis allows to track more complicated correlations between variables using clauses that contain more than one literal. More specifically, our analysis also uses clauses that record whether the value of a variable is definitely null or whether two variables are *connected*, where a variable `x` is *connected* to variable `y` in σ if they are heap-connected in σ or if the value of `x` and `y` is $null$ in σ. We record the connected-relation utilizing a non standard interpretation of the equality predicate $\cdot \simeq \cdot$:

$\sigma \models s \simeq t \iff s$ **is** connected to t in σ $\qquad$ $\sigma \models s \simeq null \iff$ the value of s in σ is $null$

$\sigma \models s \not\simeq t \iff \sigma \not\models s \simeq t$ $\qquad$ $\sigma \models s \not\simeq null \iff \sigma \not\models s \simeq null$

Note that $\cdot \simeq \cdot$ is an equivalence relation: It is reflexive, symmetric, and transitive.

$$
\begin{aligned}
\tau^{\mathrm{TR}}_{v\,:=\,0} &\stackrel{\text{def}}{=} \mathsf{frame}(v) \cup \{\{v' \simeq 0'\}\} \\
\tau^{\mathrm{TR}}_{v\,:=\,u} &\stackrel{\text{def}}{=} \mathsf{frame}(v) \cup \{\{v' \simeq u'\}\} \\
\tau^{\mathrm{TR}}_{v\,:=\,\mathtt{malloc}} &\stackrel{\text{def}}{=} \mathsf{frame}(v) \cup \{\{v' \not\simeq 0'\}\} \cup \{\{v' \not\simeq u'\} \mid u \in \mathsf{Var} \setminus \{v\}\} \\
\tau^{\mathrm{TR}}_{v\,:=\,u\mathtt{->}f} &\stackrel{\text{def}}{=} \mathsf{frame}(v) \cup \{\{v' \simeq u', v' \simeq 0'\}, \{u \not\simeq 0\}\} \\
\tau^{\mathrm{TR}}_{v\mathtt{->}f\,:=\,0} &\stackrel{\text{def}}{=} \{\{v \not\simeq 0\}\} \cup \{\{x \not\simeq y \implies x' \not\simeq y'\} \mid x, y \in \mathsf{Var} \cup \{0\}\} \cup \\
&\qquad \{\{x \simeq y \wedge x \not\simeq v \implies x' \simeq y'\} \mid x, y \in \mathsf{Var} \cup \{0\}\} \cup \\
&\qquad \{\{x \simeq y \wedge x \simeq v \implies (x' \simeq v' \vee y' \simeq v' \vee x' \simeq y')\} \mid x, y \in \mathsf{Var}\} \\
\tau^{\mathrm{TR}}_{v\mathtt{->}f\,:=\,u} &\stackrel{\text{def}}{=} \{\{v \not\simeq 0\}, \{u \not\simeq 0\}, \{u' \simeq v'\}\} \cup \{\{x \simeq y \implies x' \simeq y'\} \mid x, y \in \mathsf{Var} \cup \{0\}\} \cup \\
&\qquad \{\{x \not\simeq y \wedge x \not\simeq v \wedge x \not\simeq u \implies x' \not\simeq y'\} \mid x, y \in \mathsf{Var} \cup \{0\}\}
\end{aligned}
$$

$$
\mathsf{frame}(v) \stackrel{\text{def}}{=} \{\{x \simeq y \implies x' \simeq y'\}, \{x \not\simeq y \implies x' \not\simeq y'\} \mid x, y \in \mathsf{Var} \setminus \{v\} \cup \{0\}\}
$$

Fig. 8. The abstract transition relation of commands in the connection analysis.

Abstract domain. As connection analysis concerns only connection between variables, we restrict the vocabulary $\mathcal{F}$ to include only the constants representing the local variables, and thus have no function symbols. This means that by construction, all terms would be constants, having max-rank of zero. In this setting, the number of terms is finite, and hence the clause size is also bounded; there can be at most $O(|\mathsf{Var}|^2)$ literals (Var is the set of variables in the program.) Our analysis uses a domain $\Phi^{k,0}_{\mathsf{Var}}$ with k large enough to ensure that neither the semantic reduction operator nor the join operator filters out clauses due to their size. In this setting the join operator is additive. For these reasons, we do not use semantic reduction in the join.

Best abstract transformers. We define the abstract transformers in a uniform way using *two-vocabulary* formulae which describe the abstract transition relation of every command: Given a command c, we define a formula τ^{TR}_c over the vocabulary $\mathcal{F}_2 = \mathcal{F} \cup \mathcal{F}'$ comprised of constants $v \in \mathcal{F}$ pertaining to program variables at the pre-state and primed constants $v' \in \mathcal{F}' = \{v' \mid v \in \mathcal{F}\}$ pertaining to program variables at the post-state (For technical reasons, we also include a primed version of $null$). We note that the finiteness of our abstract domain allows us to encode any transformer in this way. The abstract transformer for c is defined as follows:

$$
[\![c]\!]^\sharp(\varphi) \stackrel{\text{def}}{=} \mathrm{remSym}(SR(\varphi \cup \tau^{\mathrm{TR}}_c), \mathcal{F})[v/v' \mid v \in \mathcal{F}]\,.
$$

$[\![c]\!]^\sharp$ conservatively determines the effect of c on the set of states that φ represents in four stages: Firstly, it conjoins φ with τ^{TR}_c, thus effectively restricting the abstract transition relation to consider only pre-states represented by φ. Secondly, it applies our semantic reduction operator to find possible semantic consequence. This helps to propagate information encoded in φ regarding the pre-state into conjuncts containing only constants coming from $\mathcal{F}'$ describing the post-state. Thirdly, the transformer removes any conjuncts which mention variables coming from the pre-state.[5] Finally, the transformer

[5] $\mathrm{remSym}(SR(\cdot, \mathcal{F})$ is the obvious extension of the symbol elimination function $\mathrm{remSym}(\cdot, v)$, described in Section 4, from a single symbol v to the removal of all symbols coming from $\mathcal{F}$.

replaces every primed constant with the corresponding unprimed constant, thus obtaining again a formula representing the set of post-states which is expressible in our domain. Note that, in particular, if $\emptyset \in SR(\varphi)$ then $\emptyset \in abs[\![c]\!](\varphi)$.

Figure 8 shows the formulae encoding the abstract transition relation of primitive commands. Assigning $null$ to a variable v records the fact that the connection relation between all other variables in the post state needs to be as in the pre-state (this is captures by the auxiliary formula $\mathsf{frame}(v)$) and that v is connected to $null$. Copying the value of a variable u into v has a similar effect, except that v is connected in the post state to u. Assigning v the address of a freshly allocated memory means that v cannot be connected to any other variable, and cannot be connected to $null$ as well. Successfully loading a value from the heap $\mathtt{y} := \mathtt{x} \rightarrow \mathtt{f}$ implies that $\mathtt{x}$ is not null (recall that our semantics blocks before dereferencing a null-valued pointer) and that either $\mathtt{y}$ becomes connected to $\mathtt{x}$ or it is nullified, in case this is the value of the f-field of u. Without loss of generality, we assume that every destructive update command $\mathtt{x} \rightarrow \mathtt{f} := \mathtt{y}$ in the program is replaced by a command $\mathtt{x} \rightarrow \mathtt{f} := \mathtt{null};\ \mathtt{if}\ (\mathtt{y} \neq \mathtt{null})\ \mathtt{x} \rightarrow \mathtt{f} := \mathtt{y}$ which nullifies the f-field of $\mathtt{x}$ and updates it later only to a non-$null$ value. Thus, applying the last transformer in Figure 8 is guaranteed to merge together the two heap components of $\mathtt{x}$ and $\mathtt{y}$. The most complicated transition relation pertains to the command $\mathtt{x} \rightarrow \mathtt{f} := \mathtt{null}$ which conservatively records that only the heap-component $\mathtt{x}$ can be affected. More specifically, the component can be split into two subcomponents, which, in turn, may split the set of variables connected to $\mathtt{x}$ into two arbitrary partitions.

Soundness. It is easy to see that the abstract transition relation indeed over-approximates the concrete one. Interestingly, it is still sound to use the paramodulation rules as means to perform semantic reduction because any logical consequence they derive is also a semantic consequence under the non-standard interpretation of $\simeq$.

Precision. We say that a pair of states σ_1, σ_2 satisfies a two-vocabulary formulae $\tau \in \Phi_{\mathcal{F}_2}$, denoted by $\langle \sigma_1, \sigma_2 \rangle \models \tau$, if $\sigma_1 \models \tau \cap \Phi_{\mathcal{F}}$ and $\sigma_2 \models \tau \cap \Phi_{\mathcal{F}'}$, i.e., σ_1 and σ_2 satisfy all the clauses of the formulae containing terms built using symbols coming $\mathcal{F}$ and $\mathcal{F}'$, respectively. The following theorem ensures that the abstract transition relation described by the formulae in Figure 8 is the most precise conservative transition relation.

Theorem 1. *Let c be a command. The following holds:*
(soundness) $\forall \sigma, \sigma' \in \Sigma.\, \langle \sigma, \sigma' \rangle \in \mathrm{TR}_c \implies \langle \sigma, \sigma' \rangle \models \tau_c^{\mathrm{TR}}$, *and*
(precision) $\forall \varphi, \varphi' \in \Phi_{\mathcal{F}}.\, (\forall \sigma, \sigma' \in \Sigma.\, \sigma \models \varphi \wedge \langle \sigma, \sigma' \rangle \in \mathrm{TR}_c \implies \sigma' \models \varphi') \implies (\forall \sigma, \sigma' \in \Sigma.\, \langle \sigma, \sigma' \rangle \models \varphi \wedge \tau_c^{\mathrm{TR}} \implies \sigma' \models \varphi')$.

The key reason for the transformers we obtain to be the best abstract transformers, is that restricting our domain to contain only terms corresponding to program variables (i.e., constant symbols) ensures that $\mathrm{remSym}(SR(\varphi), v)$ yields a *strongest* formula that is implied from φ and does not contain the symbol v.

Theorem 2. *Let $\varphi \in \Phi_{\mathcal{F}}$ be a ground CNF formula, where $\mathcal{F}$ contains only constant symbols. Let $C \in \Phi_{\mathcal{F}}$ be a clause such that $\varphi \models C$, and that v does not appear in C. It holds that* $\mathrm{remSym}(SR(\varphi), v) \models C$.

We can now prove that our abstract transformers are the *best* (most precise conservative) abstract transformers [6] in our domain. The proof goes in two stages. Firstly, we

Benchmark	C code # of lines	C code # of vars	LLVM code # of lines	LLVM code # of vars	k	d	Running time Unord	Running time Ord	Inv. Size CL	Inv. Size LI
`find_last`	10	6	37	14	2	1	30.49s	13.81s	6	9
`find_last`					2	2	85.92s	30.04s	7	10
`find_last`					2	3	191.43s	58.02s	7	10
`resource_manager`	34	14	74	17	2	∞	208.84s	40.32s	126	247
`cve_2014_7841`	40	11	68	21	2	∞	3.54s	0.94s	N/A	
`build_lists`	15	4	43	14	∞	0	—	11.12s	182	562

Fig. 9. Benchmarks characteristics and Experimental results. k is max-clause an d is max-rank. Unord and Ord stand for unordered and ordered paramodulation, respectively. A timeout of one hour was set for each test. CL and LI stand for the number of clauses and literals, respectively, in the loop invariant.

show in Theorem 1 that the abstract transition relation described by the formulae given in Figure 8 is the most precise conservative transition relation. Secondly, we use Theorem 2 to conclude that there cannot be a more precise formula in our abstract domain that can represent the possible post-states describe by $\varphi \cup \tau_c^{\mathrm{TR}}$.

Theorem 3. *The abstract transformers shown in Figure 8 are the best transformers.*

7 Implementation and Experimental Results

We implemented our analyses and applied them to analyze a few low-level pointer-manipulating programs. We noticed that most of the running time was spent in applying the paramodulation rules. Often, this happens right before a function symbol is eliminated. To improve the performance of our tool, we modified the implementation of the semantic reduction step in our transformers to use *ordered paramodulation* [15]. In a nutshell, ordered paramodulation is given an order over the function symbols $\mathcal{F}$, which it then extends to an order over terms, literals, and clauses in a standard way. Ordered paramodulation ensures that the rules are applied to literals according to their order. Nevertheless, ordered paramodulation is refutation complete. We leveraged the ordered execution of rules by setting the function symbol which is to about be eliminated to be the highest symbol in the order. Additionally, we forbid the use of a paramodulation rule in the case where none of the input clauses contain the symbol which is to be eliminated. These modifications greatly improve the performance of our tool. Theoretically, this modification reduces the precision of our transformers. However, in our experiments they did not. Another simple optimization we made, is to apply the equality resolution rule to every consequence clause that we generate by the paramodulation rules, as well as discarding any generated clause containing a trivially satisfiable literal of the form $s \simeq s$. This can help reduce the size of the clause and prevents the semantic reduction operator from filtering it out.

Experimental evaluation. Our analysis handles low level pointer programs, Thus, we implemented it to analyze Intel x86 binary code, and experimented with code generated by compiling C programs. Our tool, written in Python 2.7, compiles C programs using *GCC* Version 4.8.4 with all optimizations disabled, and then disassembles the generated

code using python package *distorm3* version 3.3.4. The tests were done using a single thread on a Windows 7 machine equipped with Intel Core i5-5300U and 8GB of RAM.

Figure 9 summarizes the characteristics of the analyzed procedures and the running time of the analysis, with different max-clause (k) and max-rank (d) parameters.

- `find_last` is our running example. We were able to prove that it either returns `NULL` or a pointer to a memory location containing the same value as `c`.
- `resource_manager` is a simulation of a controller for a microphone and a camera. It runs a state-machine which executes an infinite loop, where it receives commands from the user, such as start/end video call and start/end audio call. Our analysis proves that it always holds that if the camera is turned on, then so is the microphone.
- `CVE_2014_7841` is a simplification of a published null-pointer dereference vulnerability in the SCTP protocol implementation of the linux kernel [20]. Our analysis proves the suggested fix prevents the relevant null dereference errors.
- `build_lists` is the example shown Figure 7. Our connection analysis proved that the generated lists are disjoint. We note that existing analyses cannot prove this property because they maintain a single partitioning of the variables in every program point. In contrast, our analysis was able to prove for the end state the formula $x \not\simeq y$.

8 Related Work, Discussion, Conclusions, and Future Work

Abstract Interpretation [5] algorithms cope with a persistent tension between the precision of an analysis, specifically its ability to maintain relational and disjunctive information, and the cost of the analysis. The *reduced product* [6] and *reduced cardinal power* [6] operations are two examples of operations that increase the precision of an abstract domain, while making the analysis more expensive. The *reduced relative power* operation [10] is another such operation, which allows to keep implication correlations between two abstract domains. Applying this operation to an abstract domain and itself is known as *autodependency*. Such implication correlations can be viewed as CNF formulae where each clause contains exactly two literals. In the context of abstract interpretation of logic programs CNF have been considered from the lattice and logical perspective [4, 11, 13, 17]. In this context, the key contribution of our work is the use of paramodulation to implement both the abstract join and the abstract transformers.

Resolution, *superposition*, and *paramodulation* have been the subject of vast study in the theorem proving community (see, e.g., [15] for an introduction with extensive bibliography). While their most common use has been satisfiability checking, it has also been explored as a technique for *consequence finding* [12]. A technique called *kernel resolution* [7, 18] has been explored to obtain completeness for consequence finding. In this context, our work presents a new application for consequence finding in program analysis, as a basis for constructing abstract join and abstract transformers. In this work we explore the use of paramodulation for consequence finding, which was sufficient for our applications. More elaborate consequence finding techniques can be used to obtain more precise transformers and join operations, and this presents an interesting direction for future study.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.
3. G. Castelnuovo, M. Naik, N. Rinetzky, M. Sagiv, and H. Yang. Modularity in lattices: A case study on the correspondence between top-down and bottom-up analysis. In *12th International Static Analysis Symposium (SAS)*, 2015.
4. M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Log. Program.*, 25(3):249–274, 1995.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, New York, NY, 1979. ACM Press.
7. A. del Val. A new method for consequence finding and compilation in restricted languages. In J. Hendler and D. Subramanian, editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA.*, pages 259–264. AAAI Press / The MIT Press, 1999.
8. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, 2006.
9. R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *IJPP*, 1996.
10. R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theor. Comput. Sci.*, 216(1-2):159–211, 1999.
11. R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst.*, 20(5):1067–1109, Sept. 1998.
12. K. Inoue. Consequence-finding based on ordered linear resolution. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*, pages 158–164. Morgan Kaufmann, 1991.
13. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *LOPLAS*, 2(1-4):181–196, 1993.
14. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.
15. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.
16. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965.
17. F. Scozzari. Logical optimality of groundness analysis. *Theor. Comput. Sci.*, 277(1-2):149–184, Apr. 2002.
18. L. Simon and A. del Val. Efficient consequence finding. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 359–370. Morgan Kaufmann, 2001.

19. Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, Apr. 2015.
20. US-CERT/NIST. Vulnerability summary for cve-2014-7841. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7841, April 2014.

command	Side condition
$\langle\langle \rho, h\rangle, \texttt{skip}\rangle \rightsquigarrow \langle \rho, h\rangle$	
$\langle\langle \rho, h\rangle, \texttt{assum}(v \bowtie v')\rangle \rightsquigarrow \langle \rho, h\rangle$	$\rho(v) \bowtie \rho(v')$
$\langle\langle \rho, h\rangle, v := \texttt{c}\rangle \rightsquigarrow \langle \rho[v \mapsto c], h\rangle$	
$\langle\langle \rho, h\rangle, v := v'\rangle \rightsquigarrow \langle \rho[v \mapsto \rho(v')], h\rangle$	
$\langle\langle \rho, h\rangle, v := \texttt{f}\,(v_1, \ldots, v_k)\,\rangle \rightsquigarrow \langle \rho[v \mapsto f(\rho(v_1), \ldots, \rho(v_k))], h\rangle$	
$\langle\langle \rho, h\rangle, v := \texttt{malloc}(\,v')\rangle \rightsquigarrow \langle \rho[v \mapsto r], h[r + i \mapsto \_ \mid i = 0..\rho(v') - 1]\rangle$	$\forall 0 \leq i < \rho(v').\, h(r + i) = \bot$
$\langle\langle \rho, h\rangle, v := [v']\rangle \rightsquigarrow \langle \rho[v \mapsto h(\rho(v'))], h\rangle$	$h(\rho(v')) \neq \bot$
$\langle\langle \rho, h\rangle, [v] := v'\rangle \rightsquigarrow \langle \rho, h[\rho(v) \mapsto \rho(v')]\rangle$	$h(\rho(v)) \neq \bot$

Fig. 10. Small step operational semantics for primitive commands.

A Small Step Operational Semantics

Operational semantics. The concrete semantics of a program is defined with the help of a *concrete transition relation* $T \subseteq \Sigma \times \Sigma$ over a set of *concrete states* $\sigma \in \Sigma$. Figure 10 describes the effect of primitive statements as a relation $\langle \sigma, c\rangle \rightsquigarrow \sigma'$ which records the ability of a primitive statement c to transform memory state σ to memory state σ'. For simplicity, we assume that accessing an unallocated address blocks the execution. Our results can be easily adapted to the case where such an operation leads to an error state.

B Proofs

In this section we sketch the proofs of Theorems 1 and 2.

B.1 Proof of Theorem 1

Proof. The proof relies upon the following four claims:

Claim I. Formulae in our domain cannot distinguish between states with the same connectivity relation. More formally, we say that states σ_1 and σ_2 have *similar connectivity*, denoted by $\sigma_1 \sim \sigma_2$, if for any $\forall x, y \in \mathsf{Var} \cup \{0\}.\, \sigma_1 \models x \simeq y \iff \sigma_2 \models x \simeq y$. The claim states that

$$\forall \varphi'' \in \Phi_{\mathcal{F}}, \sigma_1, \sigma_2 \in \Sigma.\, \sigma_1 \sim \sigma_2 \implies (\sigma_1 \models \varphi'' \iff \sigma_2 \models \varphi'').$$

The claim holds because if two states agree on the interpretation of atoms, then they agree on the interpretation of formulae.

Claim II. The effect of every transition of the concrete semantics on the connectivity in a state σ can be explained by inspecting a similar *simple* state. More formally, we say that a state σ_0 is *simple* if every weakly connected component in σ_0 is comprised of two objects: a *head* object o and a *tail* object o' such that there is a single f-field of o which

points to o' and the value of every other field of o, or of o', is null. We denote the set of simple states by Σ_0. The claim states that

$$\forall \sigma, \sigma' \in \Sigma.\, (\langle \sigma, \sigma' \rangle \in \mathrm{TR}_c) \implies (\exists \sigma_0 \in \Sigma_0, \sigma'' \in \Sigma.\, \sigma \sim \sigma_0 \wedge \sigma' \sim \sigma'' \wedge \langle \sigma_0, \sigma'' \rangle \in \mathrm{TR}_c).$$

To show that the claim holds, it suffices to make the following two observations: Firstly, given a state σ, we can fabricate any similar simple state σ_0. Secondly, reviewing the possible effects of every command c on the connectivity in σ' and, in particular, on the similarities and differences between the connectivity in σ' and the connectivity in σ, it becomes clear that we can always fabricate the necessary simple states:

- The commands pertaining to the first four transformers in Figure 8, i.e., $\mathtt{v} := \mathtt{null}$, $\mathtt{v} := \mathtt{u}$, $\mathtt{v} := \mathtt{malloc}$, and $\mathtt{v} := \mathtt{u} \to \mathtt{f}$, do not change the heap connectivity, and thus, performing them on σ and on any simple state $\sigma_0 \sim \sigma$ necessarily leads to similar states, provided that when considering the last command, $\mathtt{v} := \mathtt{u} \to \mathtt{f}$, we make sure that u points to the head object in σ_0 and that the f-field of the latter points to the tail object, if the value of the f-field of the object pointed to by u in σ is not null, and that u points to the tail object in σ_0 otherwise.
- Similar care needs to be taken when considering the command $\mathtt{v} \to \mathtt{f} := 0$: In σ_0, the variables which remain connected to v in σ' need to point to the head object, and all other variables which were connected to v in σ need to point to the tail object in σ_0. Also, we need to pick a simple state σ_0 where the two objects are connected via an f-field. This ensures that nullifying this fields indeed splits the heap component.
- The simple states used when considering the command $\mathtt{v} \to \mathtt{f} := \mathtt{u}$ are such that v points to the tail object of its connected component. This ensures that the connected components of v and u would merge. (Note that our simplifying assumptions ensure that a command of the form $\mathtt{v} \to \mathtt{f} := \mathtt{u}$ is executed only when u is not null.)

Claim III. The abstract transition relation shown in Figure 8 are sound and precise with respect to the effect of commands on simple states:

$$\forall \sigma, \sigma' \in \Sigma.\, (\langle \sigma, \sigma' \rangle \models \tau_c^{\mathrm{TR}}) \iff (\exists \sigma_0 \in \Sigma_0, \sigma'' \in \Sigma.\, \sigma \sim \sigma_0 \wedge \sigma' \sim \sigma'' \wedge \langle \sigma_0, \sigma'' \rangle \in \mathrm{TR}_c).$$

The proof of this claim is similar to the proof of the previous claim: one merely needs to review the possible effects of every command c on the connectivity of states.

Claim IV. Let c be a command and $\varphi \in \Phi_{\mathcal{F}}$ be an unsatisfiable formula. It holds that $\varphi \cup \tau_c^{\mathrm{TR}}$ is not satisfiable.

Soundness. The first part of the theorem, i.e., the soundness of the abstract transition relation encoded by τ_c^{TR}, follows directly from Claims II and the "only if" ($\Leftarrow$) direction of claim III.

Precision. To prove the second part of the theorem, i.e., the precision of the abstract transition relation encoded by τ_c^{TR}, we need to consider two cases: If φ is not satisfiable, then Claim IV ensures that restricting the abstract transition relation to an empty set of pre states would not yield any post state, thus the last implication would hold vacuously. If φ is satisfiable, then assume by contradiction that there exist a formula φ' such that

$\forall \sigma, \sigma' \in \Sigma. \sigma \models \varphi \wedge \langle \sigma, \sigma' \rangle \in \mathrm{TR}_c \implies \sigma' \models \varphi'$, and states $\sigma, \sigma' \in \Sigma$ such that $\sigma \models \varphi$ and $\langle \sigma, \sigma' \rangle \models \varphi \wedge \tau_c^{\mathrm{TR}}$ but $\sigma' \not\models \varphi'$. By the "if" ($\Rightarrow$) direction of claim III, there exist a simple state $\sigma_0 \sim \sigma$ and a state $\sigma'' \sim \sigma'$ such that $\langle \sigma_0, \sigma'' \rangle \in \mathrm{TR}_c$. Hence, by Claim I, it holds that $\sigma_0 \models \varphi$ and $\sigma'' \not\models \varphi'$ (Recall that we assumed that $\sigma' \not\models \varphi'$). However, this is a contradiction to the assumption that φ' describes all the possible post-states that can arise by executing c on a state satisfying φ becasue $\langle \sigma_0, \sigma'' \rangle \in \mathrm{TR}_c$.

B.2 Proof of Theorem 2

Proof. If φ is not satisfiable, then the refutation completeness of paramodulation ensures that $SR(\varphi)$ includes the empty clause, and thus $\mathrm{remSym}(SR(\varphi), v) \models C$ because $\emptyset \models C'$ for any clause C'.

Assume φ is satisfiable. The semantic reduction operator returns a superset of the given formula, i.e., $\varphi \subseteq SR(\varphi)$. Thus, we get that $SR(\varphi) \models \varphi$. By assumption, $\varphi \models C$. Thus, $SR(\varphi) \models C$.

To ease the proof, we will base upon the refutation-completeness of *ordered paramodulation* (see [15]). Ordered paramodulation works by applying the same paramodulation rules, but in a more restricted way. It does so by defining an arbitrary fixed ordering on the set of constants $\mathcal{F}$, and adding various ordering constraints on the paramodulation rules. For the case of ground clauses with only constant symbols, the ordering constraints limit the superposition rules to cases where l is greater or equal to all symbols in C_1, and s is greater or equal to all symbols in C_2.

We define an arbitrary fixed ordering $>_v$ on the constant symbols set $\mathcal{F}$, such that v is the highest symbol under $>_v$. Now, since $SR(\varphi) \implies C$, and since ordered paramodulation is refutationally complete, there exists a series of inferences of ordered paramodulation rules with $>_v$ that start with the formula $SR(\varphi) \wedge \neg C$ and end with the empty clause (implying contradiction). Let $C_1, C_2, \ldots, C_n$ be a minimal such series of inferences. For every $1 \leq i \leq n$, C_i is inferred by an ordered paramodulation rule on one or two clauses from the set $SR(\varphi) \cup \{C_j \mid 0 \leq j < i\}$, and C_n is the empty clause. We claim that none of the clauses C_i contain the symbol v, as well as the clauses that were used to infer them. Thus, the same series of interferences proves that $\mathrm{remSym}(SR(\varphi), v) \wedge \neg C$ is unsatisfiable. Proving this, we will get the desired $\mathrm{remSym}(SR(\varphi), v) \implies C$.

Therefore, it remains to show that indeed v does not appear in any of the C_i's or the clauses that were used to infer them. First, we note that since v does not appear in C, then it is also not contained in the clauses comprising $\neg C$. We also know that $SR(\varphi)$ is saturated with respect to the paramodulation rules. Since n is minimal, C_1 must be a consequence of at least one clause from $\neg C$. Because v is the highest symbol on $>_v$, any clause containing v could not be applied together with a clause from $\neg C$, as v does not appear in $\neg C$. Thus, the clause(s) that were used to infer C_1 could not have contain the symbol v, and thus v is not contained in C_1. The exact same arguments, prove that v does not appear in C_2, C_3, etc. This completes the proof.