Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-024-09955-5

Common equivalence and size of forgetting from Horn formulae

Paolo Liberatore[1]

Accepted: 2 August 2024
© The Author(s) 2024

Abstract
Forgetting variables from a propositional formula may increase its size. Introducing new variables is a way to shorten it. Both operations can be expressed in terms of common equivalence, a weakened version of equivalence. In turn, common equivalence can be expressed in terms of forgetting. An algorithm for forgetting and checking common equivalence in polynomial space is given for the Horn case; it is polynomial-time for the subclass of single-head formulae. Minimizing after forgetting is polynomial-time if the formula is also acyclic and variables cannot be introduced, NP-hard when they can.

Keywords Logical forgetting · Knowledge representation · Logical minimization · Computational complexity

Mathematics Subject Classification (2010) 03B05 · 03B42 · 03D15

1 Introduction

Logical forgetting is removing conditions or objects from consideration [28, 34]. Also called variable elimination in the context of automated reasoning in propositional logic, it is done to work with bounded memory [34], simplify reasoning [29, 37, 89], clarify the relationship between variables [28], formalize the limited knowledge of agents [38, 75], ensure privacy [49], merge information coming from different sources [93], restore consistency [63].

The first four aims are missed if the result is too large. The space needed to store information increases instead of reducing. Reasoning from larger knowledge bases is likely harder rather than easier. The relationships between variables are probably obfuscated by an increase in size. A limit in knowledge storage ability is never enforced by enlarging a formula. Regardless of the aim, a large formula poses problems of storage, reasoning and ease of interpretation.

That forgetting increases size is counterintuitive since forgetting is removing conditions or objects from consideration. Less information should take less memory. Yet, less information may be more complicated to express. This is known to be the case in various logics [29, 35, 44, 48, 59, 65]. To complicate the matter, what results from forgetting may be large or small depending on the original formula; and may be equivalent to small formulae or not. For example, the classical syntactic definition of forget in propositional logic always doubles the size of the formula for each forgotten variable, but size may often be reduced [65].

✉ Paolo Liberatore
liberato@diag.uniroma1.it
http://www.dis.uniroma1.it/~liberato

1 DIAG, Sapienza University of Rome, Sapienza University of Rome, Via Ariosto 25, 00185 Rome, Italy

Besides forgetting, this is the classical problem of logic minimization [23, 24, 84]. It originates from electronic circuit synthesis: given a Boolean function, design a circuit that realizes it. The simpler the circuit, the better. Various solutions have been developed like the Karnaugh maps [57], the Quine-McCluskey method [70] and the Espresso algorithm [76]. The restriction to propositional definite Horn clauses is the problem of functional dependency minimization [7]. Various algorithms have been developed [39, 44, 51, 54, 90], and the problem proved to have only exponentially large solutions for some formulae [44, 51].

In spite of minimization, the result of forgetting may still be exponentially large, or just too large for the intended application. What to do in such cases? A way to make a formula smaller is to introduce new variables [9, 14, 18, 22, 71]. As an example, Mengel and Wallon [71] wrote: “It is folklore that adding auxiliary variables can decrease the size of an encoding: for example, the parity function has no subexponential CNF-representations but there is an easy linear size encoding using auxiliary variables”. Bubeck and Kleine Büning [14] wrote “Using auxiliary variables to introduce definitions is a popular and powerful technique in knowledge representation which can lead to shorter and more natural encodings”.

This may look like a vicious circle: variables are first removed, then added back. It is not. Variables that are unneeded or unwanted are forgotten; if the result is too big, other variables are introduced to make it smaller. A concrete example is a formula of 1000 variables, where only 20 are relevant to a certain aim and forgetting the remaining ones produces a formula of size 500000. Adding 14 other variables takes it down to size 200. The key is “other”: these are other variables. The 980 original variables are removed because they have to. They are not relevant to a certain context, they cause inconsistencies, they have to be hidden for legal reasons. Since the result is too large, 14 other variables are introduced to the sole aim of reducing size. Adding elements when forgetting others had been considered in the very context of forgetting, on abstract argumentation frameworks [5], answer set programming [36] and description logics [61].

While introducing variables is mainly motivated by reducing size, it may give a side benefit. If a new variable condenses a formula, it raises the question: does this happen by chance? The variable has no meaning by itself, it just reduces the size of the formula by pure luck. It so happens. But simpler explanations are usually considered better than complicated ones. They may be shorter because the added variable is a real fact, not just a technical trick to reduce size. It was missing from the original formula because the fact was hidden, not directly observable. Reducing size had the indirect benefit of uncovering it. Formal logic cannot tell its meaning, but tells that it may exist, and tells how it is related to the other facts.

Forgetting and introducing variables have something in common: they both change the alphabet of the formula while retaining some of its consequences. This is formalized by restricted equivalence [45], the equality of the consequences on a given subset of variables. When this subset comprises all variables, restricted equivalence is regular equivalence. When some variables are missing, only the consequences that do not contain those variables matter.

Forgetting is removing variables while retaining the consequences on the others; the result is equivalent to the original when restricting to the other variables.
Introducing is adding variables while retaining the consequences on the original variables; the result is equivalent to the original when restricting to the original variables.
Forgetting and introducing is removing and adding variables while retaining the consequences on the original variables not to be forgotten; the result is equivalent to the original when restricting to these variables.

Forgetting, introducing, forgetting and then introducing variables are all formalized by restricted equivalence. A subcase of it, actually: the restriction is on the variables that are shared between the input and the output formula. This is common equivalence: equality of the consequences on the common variables. Apart from the slight simplification of not explicitly requiring a set of variables, common equivalence forbids reintroducing a removed variable with a different meaning, which restricted equivalence allows.

Forget can be expressed in terms of restricted and common equivalence, but also the other way around. Both forms of equivalence amount to forget the variables that do not matter and then check regular equivalence. Theoretically, this can always be done. Computationally, it may not: if the result of forgetting is exponentially large, computing restricted or common equivalence this way requires exponential space while both problems can be solved in polynomial space. Size after forgetting matters again.

The exponentiality of forgetting is not just a possibility, nor it is due to a specific method of forgetting. For certain formulae and variables to forget, it is a certainty: the result of forgetting cannot be represented in polynomial space [44, 51]. While reducing its size is important, it is not always possible. In such cases, a forgetting algorithm takes exponential time just because its output is exponentially large. The time needed to just output it is exponential. Yet, the required memory space may not. An algorithm for the propositional Horn case is shown that runs in polynomial space even when it produces an exponential output, contrary to previous algorithms such as RBR on functional dependencies [51] and previous resolution-based elimination algorithms [28, 30, 91].

This algorithm unearths a polynomial Horn subclass: it runs in polynomial time when each variable is at most the head of a single clause.

Polynomial running time means polynomially-sized output. Fast enough, but not always small enough. A polynomial output may be quadratic. But even if it is only twice the size of the input, it is still a size increase. Forgetting fails at shortening the formula. But this is again the output of a specific algorithm. Equivalent but smaller formulae may exist. Surprisingly, it depends on whether they are required to be single-head or not. Either way, the minimal formula can be found in polynomial time if the formula is acyclic [53]. Otherwise, a sufficient condition directs the search for the clauses of the minimal formula.

Adding new variables allows reducing size. An algorithm for Horn formulae is given. It is polynomial but unable to always find the minimal formula. Not a drawback of this specific algorithm, however: the problem is NP-complete. It is NP-hard even in the single-head acyclic case. In the same conditions, the problem is polynomial without new variables. The reason is that the new variables may shorten the formula in many ways. Exploring them takes time.

Three implementations of the forgetting algorithm have been developed. They are all based on the same algorithm, but they differ in how they realize non-determinism. The first is correct only in the single-head case, which does not require non-determinism. The second employs sets to represent the possible outcomes of a non-deterministic choice; it is always correct but may take exponential space. The third exploits multiple processes; it is always correct and works in polynomial space. Checking common equivalence and minimizing with new variables are implemented as well.

A summary of the contributions of this article follows; it also outlines the organization of what follows. First, variable forgetting, introducing and forgetting followed by introducing are proved translatable into common equivalence, a specific case of restricted equivalence, which is also shown to be translatable into forgetting. Some results about common equivalence are proved, along with its Π_2^p-completeness in the general case and coNP-completeness in the Horn case. All of this is Section 3. Second, an algorithm for forgetting and checking common equivalence in the Horn case is presented. Contrary to previous methods for forgetting, it only requires a polynomial amount of working memory. It is first defined and proved correct in the definite Horn case in Section 4 and then extended to the general Horn case in Section 5. Third, a subclass of Horn formulae that makes the algorithm polynomial in time, in addition of working memory, is identified. The problem of minimizing such formulae after forgetting is analyzed in Section 6. Fourth, an algorithm for reducing size when new variables can be introduced is presented. The problem is shown NP-hard even in the simplest case considered in this article. This is the content of Section 7. The algorithms defined in this article are implemented, as summarized in Section 8. Proofs are in Appendix A, provided as supplementary material available online.

2 Preliminaries

Unless stated otherwise, logical formulae in this article are in Conjunctive Normal Form (CNF): they are sets of clauses, each clause being a disjunction of literal, where a literal is a propositional variable or its negation. Such a set is equivalent to the conjunction of its elements. Writing formulae as sets allows to compare them by containment: $A \subseteq B$ means that all clauses in A are also in B. A clause that contains all literals of another is a superclause of it. Clauses are assumed not to be tautologies. This condition is sometimes stated explicitly when important.

Some results are about Horn and definite Horn formulae. A clause is Horn if it contains at most one positive (not negated) literal. A formula is Horn if it comprises Horn clauses only. A clause is definite Horn if it contains exactly one positive literal. A formula is definite Horn if it comprises definite Horn clauses only. Horn and definite Horn formulae are also sets; therefore, they can be compared by set containment. Horn clauses are written as rules like $abc \rightarrow d$ instead of $\neg a \vee \neg b \vee \neg c \vee d$. Since tautologies are disallowed, writing $P \rightarrow x$ implicitly presumes $x \notin P$.

Two formulae are equisatisfiable if and only if either they are both satisfiable or they both are not.

The size of a formula is the number of its literal occurrences. Equivalently, it is the sum of the size of its clauses, where the size of a clause is the number of literals it contains.

Forgetting is removing one or more variables from consideration, in propositional logic. It can be defined in many ways, such as turning a formula F into $F[\mathsf{true}/x] \vee F[\mathsf{false}/x]$. The term is also applied to many other fields such as answer set programming [48], description logics [99], first-order logic [68, 101], and also refers to disregarding objects [28] or complex conditions [43]. When used to establish the satisfiability of a propositional formula, it is also called variable elimination [31, 80]; this is one of the many uses of forgetting, and has stricter running time requirements than most of the others mentioned at the beginning of the article; size bounds are guaranteed by them. In first-order, modal and description logics it is also identified by the concept of uniform interpolation [10]. The result of forgetting may or may not be expressed in the same language as the original formula [40, 50, 99]; it always does in propositional logics.

Forgetting may be defined in different ways depending on which properties it has to achieve [47]. The variants in propositional logics are literal forgetting [62], forgetting with fixed variables [72], forgetting a subformula [41], and forgetting while maintaining the abductive explanations [67]. Apart from these, forgetting some variables from a formula is another formula that entails exactly the same formulae on the other variables [62]. Formally, F' expresses forgetting the variables X from F if and only if $F' \models C$ is equivalent to $F \models C$ whenever C is a formula that does not contain any variable in X.

3 Common equivalence

Given that size is important to many applications of forgetting, the question is when forgetting decreases or increases size. A simple way to forget a variable X is by disjoining two copies of the formula, one for $x = \mathsf{true}$ and one for $x = \mathsf{false}$ [11]. The result is always double the size of the original. For example, forgetting X from F produces F':

$$\begin{aligned} F &= (x \vee y) \wedge (\neg x \vee \neg y) \wedge (a \vee b \vee c \vee d) \\ F' &= ((\mathsf{true} \vee y) \wedge (\neg \mathsf{true} \vee \neg y) \wedge (a \vee b \vee c \vee d)) \vee \\ &\quad ((\mathsf{false} \vee y) \wedge (\neg \mathsf{false} \vee \neg y) \wedge (a \vee b \vee c \vee d)) \end{aligned}$$

The resulting formula is larger than the original. Yet, it is equivalent to the shorter formula $F'' = a \vee b \vee c \vee d$. Reducing size while preserving equivalence has been extensively analyzed [20, 55] without forgetting.

With forgetting, the question is whether some formula of a given size or less expresses forgetting. In the example above, a formula of size 4 expresses forgetting X from F. No formula of size 3 or less does.

A common technique to reduce size is to introduce new variables to represent repeated subformulae [17, 22, 46, 71]. For example, forgetting X from $F = \{ab \rightarrow x, xc \rightarrow d, xc \rightarrow e, xc \rightarrow f\}$ is expressed by $F' = \{abc \rightarrow d, abc \rightarrow e, abc \rightarrow f\}$. This formula contains twelve literal occurrences, the same as F. It is minimal: it is equivalent to no smaller formula. Yet, it is shortened by introducing a new variable z to represent abc. The result $F'' = \{abc \rightarrow z, z \rightarrow d, z \rightarrow e, z \rightarrow f\}$ contains only ten literal occurrences.

The result of such an addition is not exactly equivalent to the original. The original formula does not mention z; therefore, its value is unaffected by the value of z. If a model satisfies the formula, it still does when changing the value of z. This is not the case after adding the new variable z. For example, the model that sets all variables to true satisfies the formula, but no longer does when changing the value of z to false, since $a = \mathsf{true}$, $b = \mathsf{true}$, $c = \mathsf{true}$ and $z = \mathsf{false}$ falsify the clause $abc \rightarrow z$. Having different value on this model, F and F'' are not equivalent.

Yet, F'' expresses the same information as F' apart from z, which is just a shorthand for $a \wedge b \wedge c$. They are equivalent when disregarding z. For example, they entail the same consequences that do not contain z. They are satisfied by the same partial models that do not evaluate z. Excluding variables from the comparison is restricted equivalence [45] or var-equivalence [62].

Definition 1 **(Restricted equivalence [45] or Var equivalence [62])** Two formulae A and B are restricted-equivalent or var-equivalent on the variables X if $A \models C$ holds if and only if $B \models C$ holds for every formula C over the alphabet X.

Restricted equivalence formalizes the addition of new variables to the aim of reducing size [9, 14]: the resulting formulae are equivalent to the original formula on the original variables. It aims at shrinking a formula A over variables X by producing a smaller formula B over variables $X \cup Y$ that is restricted-equivalent to A over the variables X.

This is forgetting in reverse: instead of forgetting Y from B to produce A, it adds Y to A to produce B. It can indeed be reformulated in terms of forgetting: given A over X, find for a formula B of the given size such that forgetting Y from B produces A.

In the other way around, restricted equivalence formalizes forgetting: the result of forgetting X from A is a formula B over the variables $Var(A) \backslash X$ that is restricted-equivalent to A over the variables $Var(A) \backslash X$.

In the way around the other way around: A and B are restricted-equivalent over X if forgetting $Var(A)\backslash X$ from A is equivalent to forgetting $Var(B)\backslash X$ from B [62].

Restricted equivalence formalizes variable forgetting, variable introduction and variable forgetting followed by variable introduction. All three forms of change are restricted equivalence over the variables that are not forgotten and not introduced.

The problem with restricted equivalence is that it is too powerful. It allows a forgotten variable to be reintroduced with a different meaning. Forgetting X from $\{\neg x, abc \rightarrow d, abc \rightarrow e, abc \rightarrow f\}$ is expressed by $\{abc \rightarrow d, abc \rightarrow e, abc \rightarrow f\}$, which can be shortened by introducing a variable: $\{abc \rightarrow x, x \rightarrow d, x \rightarrow e, x \rightarrow f\}$. The variables that are neither forgotten nor introduced are a, b, c, d, e, f; the two formulae are restricted equivalent over them. Restricted equivalence is blind to X being a fact that is false in the original formula and a shorthand for abc in the final. Such cases are avoided by comparing formulae on the common variables instead of an arbitrary set of variables.

Definition 2 Two formulae A and B are common-equivalent, denoted $A \equiv\equiv B$, if $A \models C$ holds if and only if $B \models C$ holds for every formula C such that $Var(C) \subseteq Var(A) \cap Var(B)$.

Common equivalence is: same consequences on the common alphabet.

Forgetting variables X from a formula A results in a formula B such that $Var(B) = Var(A)\backslash X$ and $B \equiv\equiv A$. Adding new variables to a formula A produces a formula B such that $Var(A) \subseteq Var(B)$ and $B \equiv\equiv A$. Forgetting followed by adding is $Var(A)\backslash X \subseteq Var(B)$ and $B \equiv\equiv A$. All three operations are defined in terms of common equivalence and some simple condition over the variables.

A caveat on variable forgetting and introducing defined in terms of common equivalence is that the formulae they produce are not always strictly minimal. For example, forgetting X from a formula built over the alphabet $\{x, y, z\}$ using this definition always produces a formula that contains y and z; yet, a formula that is minimal among the ones that contain y and z may be equivalent to a smaller one that only contains z. Forcing the use of y is necessary to employ common equivalence, but may artificially increase size. However, this presence is easily accomplished by subformulae such as $y \vee \neg y$. They only increase size linearly in the number of variables.

Common equivalence is in line with the view of forgetting as language reduction [28]. It is not syntactical, but based on the consequences on the common alphabet. Viewing the consequences of a formula as an explicit representation of what the formula tells, $\equiv\equiv$ compares two formulae on what they say about the things they both talk about.

Common equivalence can be defined in alternative ways based on consistency rather than entailment.

Theorem 1 *The condition $A \equiv\equiv B$ is equivalent to $A \cup S$ and $B \cup S$ being equisatisfiable for every set of literals S over $Var(A) \cap Var(B)$.*

This condition can be further restricted: instead of checking consistency over all sets of literals over the common alphabet, the ones that contain all common variables are enough. In other words, the models over the common alphabet are partial models of both formulae or none, if the formulae are common equivalent.

Theorem 2 *The condition $A \equiv\equiv B$ is equivalent to $A \cup S$ and $B \cup S$ being equisatisfiable for every set of literals S that contains exactly all variables that are common to A and B.*

A situation of particular interest is when one of the two formulae contains only some variables of the other. It is the case when forgetting some variables. It is also the case when introducing new variables. It is not when first forgetting and then introducing variables.

Lemma 1 *If $A \equiv\equiv B$ and $Var(B) \subseteq Var(A)$, then $A \models B$.*

The converse of this lemma does not hold. For example, $B = \{x\}$ does not entail $A = \{x, y\}$ in spite of their common equivalence. Contrary to regular equivalence, common equivalence is not the same as mutual implication. It only contains implication in one direction, and only when the variables of a formula are all in the other.

This particular case allows for a slight simplification of the definition.

Theorem 3 *If $Var(B) \subseteq Var(A)$, then $A \equiv\equiv B$ holds if and only if $A \models B$ and the satisfiability of $B \cup S$ implies that of $A \cup S$ for every consistent set of literals S that contains exactly all variables in $Var(B)$.*

While equivalence is transitive, common equivalence is not. An example where both $A \equiv\equiv B$ and $B \equiv\equiv C$ hold and $A \equiv\equiv C$ does not is $A = x \wedge y$, $B = x$ and $C = x \wedge \neg y$. Transitivity does not hold because A and C share the variable y while imposing different values on it, violating common equivalence; this variable is not in B, and is therefore not shared between B and A and between B and C; the different values of y in A and C do not prevent their common equivalence to B.

This cannot happen if all variables of A and C are shared with B. This is a general result: transitivity holds in these cases.

Lemma 2 *If $Var(A) \cap Var(C) \subseteq Var(B)$ then $A \equiv\equiv B$ and $B \equiv\equiv C$ imply $A \equiv\equiv C$.*

A following result requires a formula that is not in CNF. Every formula can be turned into CNF, but this may exponentially increase its size. This can be avoided by adding new variables. Does such an addition affect common equivalence? The following lemma answers: it does not. If a formula is the result of adding new variables to another, all properties related to common equivalence are preserved.

Lemma 3 *If $A \equiv\equiv A'$, $Var(A) \subseteq Var(A')$ and $(Var(A') \backslash Var(A)) \cap Var(B) = \emptyset$, then $A \equiv\equiv B$ if and only if $A' \equiv\equiv B$.*

A formula that entails a literal is not equivalent to the formula with the literal replaced by true. Yet, they only differ on that literal: the first formula entails it, the second does not mention it; their consequences are otherwise the same. This is exactly what common equivalence formalizes. It is able to express that adding a literal to a formula and setting its value in the formula are essentially the same. Regular equivalence does not.

Lemma 4 *For every formula F and variable x, the common equivalence $F \cup \{\neg x\} \equiv\equiv F[\bot/x]$ holds.*

Regular equivalence is not only transitive but also monotonic: if two formulae are equivalent, they remain equivalent after conjoining both with the same formula. The same holds for common equivalence when conjoining with a formula on the shared variables.

Lemma 5 *If $A \equiv\equiv B$ then $A \cup C \equiv\equiv B \cup C$ if $Var(C) \subseteq Var(A) \cap Var(B)$.*

Flogel et al. [45] proved restricted equivalence coNP-complete if the two formulae are Horn. Lang et al. [62] proved that var-equivalence is Π_2^p-complete in the general case. Membership to these classes extends to common equivalence, which is restricted equivalence or var equivalence on the variables that are shared among the two formulae.

Hardness in the general case is a consequence of the following theorem. It holds even when the alphabet of a formula is a subset of the other. This is the case with variables forgetting alone and with variable introduction alone.

Theorem 4 *The problem of establishing whether $A \equiv\equiv B$ is Π_2^p-complete. Hardness holds even if $Var(B) \subseteq Var(A)$.*

Being a restriction of restricted equivalence, common equivalence is in coNP as well in the Horn case. In spite of being a restriction, it is also still coNP-hard. It remains coNP-hard even when a formula is a subset of the other, which implies that its variables are a subset of those of the other. This is the case with variable forgetting: the result is a common-equivalent formula on a subset of the variables. Therefore, establishing whether a formula is a valid way of forgetting variables from another is coNP-hard in the Horn case. This is the same problem as checking whether a formula is a valid way of introducing variables in the other since forgetting is the opposite of introducing. Both problems are therefore coNP-hard. Hardness holds even if B is a subset of A.

Theorem 5 *If A and B are Horn, establishing whether $A \equiv\equiv B$ is coNP-complete. Hardness holds even if $B \subseteq A$.*

The problem remains coNP-hard even if the formulae are definite Horn. Also in this case, hardness holds even if B is a subset of the Horn clauses of A.

Theorem 6 *Checking $A \equiv\equiv B$ is coNP-hard if A and B are definite Horn and $B \subseteq A$.*

Complexity of common equivalence is established when the problem is formalized as a decision: are two Horn formulae common-equivalent? The two formulae are both assumed Horn, excluding the case when forgetting turns a Horn formula into a non-Horn formula. Or when variable introduction makes a Horn formula non-Horn.

These cases are both possible, in the sense that a specific procedure for forgetting may generate non-Horn clauses. As an example, the non-Horn formula $F' = \{x, y, x \vee y\}$ is a way of forgetting z from the Horn formula $F = \{x, y, z\}$. Not a natural way to do so, but still valid. Yet, a better way of forgetting is $F'' = \{x, y\}$, which is Horn. Incidentally, F'' is also the result of minimizing F'.

This is not specific to the example: Wang [91, Corollary 7 and 8] proved that forgetting variables from a Horn formula is equivalent to a Horn formula. In terms of common equivalence, if A is Horn, $A \equiv\equiv B$ and $Var(B) \subseteq Var(A)$, then B is equivalent to a Horn formula.

This is the case when the variables of B are a subset of the variables of A, not the opposite. For example, the Horn formula $A = \{x\}$ is common-equivalent to $B = \{x, y \vee z\}$, which is not Horn and is not equivalent to any Horn formula. In practice: forgetting variables may turn a non-Horn formula into a Horn formula. Also: introducing variables may generate a non-Horn formula.

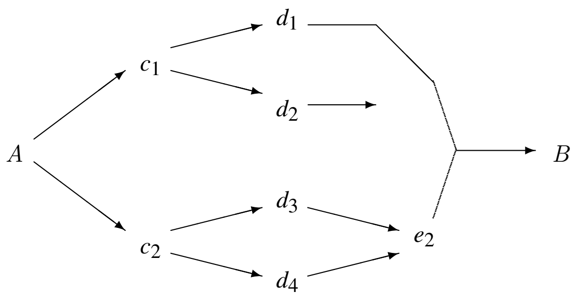
Preservation of the Horn restriction does not extend to size. Fischer et al. [44] prove that every set of functional dependencies that is equivalent over the variables $X = \{a_i, b_i | 1 \leq i \leq n\} \cup \{d\}$ to $F = \{a_i \rightarrow c_i \mid 1 \leq i \leq n\} \cup \{b_i \rightarrow c_i \mid 1 \leq i \leq n\} \cup \{c_1 \dots c_n \rightarrow d\}$ is exponential in n. Since functional dependencies are equivalent to definite Horn clauses, no formula of polynomial size over X is common equivalent to F.

4 Algorithm for definite Horn formulae

Forgetting variables can be done by resolving each out [28, 30, 51, 91]: every pair of clauses containing opposite occurrences of the same variables are resolved and removed. On Horn formulae, this is equivalent to unfolding X [89]: replace every negative occurrence of X with the negative literals of a clause where X occurs positive. Forgetting a variable at time is also common in Answer Set Programming: most algorithms forget a single variable, which implies that forgetting multiple ones is done one variable at time [8, 35, 58, 97]. Incidentally, forgetting in Answer Set Programming is significantly different from forgetting in propositional logic, as discussed in Section 10.

Unfolding a variable at time may generate large intermediate formulae even if the final result is small. An example is forgetting all variables but A and B from the following formula.

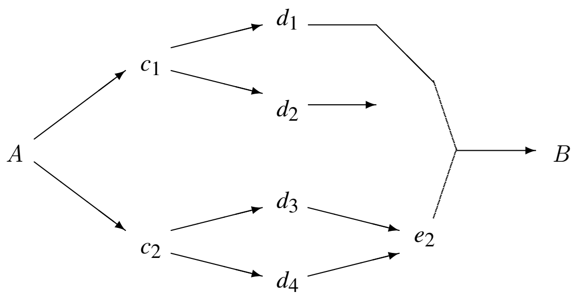
$$\begin{aligned} F = \{&a \rightarrow c_1, a \rightarrow c_2, c_1 \rightarrow d_1, c_1 \rightarrow d_2, c_2 \rightarrow d_3, c_2 \rightarrow d_4, \\ &d_1 \rightarrow e_1, d_2 \rightarrow e_1, d_3 \rightarrow e_2, d_4 \rightarrow e_2, e_1e_2 \rightarrow b\} \end{aligned}$$



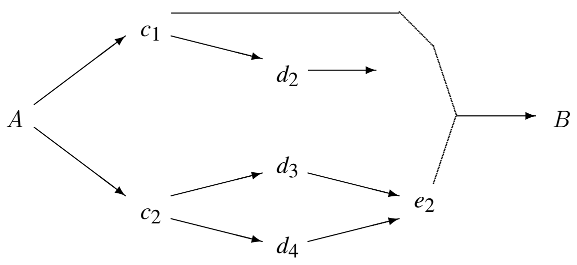
Unfolding e_1 turns $e_1e_2 \rightarrow b$ into $d_1e_2 \rightarrow b$ and $d_2e_2 \rightarrow b$. Unfolding e_2 turns the first clause into $d_1d_3 \rightarrow b$ and $d_1d_4 \rightarrow b$ and the second into $d_2d_3 \rightarrow b$ and $d_2d_4 \rightarrow b$. Unfolding the variables d_i turns all of them into $a \rightarrow b$.

Forgetting two variables generates four intemediate clauses for producing one. Increasing the number of the variables e_i from 2 to n and d_i from 4 to $2n$ increases the intermediate clauses to 2^n, while the output is still one.

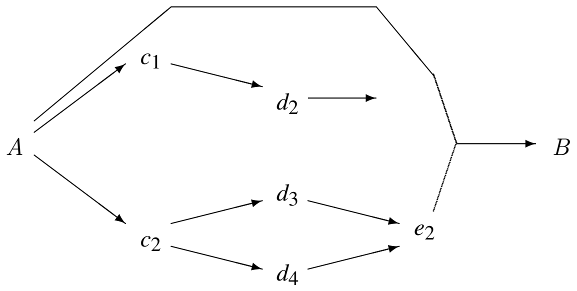
This can be avoided by doing one substitution at time in one clause at time. The other clauses and the other substitutions are left waiting until the current clause is over. Starting with $e_1e_2 \rightarrow b$, its body variable e_1 can be replaced by d_1 and by d_2, but only the first substitution is done; the other is left waiting. The result is the single clause $d_1e_2 \rightarrow b$.



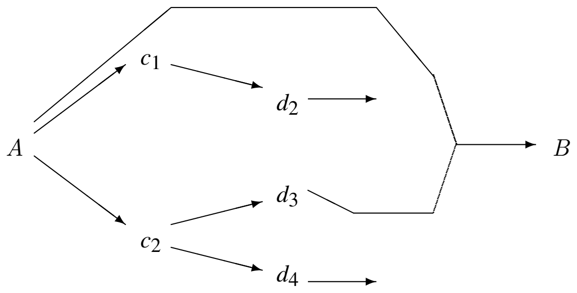
The first variable d_1 of $d_1e_2 \rightarrow b$ can be replaced by c_1 only, producing $c_1e_2 \rightarrow b$.



The first variable c_1 of $c_1e_2 \to b$ can only be replaced by A, producing $ae_2 \to b$.



Since A is not a variable to forget, it is not replaced. The only variable to forget in $ae_2 \to b$ is e_2. It can be replaced by d_3 and by d_4, but only the first is done; the other is left waiting. What results from replacing e_2 with d_3 is $ad_3 \to b$.



Replacing d_3 with c_2 in $ad_3 \to b$ produces $ac_2 \to b$, where replacing c_2 with A produces $a \to b$. This clause does not contain variables to forget. It is therefore output and not further processed.

One of the substitutions on hold is now restarted. The last was replacing e_2 with d_4 in $ae_2 \to b$. Its effect is $ad_4 \to b$. The variable d_4 is replaced with c_2, which is then replaced with A. The result is $a \to b$ again. The other replacement still waiting is replacing e_1 with d_2 in $e_1e_2 \to b$, which produces the same result.

Whenever a variable can be replaced in multiple ways, only one is done. The others are stopped. Contrary to unfolding a variable at time, a line of replacements is followed until no longer possible; only then the alternatives are considered. Replaced variables are not replaced again to avoid looping.

How much memory is required? The current clause is not larger than the number of variables, and this is linear space. Every variable is only replaced once, meaning that the replacements done on the current clause are linear. Each replacement may have alternatives:

every alternative is another clause to replace one variable in a clause. The number of alternatives is at most the number of clauses at each step. The memory required to store all of them is quadratic at most. Quadratic is polynomial.

As suggested by one of the reviewers of this article, a trivial way to forget in polynomial working space is to loop over all clauses over the variables to remember, collecting the ones entailed by the formula. It is the brute-force algorithm Gottlob [51] commented on: "The advantage of RBR over the brute-force approach to compute F^+ and projecting it on R is that the latter method is always exponential, since F^+ contains more than 2^n trivial dependencies, while RBR is exponential in bad cases only." The same comparison holds here: looping over all clauses is always exponential in time, replacing heads is so only in certain cases.

4.1 Summary

A summary of the proof of correctness is given here.

The first step is a basic property of entailment in Horn logic: Lemma 6 in Section 4.2 shows that if a formula entails a non-tautological Horn clause, it contains a clause with the same head and with the body entailed by the body of the entailed clause.

The core of the method is a non-deterministic procedure **body_replace**(F, R, D), presented and analyzed in Section 4.3. It recursively replaces some variables of R with their premises in F, where D are the variables already replaced. This procedure has two return values. The first is a possible result of replacing variables in R. The second is the set of variables that have been replaced. Once a variable is replaced with others, it is not replaced again but just deleted; this is essential to avoid looping, as otherwise the algorithm would replace variables in cycles of clauses forever. Calling **body_replace**$(F, P, \emptyset)$ with certain non-deterministic choices produces a piece of the result of forgetting. Namely, if P is the head of a clause $P \rightarrow x$ of F, the first return value P' is the body of a clause $P' \rightarrow x$ in the result of forgetting. The result of forgetting is the set of all clauses returned this way.

Most of the work is done by **body_replace**$(F, P, \emptyset)$. Section 4.4 presents a procedure **head_implicates**(F) that calls it on all bodies of the clauses of F. With certain non-deterministic choices, it returns the result of forgetting a set of variables from F.

This procedure is in turn called by **common_equivalent**(A, B) to generate all clauses of forgetting, checking whether each is entailed by the other formula. It only takes polynomial space, which is not obvious since the result of forgetting may be exponentially large. This is the content of Section 4.5.

Finally, **forget**(F, X) forgets variables X from formula F. This procedure is described in Section 4.6.

All these algorithms become deterministic if each variable is the head of at most one clause. Since non-determinism is what requires exponential time, this restriction makes forgetting and checking common equivalence polynomial-time. A formula satisfying this condition is called single-head. Section 4.7 proves it makes **common_equivalent**(A, B) run in polynomial time.

4.2 Set implies set

The base of the algorithm for forgetting is a lemma relating entailment and containment. It involves removing all clauses including a variable.

Definition 3 For every set of clauses F, the set F^x contains all clauses of F that contain neither x nor $\neg x$.

That no clause is tautologic is generally presumed throughout this article as stated in the preliminaries. It is however explicit in the lemma because it is important in the proof.

Lemma 6 *If F is a definite Horn formula, the following three conditions are equivalent, where $P' \rightarrow x$ is not a tautology ($x \notin P'$).*

1. $F \models P' \rightarrow x$;
2. $F^x \cup P' \models P$ *for some* $P \rightarrow x \in F$;
3. $F \cup P' \models P$ *for some* $P \rightarrow x \in F$.

This lemma proves that all clauses $P' \rightarrow x$ entailed by F are either themselves in F, or are consequences of another clause $P \rightarrow x \in F$ thanks to F^x implying $P' \rightarrow P$. Seen from a different angle, all derivations of x from P' use $P \rightarrow x \in F$ as the last step, and only clauses of F^x in the previous.

The condition $F^x \cup P' \models P$ can be rewritten as $F^x \models P' \rightarrow y$ for every $y \in P$. This allows applying the lemma again, to $P' \rightarrow y$: there exists $P'' \rightarrow y \in F$ such that $F^{xy} \models P' \rightarrow P''$.

This recursion is not infinite: it ends when $P' = P$; not only the lemma does not rule this case out, it is its base case. More generally, it does not forbid P and P' to intersect. Rather the opposite: at some point of recursion they must coincide. Indeed, F becomes F^x, then F^{xy} and so on; infinite recursion is impossible because the formula shrinks at every step; at the same time, it cannot become empty since the lemma implies that the formula contains at least a clause.

4.3 body_replace()

The recursive application of Lemma 6 allows for repeatedly replacing unwanted variables with others implying them. When forgetting y, every entailed clause $F \models P' \rightarrow x$ not containing y must survive. By Lemma 6, the entailment implies $F^x \models P' \rightarrow P$ and $P \rightarrow x \in F$. If $y \notin P$, clause $P \rightarrow x$ survives the removal of y. If $y \in P$, Lemma 6 kicks in: $F^x \models P' \rightarrow y$ requires some clauses $P'' \rightarrow y \in F$. Each can be combined with $P \rightarrow x$ to obtain $((P'' \cup P)\backslash\{y\}) \rightarrow x$, which does not contain y. This is equivalent to resolving the two combined clauses. It leaves $P' \rightarrow x$ entailed in spite of the removal of $P \rightarrow x$. The same procedure allows forgetting other variables at the same time.

```
## recursively replace some variables in $R$ with their preconditions in $F$
# input $F$: a formula
# input $R$: a set of variables, some of which are replaced
# input $D$: a set of variables to delete
# first output: $R$ with some variables replaced
# second output: the variables replaced in the process

variables, variables body_replace(formula $F$, variables $R$, variables $D$)

1. choose $R' \subseteq R\backslash D$
2. $S' = \emptyset$
3. $E' = \emptyset$
4. foreach $y \in R'$

    (a) if $y \in D \cup E'$ continue
    (b) choose $P \to y \in F$ else **fail**
    (c) $S, E = \textbf{body\_replace}(F^y, P, D \cup E')$
    (d) $S' = S' \cup S$
    (e) $E' = E' \cup E \cup \{y\}$

5. return $(R \backslash D \backslash R') \cup S', E'$
```

The procedure is non-deterministic. The points of branching are the choice of the subset R' of $R \backslash D$ and the choice of the clause $P \to y$ of F. A non-deterministic branch is created for each such subset and each such clause. The difference is that the first choice is always possible, the second may not. Even if $R \backslash D$ is empty, it still has the empty set as a subset. Instead, if F does not contain any clause $P \to y$, no clause can be chosen. The current non-deterministic branch of execution terminates without producing any result. This is the meaning of "else **fail**": it terminates the current branch of execution without any contribution to the output if the choice is impossible.

The choice of the initial values of R and D and the choice of R' is left open at this point because it simplifies some proofs. The intention is that R is initially the body of some clause $R \to x$ where X is a variable not to be forgotten, D is initially empty and R' contains exactly all variables of $R \backslash D$ to forget. When R' is empty, no recursive call is performed. In each call:

- R is the body of some clause $R \to x$ where X is a variable not to forget;
- **body_replace**(F, R, D) tries to replace the variables of R to forget with others that entail them and are not to forget;
- the first return value is the set of replacing variables;
- D contains the variables that have already been replaced, so they can be deleted;
- the second return value contains the variables that have been replaced in this call.

This procedure hinges around its second parameter R, a set of variables. The procedure replaces some elements of R with variables that entail them with F; the replacing variables are the first return value. In the base case, $y \in R$ is replaced by some P with $P \to y \in F$, but then some elements of P may be recursively replaced in the same way. This is how forgetting happens: if all variables to be forgotten are replaced by all sets of variables that entail them, they disappear from the formula while leaving all other consequences intact.

The last parameter D is empty in the first call, and changes at each recursive call. It is the set of variables already replaced. Every time a variable $y \in R$ is replaced, it is added to E', which is then passed to every subsequent recursive call and eventually returned as the second return value. This way, if y has already been replaced it is removed instead of replaced again; this is correct because its replacing variables are already in S', which is part of the first return value.

The two return values have the same meaning of R and D but after the replacement: $(R \backslash D \backslash R') \cup S'$ is R with some variables replaced; E' is the set of these replaced variables; to be precise, these are only the variables replaced in this call and it subcalls, not in some previous recursive call.

Two conditions prevent a recursive call: the presence of y in $D \cup E'$ (Step 4a) and the lack of a clause $P \to y$ in F (Step 4b). They differ radically. The first tells that y was already replaced, the second that y cannot be replaceed.

An example is a formula that contains $y \to x$, $y' \to x$ and $z \to y'$, where x is to be replaced in a recursive subcall where a previous recursive subcall already replaced y'.

$$\begin{tikzcd}
& y \arrow[dr] & \\
& & x \\
z \arrow[r] & y' \arrow[ur] &
\end{tikzcd}$$

Two variables can replace x: y and y'. Each corresponds to a different choice of the clause in Step 4b. The algorithm splits into two non-deterministic branches, one for y and one for y'.

The first non-deterministic branch executes a recursive subcall to replace y, but F contains no clause with head y. This non-deterministic branch fails because x cannot be replaced this way. It does not just go ahead with the other variables to replace. Since y cannot be replaced, x cannot be replaced this way. The other variables are irrelvant.

The other non-deterministic branch executes a recursive subcall to replace y'. Since y' was already replaced in a previous subcall, z was already returned by that subcall. Since the returned sets are accumulated in Step 4d and eventually returned, z is already guaranteed to be in the final output. Variable y' is just removed. Yet, the other variables to replace still need to be replaced. This non-deterministic branch cannot terminate.

In summary, a variable that is impossible to replace cause the non-deterministic branch to fail; a variable that was already replaced is just removed.

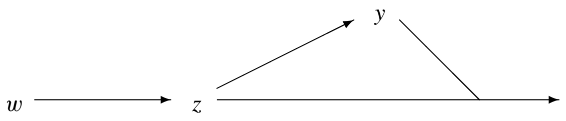
Termination is guaranteed by:

(1.) passing F^y to the recursive calls, and
(2.) maintaining the set of already-replaced variables in D.

The first termination mechanism (1) makes the formula passed to the subcalls smaller and smaller; when it is empty, the non-deterministic choice of a clause in the loop fails. The second termination mechanism (2) makes replacing y when $y \in D$ just a matter of removing y without a recursive call.

$$\begin{tikzcd}
z \arrow[r, bend left] & y \arrow[l, bend left] \arrow[r] & x
\end{tikzcd}$$

The first termination mechanism (1) breaks loops like in $\{y \rightarrow z, z \rightarrow y, y \rightarrow x\}$ when forgetting y and z: after replacing y by z, the clause $y \rightarrow z$ is removed from the formula passed to the subcall. This disallows replacing z by y, which would create an infinite chain of replacements.



The second termination mechanism (2) forbids multiple replacements, like in $\{w \rightarrow z, z \rightarrow y, yz \rightarrow x\}$ when forgetting y and z. The algorithm would otherwise replace z by w twice: after replacing y by z and z by w in $yz \rightarrow x$, the clause $yz \rightarrow x$ becomes $wz \rightarrow x$, and the procedure moves to replacing z; this is recognized as unnecessary because z has already been replaced (by w), so it is simply deleted.

While termination is guaranteed, success is not. The set R' may contain a variable y with no clause $P \rightarrow y$ in F. In such cases, y cannot be replaced by its preconditions in F. For example, when replacing $\{y, z\}$ in $F = \{w \rightarrow y, yz \rightarrow x\}$, the first variable y can be replaced by w, but the second variable z cannot be replaced by anything, since no clause has z as its head. Running **body_replace**$(F, \{y, z\}, \emptyset)$ fails if R' is chosen to contain z. This is correct because no subset of the other variables $\{x, w\}$ entail z with F. Failure indicates that no such replacement is possible.

The first step of proving that the procedure always terminates shows when no recursive subcall is done. This is the base case of recursion. The conditions preventing a subcall in Step 4a and Step 4b are false at the first iteration if R' is not empty and the algorithm does not fail. The following lemma shows that this is a necessary and sufficient condition.

Lemma 7 *A successful call (one that does not execute "***fail***") to* **body_replace**() *does not perform any recursive subcall if and only if* $R' = \emptyset$.

The main component of the proofs about **body_replace**() are its two invariants. The first is recursive, a relation between parameters and return values. The second is iterative, a relation between the local variables at each iteration of the loop. All properties of the procedure are consequences of the first, which requires the second to be proved. More precisely, each proves the other. This is why they are both proved in the same lemma.

Lemma 8 *The following two invariants hold when running* **body_replace**(F, R, D)*:*

recursive invariant: if it returns S, E, then $F \cup S \cup D \models R \cup E$;

loop invariant: at the beginning and end of each iteration of its loop (Step 4), it holds $F \cup S' \cup D \models E'$.

The aim of **body_replace**(F, R, D) is to replace some variables of R with others implying them with F. The recursive invariant after $S, E =$ **body_replace**(F, R, D) is $F \cup S \cup D \models R \cup E$. The first return value entails the variables to be replaced, when $D = \emptyset$ as it should be in the first recursive call. Otherwise, D is a set of variables that have already been replaced; their replacements are in the set S for some call somewhere in the recursive tree. Since all first return values are accumulated, they are returned by the first call.

All of this happens if the recursive call returns.

Forgetting maintains all consequences on the non-forgotten variables. Let $P' \rightarrow x$ be such a consequence: $F \models P' \rightarrow x$. This condition is the same as $F^x \models P' \rightarrow P$ and $P \rightarrow x$ by Lemma 6. While X is not one of the variables to forget by assumption, P may contain some. If it does, they can be replaced by their preconditions, as **body_replace**$(F, P, \emptyset)$ does. But variables may have more than one set of possible preconditions, and some may not be useful to ensure the entailment of $P' \rightarrow x$. Only the ones entailed by P' are. The others are not: if a replacing variable is not entailed by P', the result of replacement cannot be used to entail $P' \rightarrow x$, still because of Lemma 6. The following lemma confirms that the right choice of preconditions is always possible.

Lemma 9 *If* $F \cup P' \models P$*, some non-deterministic choices of clauses in* **body_replace**$(F, P, \emptyset)$ *and its subcalls ensure their successful termination and the validity of the following invariants for every recursive call* $S, E =$ **body_replace**(F, R, D) *and every possible choices of* R' *that do not include a variable in* P'*:*

$$F \cup P' \models R$$
$$F \cup P' \models S$$

Since the invariants hold for all calls, they hold for the first: $F \cup P'$ implies the first return value of **body_replace**$(F, P, \emptyset)$ for appropriate non-deterministic choices. In the other way around, the non-deterministic choices can be taken so to realize this implication.

The second requirement of forgetting is that all variables to be forgotten are removed from the formula. Calling **body_replace**$(F, P, \emptyset)$ replaces the variables in R' from P. Forgetting is obtained by choosing R' as the variables to be forgotten in P in the first call. In an arbitrary call **body_replace**(F, R, D), the variables in D have already been replaced, so they are not to be replaced again. Therefore, R' comprises the variables to be forgotten of $R \backslash D$.

The following lemmas concern **body_replace**(F, R, D) when R' is always $R \backslash D \backslash V$ for some set of variables V. Forgetting is achieved if V is the set of variables to be retained.

Lemma 10 *If R' is always chosen equal to $R \backslash D \backslash V$ for a given set of variables V, the first return value of* **body_replace**$(F, P, \emptyset)$ *and every recursive subcall is a subset of V, if the call returns.*

Termination is not guaranteed by this lemma; for example, a failure is generated by **body_replace**$(F, \{y\}, \emptyset)$ when $F = \{y \rightarrow x\}$ with $V = \{x\}$, since $R' = R \backslash \emptyset \backslash \{x\} = \{y\}$, but no clause of F has y in the head. The claim of the lemma only concerns the case of termination, as the words "if this call returns" specify.

The following lemma shows that Lemma 9 holds even if R' is always chosen to be $R \backslash D \backslash V$. This is not obvious because that lemma only states that its claim holds for some non-deterministic choices.

Lemma 11 *For some non-deterministic choices of clauses the call $S, E =$* **body_replace**$(F, P, \emptyset)$ *succeds and the first return value satisfies $F \cup P' \models S$, provided that $F \cup P' \models P$, $P' \subseteq V$ and the non-deterministic choices of variables are always $R' = R \backslash D \backslash V$.*

The conclusion of this series of lemmas is that $S, E =$ **body_replace**$(F, R, \emptyset)$ returns a set $S \subseteq V$ such that $F \models S \rightarrow R$ and $F \models R \rightarrow P$ when it does certain non-deterministic choices.

4.4 head_implicates()

Forgetting requires all consequences $P' \rightarrow x$ on the variables not to forget to be retained while all variables to forget disappear. By Lemma 6, $F \models P' \rightarrow x$ is the same as $F^x \models P' \rightarrow P$ and $P \rightarrow x$. A way to entail $P' \rightarrow x$ from another formula G is by ensuring $G \models P' \rightarrow S$ and $S \rightarrow x \in G$. The lemmas in the previous section say how to obtain such a formula G: by running $S, E =$ **body_replace**$(F, P', \emptyset)$, which ensures $S \subseteq V$, $F \models P' \rightarrow S$ and $F \models S \rightarrow P$. Since $S \rightarrow x$ comprises the common variables, it is a valid choice for G. It allows entailing $P' \rightarrow x$ if $G \models P' \rightarrow S$. The latter condition is achieved by ensuring $G \models P' \rightarrow s$ for each $s \in S$ in a similar way. In other words, if every $P \rightarrow x \in F$ is turned into $S \rightarrow x$ for all S that are the first return value of **body_replace**$(F, P', \emptyset)$, all common consequences are retained while all variables to forget are removed.

This is what the following procedure does.

```
## replace part of a body of $F$ with their preconditions
# input $F$: a formula
# output: a clause of $F$ with part of its body replaced
clause **head_implicates**(formula $F$)

 1. choose $x \in Var(F)$ else return $\emptyset$
 2. choose $P \rightarrow x \in F$ else return $\emptyset$
```

```
3. $S, E =$ body_replace($F^x, P, \emptyset$)
4. return $\{S \rightarrow x\}$
```

In order to derive all common consequences $P' \rightarrow x$ from the resulting formula, replacing every $P \rightarrow x$ with $S \rightarrow x$ is not enough. The same is required for $P' \rightarrow s$ for every $s \in S$. Common equivalence is achieved only when the replacement is done to all clauses, not only the ones with head x.

A first required lemma is that no tautology is returned. It looks obvious, and could also be shown as a consequence of a condition on **body_replace**(F, R, D) always returning a subset of $Var(F) \cup R$, but that would require a recursive proof. Instead, it is obvious when $D = \emptyset$.

Lemma 12 *No run of* **head_implicates**(F) *produces a tautological clause.*

The following lemma proves that head_implicates(F) produces the required clause $S \rightarrow x$. It may also produce other clauses, depending on the non-deterministic choices.

Lemma 13 *If $F \models P' \rightarrow x$, $P' \subseteq V$ and $x \notin P'$ then* **head_implicates**(F) *outputs a clause $S \rightarrow x$ such that $F^x \cup P' \models S$, provided that* **body_replace**$()$ *always choses R' as $R \backslash D \backslash V$ for some fixed set of variables V.*

In the following sections V always comprises the variables to be retained. Yet, the following properties of **head_implicates**(F) are not limited to any choice of V.

The reason why the above is a lemma and not a theorem is that the conclusion that all common consequences are maintained requires its repeated application: not only $P \rightarrow x$ is replaced by $S \rightarrow x$, but also $P'' \rightarrow s$ is similarly replaced for every $s \in S \backslash V$.

Theorem 7 *With the non-deterministic choices $x \in V$ and $R' = R \backslash D \backslash V$,* **head_implicates**$(F)$ *returns only clauses $S \rightarrow x$ that are on the alphabet V and are consequences of F. If $F \models P' \rightarrow x$ and $Var(P' \rightarrow x) \subseteq V$ then $P' \rightarrow x$ is entailed by some clauses produced by* **head_implicates**(F) *with the non-deterministic choices $x \in V$ and $R' = R \backslash D \backslash V$.*

This theorem ensures the correctness of **head_implicates**(F) to forget variables and check common equivalence, as it produces only clauses over V that still entail the same consequences of F over V.

What about its efficiency? It works in polynomial space.

Theorem 8 *For every set of non-deterministic choices,* **head_implicates**(F) *works in polynomial space.*

4.5 Common equivalence

The algorithm **head_implicates**(F) generates a clause for each sequence of non-deterministic choices. The set of these clauses can be seen as a CNF formula. Its variables are all in V, and it has the same consequences of F on V by Theorem 7: it is a way to forget the variables of $Var(F) \backslash V$ from F. Common equivalence can be checked by forgetting all non-shared variables and then verifying regular equivalence: **head_implicates**$(A) \equiv$ **head_implicates**(B) with $V = Var(A) \cap Var(B)$. This is correct but may take not only exponential time but also exponential space because the output of **head_implicates**$()$ may be exponentially larger than A.

This is avoided by generating and checking a clause at a time: every clause produced by **head_implicates**(A) is checked against B for entailment, and the other way around.

```
## check common equivalence between two formulae
# input $A$: a formula
# input $B$: a formula
# output: true if $A$ is common equivalent to $B$, false otherwise
boolean common_equivalence(formula $A$, formula $B$)
 1. $V = Var(A) \cap Var(B)$
 2. for each $S \rightarrow x$ generated by head_implicates($A$) where $X$ is always chosen as a variable
    in $V$ and $R'$ is always chosen as $R \backslash D \backslash V$ in all function calls
    (a) if $B \not\models S \rightarrow x$ return false
 3. for each $S \rightarrow x$ generated by head_implicates($B$) where $X$ is always chosen as a variable
    in $V$ and $R'$ is always chosen as $R \backslash D \backslash V$ in all function calls
    (a) if $A \not\models S \rightarrow x$ return false
 4. return true
```

The following theorem proves the correctness of the algorithm.

Theorem 9 *Algorithm* **common_equivalence**(A, B) *returns whether A and B are common equivalent.*

4.6 Forget

Variables are forgotten by iteratively replacing them with other variables. This is what **head_implicates**() does by calling **body_replace**(): if the body of a clause $abc \rightarrow d$ contains a variable to forget A, it replaces A with the body of another clause with A in the head, like $ef \rightarrow a$. Many clauses may contain one or more variables to forget, and each variable to forget may be the head of multiple clauses. The choice of which variable to replace in which clause by which body is non-deterministic. A sequence of choices produces a single clause. Forgetting is the result of calling **head_implicates**(F) and collecting all clauses it produces in all its non-deterministic branches.

```
## forget variables $X$ from formula $F$
# input $F$: a formula
# input $X$: a set of variables
# output: a formula that expresses forgetting $X$ from $F$
formula forget(formula $F$, variables $X$)
 1. $G = \emptyset$
 2. for each $S \rightarrow x$ generated by head_implicates($F$) when it always chooses $x \in$
    $Var(F) \backslash X$ and $R' = R \backslash D \backslash (Var(F) \backslash X)$:
    (a) $G = G \cup \{S \rightarrow x\}$
 3. return $G$
```

Both **body_replace**() and **head_implicates**() depend on non-deterministic choices. Those of X and R' are specified in **forget**(), the others only affect the order of generated clauses, which is irrelevant to **forget**() as it was to **common_equivalence**().

The following lemma proves that **forget**(F, X) produces the expected result: a formula that is common equivalent to F and contains only the variables of F but X.

Theorem 10 *Algorithm* **forget**(F, X) *returns a formula on alphabet* $Var(F)\backslash X$ *that has the same consequences of* F *on this alphabet.*

The definition of forgetting in terms of common equivalence constrains the variables to be exactly $Var(F)\backslash X$. This is slightly different from the output of **forget**(F, X), which may not contain all of them. For example, when forgetting D from $F = \{a \rightarrow b, c \rightarrow d\}$, it produces $\{a \rightarrow b\}$, which does not contain $c \in Var(F)\backslash\{d\}$. This minor difference is removed by trivially adding tautologies like $\neg c \vee c$.

```
## forget variables $X$ from formula $F$
# input $F$: a formula
# input $X$: a set of variables
# output: a formula on variables $Var(F)\backslash X$ that is common equivalent to $F$
formula forget_ce(formula $F$, variables $X$)
    return forget$(F, X) \cup \left\{\bigvee\{x, \neg x \mid x \in Var(F)\backslash X\}\right\}$.
```

The formula returned by this algorithm is equivalent to the one returned by **forget**(F, X) since the added clause is a tautology. As a result, it has the same consequences: the same of F on the alphabet $Var(F)\backslash X$ by Lemma 10. This alphabet is also the set of their common variables.

Theorem 11 *Algorithm* **forget_ce**(F, X) *returns a formula that contains exactly the variables* $Var(F)\backslash X$ *and is common equivalent to* F.

Since **head_implicates**() works in non-deterministic polynomial space by Theorem 8, so does **forget**(). Like the common equivalence algorithm does, one this requires the clauses to be generated one at a time rather than all together. Each of them is then output rather than being space by Theorem 8, so does **forget**(). Like the common equivalence algorithm does, this one requires the clauses to be generated one at a time rather than all together. Each of them is then output rather than being accumulated in a set G.

While **common_equivalence**() is polynomial in space, **forget**() is only polynomial in working space; its output may be exponentially large. In such cases, both algorithms are exponential in time. A formula hitting these upper bounds is the classical example by Fisher et al. [44] mentioned above: $F = \{x_i \rightarrow z_i, y_i \rightarrow z_i \mid 1 \leq i \leq n\} \cup \{z_1 \ldots z_n \rightarrow w\}$.

The call **body_replace**$(F^w, \{z_1, \ldots, z_n\}, \emptyset)$ with $V = \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_n\} \cup \{w\}$ has two ways to replace z_1: by x_1 or by y_1. For each of these two non-deterministic choices, other two are generated for replacing z_2 by either x_2 or y_2. The same for z_3 and the other variables up to z_n. Doubling for each i from 1 to n is exponential in n: the final number of non-deterministic branches is 2^n. This means exponential time for **common_equivalence**() and exponential output for **forget**().

This is however not specific to these algorithms. Forgetting $\{z_1, \ldots, z_n\}$ from F always produces an exponential number of clauses. The formula $F' = \{P \rightarrow w \mid P \in$

body_replace$(F^w, \{z_1, \ldots, z_n\}, \emptyset)\}$ is minimal since all its clauses are superredundant [65]. All formulae equivalent to it are the same size or larger. Being exponentially large, they cannot be generated in less than exponential time. No algorithm is faster than **forget**() on it.

Yet, a small variant is backbreaking for **forget**(): removing x_n and y_n from V makes **body_replace**$(F^w, \{z_1, \ldots, z_n\}, \emptyset)$ output nothing, but still take exponential time if the variables are replaced in increasing order of i. The same exponentially many non-deterministic branches are produced, but they all eventually fail because z_n cannot be replaced.

A simple entailment check avoids these branches. The aim of **body_replace**$(F^w, \{z_1, \ldots, z_n\}, \emptyset)$ is to find a set $P' \subseteq V$ such that $F \cup P' \models \{z_1, \ldots, z_n\}$. The largest possible set P', the one having the most consequences, is V itself. If $F \cup V$ does not entail $\{z_1, \ldots, z_n\}$, no proper subset of V does. Computation can be cut short when this happens. Since F is Horn, this check only takes polynomial time:

$$\text{if } F \cup V \not\models P \text{ then } \textbf{fail}$$

This check is added right before each call to **body_replace**, both the first and the recursive ones.

In the example, $F \cup V$ does not entail z_n; therefore, it does not entail $\{z_1, \ldots, z_n\}$ either. The very first call of **body_replace** is skipped. As it should: no subset of V can ever replace $\{z_1, \ldots, z_n\}$.

Not only this check is useful in avoiding a call that would fail anyway. It otherwise guarantees its success — its usefulness. If $F \cup V \models P$ then **body_replace**() can replace P with some variables of V entailing it.

All of this is now formally proved: the added check does not harm the algorithm, but ensures the usefulness of the following call.

The proof of the following lemma requires that the first recursive call is done with an empty third argument: **body_replace**$(F, P, \emptyset)$. This is what happens anyway, but is specified because the lemma does not work if the third argument is not empty.

Lemma 14 *If R' is always chosen to be $R \backslash D \backslash V$ within a call* **body_replace**$(F, P, \emptyset)$*, then $F \cup V \models R \cup D \cup E$ holds after every successful subcall $S, E =$* **body_replace**(F', R, D).

When forgetting and checking common equivalence, the first recursive call has an empty third argument. Adding the instruction "if $F \cup V \not\models P$ then **fail**" before the recursive call does not affect the final result.

Lemma 15 *If $F \cup V \not\models P$, the recursive call* **body_replace**$(F^y, P, D \cup E')$ *fails if R' is always chosen equal to $R \backslash D \backslash V$.*

This lemma proves that the additional check "if $F \cup V \not\models P$ then **fail**" before a recursive subcall does not change the result of the algorithm, since the following subcall would fail anyway.

The contrary also holds: if the check succeds, the following call does not fail.

Lemma 16 *If $F \cup V \models P$ then* **body_replace**$(F^y, P, D \cup E')$ *succeeds if R' is always chosen equal to $R \backslash D \backslash V$.*

The check $F \cup V \models P$ improves efficiency because it requires polynomial time (being F Horn) but may save an exponential amount of recursive calls. It may look like it makes the running time polynomial in the size of the output [81], since every call succeeds and therefore produces some clauses. Yet, these are not guaranteed to be different from the previously generated ones. A counterexample is the following.

$$F = \{a_i \rightarrow x_i \mid 1 \leq i \leq n\} \cup \{x_i \rightarrow z_i, y_i \rightarrow z_i \mid 1 \leq i \leq n\} \cup \{z_1 \dots z_n \rightarrow w\}$$

When replacing $\{z_1, \dots, z_n\}$ with $V = \{a_i \mid 1 \leq i \leq n\} \cup \{w\}$, every z_i is replaced by either x_i or y_i, leading to exponentially many combinations. The additional check does not avoid them, since $F \cup V \models \{z_1, \dots, z_n\}$ holds. They all produce the same replacement $\{a_1, \dots, a_n\}$. This is correct because the only consequence of F on the alphabet of V is $a_1 \dots a_n \rightarrow w$. The algorithm takes exponential time to produce a single clause.

A simple test similar to $F \cup V \models P$ does not suffice. A recursive call searching for a replacement for P is useful only if it produces clauses not found so far. If these are $A_1, \dots, A_m$, one needs to check whether an element of each A_i can be removed from V so that the result still implies P with F. The problem is not that M may be exponential, since the aim is to bound the running time by a polynomial in M. It is the choice of the elements, since these may be 2^m even if each A_i only contains two variables.

These considerations extend from the forgetting algorithm to the common equivalence algorithm. It calls the subprocedure of head replacement, but always output a single bit. Its running time may be exponential in the size of the input, as suggested by the coNP-hardness result of Theorem 6, but comparing it with the size of the output does not make senze. Reducing it is still beneficial, just not measurable by the size of the output.

4.7 Single-head

The source of exponentiality in **body_replace**(F, R, D) is the non-deterministic choice of the clause $P \rightarrow y$. Many such clauses may exist, each spawning a branch of execution. This is not the case if each variable y is the head of at most a clause $P \rightarrow y$. Formulae with this property are called *single-head*.

This restriction removes non-determinism in the algorithm. The choice $R' \subseteq R \backslash D$ is forced to be $R' = R \backslash D \backslash V$ and $x \in Var(F)$ can be turned into a loop over the variables of F. The remaining non-deterministic choices are $P \rightarrow x \in F$ and $P \rightarrow y \in F$, which become deterministic since only a single such clause may exist for each given X and y.

The algorithm simplifies to: for each variable $x \notin V$, remove the only clause $P \rightarrow x \in F$ and replace all remaining occurrences of X with P. This takes polynomial time because once a variable is replaced, it disappears from the formula; it is never replaced again. While the polynomiality of this case looks obvious, it still requires a formal proof.

Lemma 17 *If F is a single-head definite Horn formula and V a set of variables, computing all possible return values of* **head_implicates**(F) *with the non-deterministic choices $x \in V$ and $R' = R \backslash D \backslash V$ takes time polynomial in the size of F.*

This proves that both forgetting and checking common equivalence take polynomial time in the single-head restriction.

Theorem 12 *Checking whether $F \equiv\equiv G$ holds can be verified in polynomial time if both F and G are single-head definite Horn formulae.*

5 Non-definite horn formulae

General Horn formulae allow for negative clauses like $\neg x \vee \neg y$, in addition to definite Horn clauses. Such formulae can be transformed so that all non-definite clauses are unary. Forgetting can be done by:

- making all non-definite clauses unary;
- running the algorithm in the previous section on the definite clauses;
- adding the non-definite (unary) clauses to the result.

Somehow, a non-definite clause $\neg x \vee \neg y$ has a head: it is the same as $xy \rightarrow \bot$. The head of a negative clause is the truth value of false. In this form, a negative clause is a definite clause, only with a special symbol as its head. The algorithm for common equivalence still works taking $\bot$ as a variable. Equivalently, it is a variable forced to be false.

Technically, a general Horn formula F can be put into a normal form where the only negative clauses are unary, that is, they only comprise a single (negative) literal.

Definition 4 The definite Horn part $def(F)$ of a Horn formula F is the set of definite clauses of F—the clauses of F that contain exactly one positive literal.

The definition implies $def(F) \subseteq F$ in general and $def(F) = F$ if F is definite Horn.

The negative clauses $F \backslash def(F)$ can be made definite by adding the same positive literal to all of them, or a different one to each. The common equivalence algorithm works either way. For this reason, the following definition does not constraint these additions.

Definition 5 A headed version of a Horn formula F is any definite Horn formula that comprises exactly the clauses of F, each one possibly added a variable not in F.

Since the result is Horn and definite, exactly the negative clauses are added a new positive literal. The definition does not forbid some of the added variables to be the same. Abusing notation, a headed version of F is denoted $headed(F)$, the added variables Z. A headed version of a formula is not equivalent to the formula since a clause C differs from a clause $C \vee z$. Yet, they are the same if z is always false: $headed(F) \cup \{\neg z \mid z \in Z\}$ is common equivalent to F.

The addition of $\{\neg z \mid z \in Z\}$ may look unnecessary when looking for common equivalence and the same variable z is the added head of all negative clauses: if $C \in A$ and $A \equiv\equiv B$, then B implies C. Therefore, $headed(A)$ contains $C \vee z$ and $headed(B)$ implies $C \vee z$. The other direction is false. The problem is that $headed(A)$ and $headed(B)$ may differ on whether they entail a definite clause $P \rightarrow x$, but this difference is leveled by another clause $P' \rightarrow z$ with $P' \subseteq P$. When z is not constrained to be false these two clauses are independent; when it is, the second supersedes the first, making it redundant and canceling the difference.

Checking formulae for redundancy is not the solution, as the differing clause may be entailed rather than present in a formula. The following is an example.

$$A = \{\neg a \vee \neg b, a \rightarrow a', b \rightarrow b', a'b' \rightarrow c, c \rightarrow d\}$$
$$B = \{\neg a \vee \neg b, c \rightarrow d\}$$

The headed first formula $headed(A)$ entails $ab \rightarrow c$ while the headed second $headed(B)$ does not. The clause is on their common variables $Var(A) \cap Var(B) = \{a, b, c, d\}$. It proves that $headed(A)$ and $headed(B)$ are not common equivalent. Yet, A and B are common equivalent. The differing clause $ab \rightarrow c$ is entailed by $\neg a \vee \neg b$, which is in both formulae. This is the only case where equivalence in the definite part of the formulae does not entail equivalence in the whole: a differing definite clause is superseded by a negative clause that is entailed by both formulae.

The solution is to make this negative clause stand out from the definite. It does as soon as its new head is forced to be false: since $headed(B)$ contains $ab \rightarrow z$ for some $z \in Z$, adding $\neg z$ makes this clause equivalent to $\neg a \vee \neg b$, which entails $ab \rightarrow z$.

The general solution is to set all new heads $z \in Z$ to false.

The replacement of the non-shared variables is done on the definite Horn version $headed(A)$ and $headed(B)$ since **head_implicates**() only works on definite Horn clauses; the new heads are negated in the resulting formula.

```
## check common equivalence of two general Horn formulae
# input $A$: a Horn formula
# input $B$: a Horn formula
# output: true if $A \equiv\equiv B$ holds, false otherwise
boolean common_equivalence_horn($A$, $B$)

1. $V = Z \cup (Var(A) \cap Var(B))$
2. for each $P \rightarrow x$ generated by head_implicates($headed(A)$) where $X$ is always chosen
   in $V$ and $R'$ is always chosen as $R \backslash D \backslash V$
    (a) if $B \cup \{\neg z \mid z \in Z\} \not\models P \rightarrow x$ return false
3. for each $P \rightarrow x$ generated by head_implicates($headed(B)$) where $X$ is always chosen
   in $V$ and $R'$ is always chosen as $R \backslash D \backslash V$
    (a) if $A \cup \{\neg z \mid z \in Z\} \not\models P \rightarrow x$ return false
4. return true
```

The following lemma proves that no matter how heads are added to the negative clauses, the result of this algorithm is correct.

Lemma 18 *The common equivalance $A \equiv\equiv B$ holds if and only if $B \cup \{\neg z \mid z \in Z\} \models$* **head_implicates**$(headed(A))$ *and $A \cup \{\neg z \mid z \in Z\} \models$* **head_implicates**$(headed(B))$*, if $V = Z \cup (Var(A) \cap Var(B))$.*

The same mechanism works for forgetting: make negative clauses definite by adding new variables as heads, forget, set the new heads to false. The shorthand $[\bot/Z]$ stands for the set of substitutions $[\bot/z]$ for all $z \in Z$.

```
## forget variables $X$ from a general Horn formula $F$
# input $F$: a Horn formula
# input $X$: a set of variables
# output: a formula expressing forgetting $X$ from $F$
formula forget_horn(formula $F$, variables $X$)

1. $F' =$ forget_ce($headed(F)$, $X$)
2. return $F'[\bot/Z]$
```

This algorithm is proved correct by the following lemma regardless of how $headed(F)$ assigns the new heads to the negative clauses.

Lemma 19 *For every formula F and set of variables X, if $F' = \textbf{forget\_ce}(headed(F), X)$ then $F'[\bot/Z]$ contains exactly the variables $Var(F)\backslash X$ and is common equivalent to F.*

This lemma ensures the correctness of **forget_horn**() regardless of whether the added heads are different or not.

6 Single-head minimization

The algorithm for forgetting generates a formula expressing forgetting. This formula may be exponentially large. Yet, it is not the only formula expressing forgetting. Another may be smaller. When the formula is single-head, forgetting and minimization can be done separately: first forget, in polynomial time; then minimize. The problem of forgetting within a certain size reduces to the classic Boolean minimization problem [23, 24, 52, 70, 76, 82, 87] when the formula is single-head.

Yet, the single-head case has some specificity:

- the minimally sized formula equivalent to a given single-head one may not be single-head;
- therefore, minimizing is actually two distinct problems: find a minimal formula or a minimal single-head formula;
- acyclic formulae are easy to minimize, clause by clause;
- this cannot be done in the cyclic case;
- a sufficient condition to minimality exists; it provides directions for minimizing when a formula is not minimal.

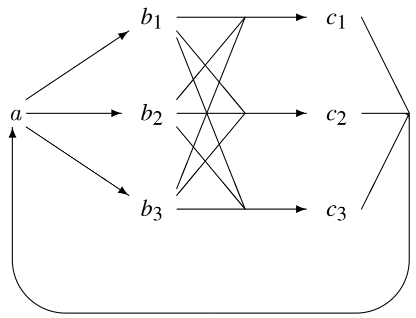
The first point is that minimizing a single-head formula depends on whether the minimal formula is required to be single-head or not. This is proved by an example, a single-head formula that is equivalent to a non-single-head one of size k but to no single-head one of the same or lower size.

It hinges on how equivalences are achieved in single-head formulae: by loops of implications. These are implications and equivalences between sets of variables. To simplify them, the notation $A \rightarrow B$ stands for the first formulae implying the second: $\{A \rightarrow b \mid b \in B\}$. The notation $A \equiv B$ stands for their equivalence: $(A \rightarrow B) \cup (B \rightarrow A)$.

If F implies $A \equiv B$ and $B \equiv C$ for three disjoint sets of variables A, B and C, the single-head condition requires each set to imply another in a loop. An example is $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$. This is not the case in general, where the same equivalences can be realized by entailments centered on one set, for example A, like in $A \rightarrow B$, $B \rightarrow A$, $A \rightarrow C$ and $C \rightarrow A$. An example shows that this may be smaller.

$$F = \{a \rightarrow b_1, a \rightarrow b_2, a \rightarrow b_3, b_1b_2b_3 \rightarrow c_1, b_1b_2b_3 \rightarrow c_2, b_1b_2b_3 \rightarrow c_3, c_1c_2c_3 \rightarrow a\}$$

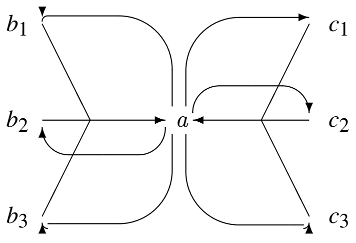
This formula is a specific case of the example above where $A = \{a\}$, $B = \{b_1, b_2, b_3\}$ and $C = \{c_1, c_2, c_3\}$. It is a single-head formula that entails the equivalence of A, B and C by a loop of entailments: A entails B, which entails C, which entails A.



This formula has 22 literal occurrences. Every other single-head equivalent formula has the same size: every loop of sets of size 1, 3 and 3 is a closed sequence with a $1-3$ edge, a $3-3$ edge and a $3-1$ edge.

An alternative that is not single-head is to state the equivalence of both B and C with A.

$$F' = \{a \rightarrow b_1, a \rightarrow b_2, a \rightarrow b_3, b_1b_2b_3 \rightarrow a, a \rightarrow c_1, a \rightarrow c_2, a \rightarrow c_3, c_1c_2c_3 \rightarrow a\}$$



.

This formula is smaller because it centers around the small set $A = \{a\}$, saving from the costly entailment between two sets of size three. The size of F' is 20.

This formula is not single-head as it contains two clauses with head A. The answer to the question "is the single-head formula F equivalent to a formula of size 20 or less" is "yes". The answer to the similar question "... equivalent to a *single-head* formula of size 20 or less" is "no".

Regardless of whether the minimal formula has to be single-head or not, finding it is polynomial if the clauses form no loop, where each clause $P \rightarrow x$ links every variable in P to X [53]. Acyclicity is a subcase of in-equivalence: no two minimally-sized different sets of variables are equivalent.

Condition 1 *A formula F is* in-equivalent *if $F \models A \equiv B$ implies $F \models A \equiv A \cap B$ for every two sets of variables A and B.*

The minimal form of an in-equivalent single-head formula F can be determined from the order $B \leq_F A$ defined as $F \models A \rightarrow B$. The strict part of this order $B <_F A$ is as usual: $B \leq_F A$ but not $A \leq_F B$. The set $MIN(F)$ is built from this order.

$$MIN(F) = \{A \rightarrow x \mid F \models A \rightarrow x \text{ and } \nexists B \,.\, x \notin B,\ F \models B \rightarrow x \text{ and } B <_F A \text{ or } B \subset A\}$$

Every formula equivalent to F implies $MIN(F)$ since the clauses in $MIN(F)$ are all entailed by F. In the case of in-equivalence, this fact can be pushed further: every formula equivalent to F contains either a clause of $MIN(F)$ or a superclause of it; a superclause of another is a clause that contains at least all literals of another.

Lemma 20 *If F is in-equivalent, it contains a superclause of every clause in $MIN(F)$.*

Another important property of $MIN(F)$ is that it is equivalent to F if F is single-head [66].

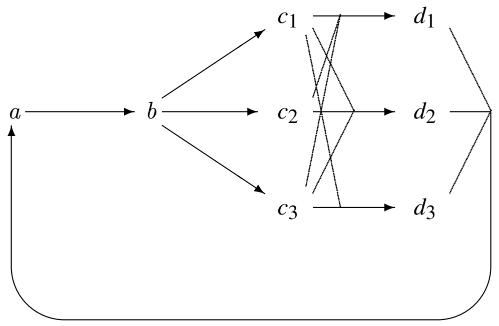
If F is single-head then $MIN(F)$ is single-head as well. This makes the problems of the minimal formula and the minimal single-head formula coincide. Since $MIN(F)$ can be determined in polynomial time if F is in-equivalent, both problems are polynomial in time in the single-head in-equivalent case.

Does tractability extend from in-equivalent single-head formulae to the general case? A counterexample suggests it does not.

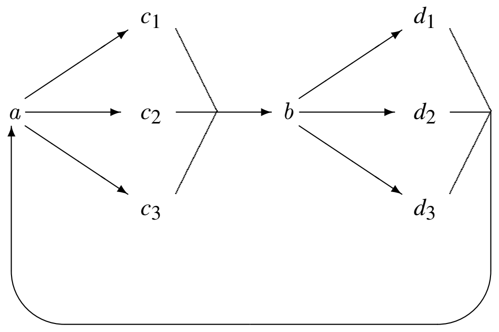
Building $MIN(F)$ is easy: each $P \rightarrow x \in F$ is minimized by repeatedly replacing it with $P' \rightarrow x$ if such a clause is entailed by F and either $P' <_F P$ or $P' \subset P$. This is not in general possible on formulae containing loops.

As an example, four sets of variables sized 1, 1, 3 and 3 are made equivalent by a loop of clauses making each implies its next. This depends on the order of the sets in the loop: the order 1, 1, 3, 3 comprises 24 literal occurrences, the order 1, 3, 1, 3 only 20.

$$F = \{a \rightarrow b, b \rightarrow c_1, b \rightarrow c_2, b \rightarrow c_3, c_1c_2c_3 \rightarrow d_1, c_1c_2c_3 \rightarrow d_2, c_1c_2c_3 \rightarrow d_3, d_1d_2d_3 \rightarrow a\} \quad (1)$$



$$F' = \{a \rightarrow c_1, a \rightarrow c_2, a \rightarrow c_3, c_1c_2c_3 \rightarrow b, b \rightarrow d_1, b \rightarrow d_2, b \rightarrow d_3, d_1d_2d_3 \rightarrow a\} \quad (2)$$



While F is not minimal, all its individual clauses are minimal: F entails no proper subset of $a \rightarrow b$, none of $b \rightarrow c_1$, and so on.

The question is: how to minimize a single-head formula that is not in-equivalent? The target is a formula that is single-head and minimal: no other single-head formula is smaller. A simple sufficient condition is based on the following property.

Lemma 21 *If F is equivalent to a single-head formula F' that contains the clause $P \rightarrow x$, then F contains $P' \rightarrow x$ with $F \models P \equiv P'$.*

The sufficient condition not only says when a formula is not minimal, but also shows how clauses can be replaced.

Lemma 22 *If a single-head definite Horn formula F is not minimal then it contains a clause $P \rightarrow x$ such that $F \models (BCN(P, F)\backslash\{y\}) \rightarrow y$ where $y \in P$ and $BCN(P, F) = \{x \mid F \cup P \models x\}$.*

If F is not minimal, it contains a clause $P \rightarrow x$ such that $F \models (BCN(P, F)\backslash\{y\}) \rightarrow y$ with $y \in P$. This condition can be checked easily: for each clause $P \rightarrow x$, determine the set of all its consequences $BCN(P, F)$ by iteratively adding X to P whenever a clause $P' \rightarrow x$ is in F with $P' \subseteq P$. When no such clause remains, check $F \models (BCN(P, F)\backslash\{y\}) \rightarrow y$ for each $y \in P$. Such an entailment suggests that $P \rightarrow x$ could be replaced by a clause $P' \rightarrow x$ where P' does not contain y. In other words, it not only says that the formula is not minimal, but it also gives some directions for minimizing it. Yet, whether minimizing a single-head formula is tractable in general is an open problem.

7 Extending the alphabet

Some formulae cannot be reduced in size by adding new variables, others can. For example, $F = \{abcd \rightarrow e, abch \rightarrow f, abci \rightarrow g\}$, shortens to $F' = \{abc \rightarrow n, nd \rightarrow e, nh \rightarrow f, ni \rightarrow g\}$, of size 13 instead of 15. Size reduction is due to summarizing abc as n: the new clause $abc \rightarrow n$ allows replacing each occurrence of abc with the single new variable n.

This mechanism is the base of an algorithm that attempts to reduce the size of a formula by adding new variables. A large set of variables present in many bodies is summarized by a new variable thanks to an added clause; all previous occurrences of the set are replaced by the new variable. This summarization is repeated as long as it shortens the formula. The complete algorithm is not optimal for two reasons: it is greedy (only works a single subset at a time) and does not always find the best summarizable set of variables.

The core of the algorithm is a function that replaces every occurrence of a set of variables by a new one.

$$newvar(P, F) = \{((A\backslash P)\cup\{x\}) \rightarrow y \mid A \rightarrow y \in F,\ P \subseteq A\}\cup\{A \rightarrow y \mid A \rightarrow y \in F, P \not\subseteq A\}\cup\{P \rightarrow x\}$$

The result of newvar(P, F) is common-equivalent to F regardless of P.

Lemma 23 *For every formula F and set of variables A, it holds $F \equiv\equiv newvar(A, F)$.*

Calling newvar(P, F) as many times as needed is not a problem, since it is linear-time. Therefore, it can be called to gauge the gain from summarizing a set of variables P. A complete algorithm could call it over all subsets P and check how large the results are, then replace F with the shortest and repeat.

The problem is the exponential number of sets of variables. Yet, many can be excluded. Only the sets that are contained in a body need to be checked. Still better, since the aim is to reduce size, only the sets that are contained in at least two bodies may be useful. Two bodies are sometimes enough: $F = \{abcdefg \rightarrow i, abcdefh \rightarrow j\}$ shortens to $F' = \{abcdef \rightarrow n, ng \rightarrow i, nh \rightarrow j\}$.

Lemma 24 *If $|newvar(A, F)| \leq |F|$ for some set of variables A then $|newvar(B, F)| \leq |newvar(A, F)|$ for some intersection B of the bodies of some clauses of F.*

This lemma gives directions to the algorithm by restricting to sets P that are intersections of bodies of the formula.

This is not enough for polynomiality because the intersections are exponentially many. Another considerationi further shortens searching: size tends to decrease more for large sets P than for small, and the intersection of few sets tends to be larger than the intersection of many. Following this principle, the algorithm searches for the best intersection of two bodies and then tries to improve it by intersecting it with a third body and so on.

This procedure is always correct because it repeatedly applies newvar(P, F), which is proved common-equivalent to F by Lemma 24. Since it only performs a replacement with a shorter formula, it is guaranteed to reduce size.

```
## try to reduce size by introducing new variables
# input $F$: a Horn formula
# output: a formula that is common equivalent to $F$
formula **minimize**(formula $F$)

1. $F' = F$
2. for each $A \rightarrow x, B \rightarrow y \in F$
   (a) if $|newvar(A \cap B, F)| < |F'|$
        i. $N = A \cap B$
       ii. $F' = newvar(N, F)$
3. if $F' = F$ return $F$
4. $F'' = F$
5. for each $C \rightarrow x \in F$
   (a) if $|newvar(N \cap C, F)| < |F''|$
        (i.) $M = N \cap C$
       (ii.) $F'' = newvar(M, F)$
6. if $F'' \neq F'$
   (a) $N = M$
   (b) goto Step 4
7. $F = F''$
8. goto 1
```

Running time is polynomial in the size of the formula. The first loop is quadratic in the size of the formula, the second is linear. The jump instructions are performed only if the formula shrinks. Therefore, the number of runs of the loops is at most linear in the size of the formula.

This algorithm is proved correct by the following lemma. It does not always shorten the formula F, which may already be minimal. Yet, it never enlarges F. That it is optimal is later disproved by a counterexample.

Lemma 25 *The formula returned by* **minimize**(F) *is common-equivalent to F and not larger than it.*

This lemma proves that **minimize**(F) meets the minimal requirements for correctness and usefulness: it outputs a formula that is the same as the input apart from the new variables and is shorter or the same size. Hopefully, it is shorter. Ideally, it is as short as possible. This is not always the case, as the following counterexample shows.

$$F = \{x_1x_2x_3x_4x_5x_6 \rightarrow z_1, x_1x_2x_3x_4x_5x_6 \rightarrow z_2, x_1x_2x_3 \rightarrow z_3, x_4x_5x_6 \rightarrow z_4\}$$

The intersections of two bodies of this formula are $A = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $B = \{x_1, x_2, x_3\}$, $C = \{x_4, x_5, x_6\}$ and $D = \emptyset$. The first step the algorithm takes is determining the minimum between newvar(A, F), newvar(B, F), newvar(C, F) and newvar(D, F). These formulae and their size are in the table below.

formula	size
$\text{newvar}(A, F) = \{x_1x_2x_3x_4x_5x_6 \rightarrow y, y \rightarrow z_1, y \rightarrow z_2, x_1x_2x_3 \rightarrow z_3, x_4x_5x_6 \rightarrow z_4\}$	19
$\text{newvar}(B, F) = \{x_1x_2x_3 \rightarrow y, yx_4x_5x_6 \rightarrow z_1, yx_4x_5x_6 \rightarrow z_2, y \rightarrow z_3, x_4x_5x_6 \rightarrow z_4\}$	20
$\text{newvar}(D, F) = \{y, yx_1x_2x_3x_4x_5x_6 \rightarrow z_1, yx_1x_2x_3x_4x_5x_6 \rightarrow z_2, yx_1x_2x_3 \rightarrow z_3, yx_4x_5x_6 \rightarrow z_4\}$	27

The algorithm chooses A. It then checks whether intersecting $A = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ with another body further reduces size. These intersections are equal to B and C, which have been shown to produce a larger formula instead. Therefore, the first step ends with newvar(A, F) replacing F.

The second step tries to further reduce the size of newvar(A, F) by an intersection of its bodies. The intersections are the same as B, C and D. As shown above, newvar$(\emptyset$, newvar$(A, F))$ is never smaller than newvar(A, F), which excludes D. Also, C can be disregarded by symmetry: only newvar$(B$, newvar$(A, F))$ is analyzed. Its size is 19, the same as newvar(A, F). This formula is not shorter than the current. Therefore, the algorithm returns the current: newvar(A, F).

$$\begin{aligned}\text{newvar}(B, \text{newvar}(A, F)) = \{&x_1x_2x_3 \rightarrow y', y'x_4x_5x_6 \rightarrow y,\\ &y \rightarrow z_1, y \rightarrow z_2, y' \rightarrow z_3,\\ &x_4x_5x_6 \rightarrow z_4\}\end{aligned}$$

A shorter common equivalent formula exists, and can even be found by a sequence of newvar() calls. Using B and C on F produces the following formula, of size 18.

$$\begin{aligned} newvar(B, newvar(C, F)) = \{&x_1x_2x_3 \rightarrow y, x_4x_5x_6 \rightarrow y',\\ &yy' \rightarrow z_1, yy' \rightarrow z_2, y \rightarrow z_3, y' \rightarrow z_4\}\end{aligned}$$

In summary: the algorithm first finds the single best intersection A of two bodies; it then tries to intersect it with another body, but none further reduces size; it therefore replaces F with newvar(A, F); on this new formula, it searches all intersections of two bodies, but none reduces size. Yet, a smaller formula exists, and is obtainable by two non-minimal intersections.

The problem is with the greedy procedure: the bird in the hand A is not worth the two B and C in the bush. Every algorithm driven by the best immediate gain falls in this trap.

The counterexample is single-head and acyclic (in-equivalent). Even in this very restricted case, the greedy algorithm may fail to find the shortest common-equivalent formula. So does any other polynomial-time algorithm, the NP-completeness of the problem suggests.

Membership to NP is easy to prove.

Lemma 26 *Given a single-head formula F and an integer l, deciding the existence of a common-equivalent single-head formula G such that $Var(F) \subseteq Var(G)$ and $|G| \leq l$ is in* NP.

NP-hardness holds even in the subcase of single-head acyclic definite Horn formulae. It does not require the output formula to be single-head. In other words, whether a single-head acyclic definite Horn has a common-equivalent formula of a certain size is NP-hard, regardless of whether such an equivalent formula is required to be single-head or not.

The reduction is from the vertex cover problem. Given a graph (V, E), where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$, a vertex cover is a subset $C \subseteq V$ such that either $v_i \in C$ or $v_j \in C$ for every $e_l = (v_i, v_j) \in E$. The vertex cover problem is to establish whether a given graph has a vertex cover of size k or less.

Definition 6 *The* shortening-extendable formula *of a graph* (V, E), where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$, is the following formula, defined on the alphabet that comprises five variables v_i, r_i, r'_i, s_i and s'_i for each node $v_i \in V$, two variables e_l and e'_l for each edge $e_l \in E$ and a single other variable w.

$$\begin{aligned} A = \{&v_i w v_j \rightarrow e_l,\ v_i w v_j \rightarrow e'_l \mid e_l = (v_i, v_j) \in E\} \cup \\ \{&v_i w r_i \rightarrow s_i, v_i w r'_i \rightarrow s'_i \mid v_i \in V\} \end{aligned}$$

For each edge (n_i, n_j) in the graph, the formula is shortened by summarizing either $v_i w$ or $v_j w$ as a new variable. The first change alone shortens the formula by one literal, the second alone does the same by symmetry.

$$\begin{array}{rcl} v_i w v_j \rightarrow e_l & \Rightarrow & y_i v_j \rightarrow e_l \\ v_i w v_j \rightarrow e'_l & \Rightarrow & y_i v_j \rightarrow e'_l \\ v_i w r_i \rightarrow s_i & \Rightarrow & y_i r_i \rightarrow s_i \\ v_i w r'_i \rightarrow s'_i & \Rightarrow & y_i r'_i \rightarrow s'_i \end{array}$$

The two shortenings do not sum up: once $v_i w v_j \rightarrow e_l$ is shortened to $y_i v_j \rightarrow e_l$, it is not shortened further by summarizing $v_j w$. This mechanisms corresponds to covering the edges of a graph: one node for each edge is necessary; both nodes are allowed but not necessary. In the same way, replacing either $v_i w$ or $v_j w$ with a new variable reduces size, replacing both only maintains it. The first is necessary, the second is not: it may shorten some other edges, but not this one.

The problem to be proved hard is whether the size of a formula can be reduced below a certain bound M by adding new variables. The formal proof comprises three steps:

1. the formula A in Definition 6 is a minimal formula; it cannot be shortened without introducing new variables;

2. if a formula is common-equivalent to A, it can be put in a certain form without a size increase while maintaining common equivalence; this form is the one employed in the proof sketch above: new clauses $v_i w \rightarrow y_i$ are introduced to shorten other clauses by replacing $v_i w$ with y_i; it is acyclic and single-head, proving that at least an acyclic single-head formula is minimal;
3. such formulae correspond to vertex covers.

The first part of the proof shows that A is minimal.

Lemma 27 *The shortening-extendable formulae of graphs are all minimal.*

The second lemma is the core of the proof. It shows how to optimally shrink a shortening-extendable formula by possibly adding new variables.

Lemma 28 *If a formula B is common-equivalent to a shortening-extendable formula A and $Var(A) \subseteq Var(B)$, then B is at least as large as a formula obtained from A by introducing some clauses $v_i w \rightarrow y_i$ and replacing some occurrences of $v_i w$ with y_i.*

The proof is completed by linking the vertex covers of a graph to the formulae built according to the previous lemma.

Lemma 29 *A graph $G = (V, E)$ has a vertex cover C if and only if a formula built from its shortening-extendable formula A by introducing some clauses $v_i w \rightarrow y_i$ and replacing some occurrences of $v_i w$ with y_i has size $|E| \times 6 + |C| + |V| \times 8$.*

These three lemmas prove the problem NP-hard.

Theorem 13 *Given a single-head minimal-size acyclic definite Horn formula A and an integer m, deciding the existence of a common equivalent definite Horn formula B such that $Var(A) \subseteq Var(B)$ and the size of B is bounded by m is* NP*-hard. The same holds if B is constrained to be acyclic or single-head.*

Minimizing a formula by introducing new variables is NP-hard in the single-head case, and remains hard in the acyclic case even when releasing the constraint on the minimal formula to be acyclic or single-head. This complexity excuses the greedy mechanism of **minimize**(F) for not being optimal.

8 Implementations

The algorithms presented in this article have been implemented in Python [86] and Bash [4]: forgetting, common equivalence verification and minimization by variable introduction. They can currently be retrieved from https://github.com/paololiberatore/forget.

The most relevant issue is how non-determinism is realized. The three solutions are approximation, sets and multitasking.

approximation: when the input formula is single-head, the choice of a clause $P \rightarrow y \in P$ with head y becomes deterministic since only zero or one such clause may exist; it is still realized as a loop, but only the first clause with head y is used; this is correct as long as the formula is single-head; the algorithm is simple as it does not implement any special mechanism for non-determinism, but is only correct on single-head formulae;

sets: the function **body_replace()** returns a set of pairs instead of a single pair as in the algorithm; each pair is a way to replace the variables to forget; each is further pursued by the caller, possibly leading to other replacements; these are collected in a set of pairs, which is returned; the increased complication of the code was expected, but this implementation suffers from a more serious drawback: since the non deterministic choices are realized by sets, these may grow exponentially even if the output does not;

multitasking: each non-deterministic possible choice is made into a branch of execution; for each non-deterministically chosen clause, execution forks into a branch for that clause and another that waits for the first to terminate and only then chooses another clause; the cost of forking is small in operating systems implementing copy-on-write [78]; the number of processes in simultaneous execution is also small: each non-deterministic choice generates a waiting parent and a running child; since the parent is waiting it does not spawn any other process until the child terminates; overall, only one process is running while a polynomial number of other processes are waiting, one for each unfinished recursive call; polynomiality is guaranteed by the second parameter of **body_replace()**, which forbids replacing the same variable twice.

9 Related work

Common equivalence formalizes forgetting [28], but also variable introduction [14] and both. Some results in this article relate to similar ones about forgetting. The reformulation of common equivalence in terms of equisatisfiability (Theorem 1 and Theorem 2) extends similar properties of forgetting [65]. That adding a negative literal or replacing its variable with false are the same (Lemma 4) could be recast as a consequence of some properties of forgetting: Proposition 6 and Proposition 7 by Lang et al. [62].

The Π_2^p-hardness of common equivalence (Theorem 4) extends the similar result for var-equivalence by Lang et al. [62] by showing that var-equivalence is not only Π_2^p-hard in general, but also when equivalence is on the variables shared by the two formulae. The proof by Lang et al. exploits the var-equivalence of a formula with $\top$ on a set of variables A which may not be empty. Such a proof does not work for common equivalence, as the set of common variables is empty in this case.

The coNP-hardness of common equivalence in the Horn case extends the similar proof for restricted equivalence by Flogel et al. [45]. The two formulae they check for equivalence share some variables that are not among the restricted ones. This is not common equivalence.

Formula size has been analyzed by several authors in different areas. They mostly show the output size of a given, fixed algorithm [1, 47] or the size across all formulae [28, 35, 37, 61]. The minimal achievable size for a specific formula regardless of the algorithm is only considered by Zhou [100], who proved that checking the existence of a formula of a given size expressing forgetting from another formula is Σ_2^p-complete. That other formula is not constrained to have minimal size. This allows for reductions in size that are not due to forgetting but to the redundancy of that other formula. Forgetting from minimally-sized formulae constraints the reduction in size to be due to forgetting [65].

Minimizing a formula without forgetting variables is a large research area [15, 24]. Its complexity sparked the study of the polynomial hierarchy [79]. From the point of view of computational complexity, the decision problem is the existence of a Boolean formula of a given size that is equivalent to a given formula. This problem remained open for about twenty years before being closed [79, 83].

The comparison with common equivalence shows similarities and differences. The decision problem in common equivalence is the existence of a common equivalent formula of a given size. It resembles formula minimization, the existence of an equivalent formula of a given size. It differs because equivalence on the shared variables differs from equivalence on all variables. Two common equivalent formulae are not always equivalent.

Producing a formula of small size is formula synthesis [24, 70]. While formula minimization takes a Boolean formula as its input, formula synthesis takes a Boolean function expressed in some form. Since a Boolean formula expresses a Boolean function, is a valid input to formula synthesis. Minimizing a formula is therefore a particular case of synthesizing a formula of minimal size, which relates to common equivalence. In theory. In practice, formula minimization and formula synthesis differ in their aims. The main aim of formula minimization is a formula of minimal size, or at least of a given size. The main aim of formula synthesis is producing a formula that represents a function; small size is a desideratum, not a constraint. Synthesis algorithms [24, 70] try to quickly generate small formulae, but they do not guarantee minimality. Formula minimization aims at minimality; speed and therefore success within a given time limit are not primary concerns.

Forgetting variables that are not given but chosen by the algorithm is usually called variable elimination. Its aim is to simplify a formula before some kind of processing, such as checking satisfiability [31, 80]. Contrary to forgetting, the variables to remove are not given. The algorithm chooses the ones whose removal simplifies the formula the most. If no removal simplifies the formula, nothing is removed. Forgetting may be aimed at formula simplification like variable elimination is, but removing variables is its very definition.

Common equivalence is variable removal and variable introduction at the same time. Each has been studied in isolation. Like variable elimination, variable introduction aims at simplifying a formula [14, 46, 73, 95]. It also aims at prove its properties [2, 16].

Knowledge compilation also aims at simplifying the formula, but differs fromvariable elimination and forgetting as it maintains the whole content of the formula in most cases. An exception are Horn envelopes [13], which approximate a CNF formula by a Horn formula. Reasoning in the Horn fragment is easier than in the general case, possibly repaying the cost of the approximation. Most other compilation techniques aim at simplifying a formula without loss of information [19, 26].

Simplifying a formula may be a goal of common equivalence and forgetting, but not always. Forgetting may be motivated by privacy regulations rather than efficiency. Furthermore, it is a form of approximation, since the consequences of the original formula that involve the forgotten variables are lost. Most trends in knowledge compilation retain the exact semantics of the formula, and aim at efficiency [27].

Definite Horn formulae are close to functional dependencies in databases [7]. The formula that proves the exponentiality in size of forgetting comes from the study of functional dependencies [44, 51]. The resolution-based algorithm for propositional forgetting [28, 30, 91] is the same as a much earlier algorithm to calculate the inferred functional dependencies on a subset of attributes [51]. Later works on functional dependencies extend them in various ways [39, 54, 90].

Other logics where forgetting is investigated are description logics [33, 96], modal logics [85, 98], temporal logics [42], logics for reasoning about actions [37, 75], and defeasible logics [1].

Forgetting in description and modal logics is often studied as its dual concept of uniform interpolation. An interpolant between two formulae is another formula on their common variables that is entailed by the first and entails the second [25]. An interpolant is close in its definition and spirit to common equivalence, as it works on the set of common variables and it involves entailment, which is one part of equivalence. Equivalence is however both entailment and entailment in reverse.

Forgetting is instead the dual of uniform interpolation, the existence of a single interpolant for all formulae on a given alphabet [56]. The uniform interpolant is the result of forgetting the other variables, if one exists. The early algorithm by Wang et al. [88] employs a rule similar to replacing heads with bodies. This rule is then extended to a form similar to general resolution. A number of later algorithms employ resolution to add consequences and then remove the symbols to forget [60, 61, 69, 94]. Resolution generalizes replacing heads with bodies in no specific order.

10 Conclusions

Some comments about the results in this article:

- The increase in size due to variable forgetting may be countered by the introduction of new variables. Common equivalence is the semantical foundation of this process. In turns, common equivalence can be expressed in terms of forgetting. However, the computational properties of the two concepts differ: while both take a polynomial amount of memory, only forgetting may produce an exponentially large output. Checking common equivalence by forgetting is always possible but requires some care to avoid an unnecessarily large consumption of memory. In addition, some properties of common equivalence such as its limited transitivity and its insensitivity to new variables are not obvious to express in terms of forgetting. Yet, checking common equivalence by the forgetting algorithm has a clear advantage in terms of efficiency over implementing the guess-and-check algorithm implied by the complexity results: the latter is always exponential in time, the former is only in the worst cases.
- Common equivalence is Σ_2^p-complete even when it expresses forgetting and coNP-complete for Horn formulae. Unsurprisingly, it is one level higher in complexity than entailment and satisfiability. This is a common trait of many forms of complex reasoning [6, 32]. Efficient solvers for problems of this complexity exist [3, 74], but the increase in hardness still affects speed.
- An algorithm for forgetting and checking common equivalence in the Horn fragment is presented. It runs in polynomial space. It is equivalent to resolving out or unfolding variable occurrences in a certain order. It does not process all occurrences of each variable at time as previous algorithms do [28, 30, 51, 91], which may exponentially enlarge the formula. A side effect of this result is that while forgetting multiple variables is semantically equivalent to forgetting each variable at time, it is computationally different. The algorithm presented in this article forgets a set of variables in polynomial space, which may not be possible when forgetting variable by variable.
- The algorithm runs in polynomial time, in addition to polynomial space, when each variable is the head of one clause at most. If loops of variables are also forbidden, forgetting can be performed in polynomial time, and the result can be minimized in polynomial time. This additional constraint can be satisfied in simple cases like A equivalent to B by replacing all occurrences of B with A.
- An algorithm for minimizing a formula when new variables can be introduced is shown. It is not optimal, but the problem itself is NP-hard even in the single-head acyclic case. According to the prevailing opinion in complexity theory, an algorithm for such a problem is either incomplete or exponential in time. The one presented in this article is incomplete: it does not always find a minimal formula. At the same time, it is guaranteed not to enlarge the formula.

The single-head restriction formalizes situations where each fact is a consequence of a single set of premises, nothing has alternative causes. While the frequency of such cases in practice is still unclear, the restriction also allows for an algorithm that runs in the general case: it selects one clause per head in all possible combinations and conjoins the results. Indeed, a propagation never needs to derive a variable more than once; the clause used for that is necessary, the others with the same head are not. If the equal heads are too many, some selections of clauses can be discarded. This is as a form of approximated knowledge compilation, like for example Horn envelopes [12], that needs further investigation. Theorem 13 proves that minimizing the result of forgetting is hard even in very restricted cases; releasing the strict minimality constraint may be necessary when using forgetting in knowledge compilation.

A different kind of solution is to cap the size of the formulae produced by forgetting while maintaining the consequences of the original formula as much as possible. If the size bound is a hard constraint, something which cannot be overcome, this may be the only possible solution when no formula expressing forgetting is sufficiently small. Capping size changes propositional forgetting, like approximation changes formula minimization [9, 52], bounding the number of quantifiers changes first-order forgetting [101] and limiting size changes several PSPACE problems [64].

Another solution is to turn a formula single-head if possible. For example, $\{a \rightarrow b, b \rightarrow a, b \rightarrow c, c \rightarrow b\}$ is not single-head, but is equivalent to the single-head formula $\{a \rightarrow b, b \rightarrow c, c \rightarrow a\}$. The problem of identifying such formulae is not trivial as it may look [66].

An open question is whether minimization is polynomial-time or NP-hard in the single-head cyclic case with no variable introduction. It is polynomial-time for acyclic formulae, but is NP-hard with variable introduction. The open case is in the middle: not acyclic, but no new variables either.

Another open question is what extends from propositional logics to other logics where forgetting is applied: first-order logic [68, 101], description logics [33, 96] and modal logics [85, 98], where forgetting is often referred to as its dual concept of uniform interpolant, and also temporal logics [42], logics for reasoning about actions [37, 75], circumscription [92], defeasible logics [1] and abstract argumentation systems [5]. Answer set programming [47, 48] is very close to Horn logics, as clauses like $abc \rightarrow d$ resemble positive rules like `d <- a,b,c`. Yet, answer set programming significantly differs from propositional logic because of negation as failure. For example, a method for forgetting B from `a <- b` and `b <- c` produces the empty program [35] instead of `a <- c` like in propositional logic. The original program has the empty set as its only answer set: since no rule forces a variable to be true, they are all false. Removing B from this answer set leaves it empty, and the empty set is also the only answer set of the empty program for the same reason. This way of forgetting is therefore correct if forgetting is defined as removing variables from the answer sets. Other ways of forgetting instead produce `a <- c`, similarly to propositional logic [8, 58, 97].

In the other direction, different restrictions of the propositional formulae may simplify forgetting. Binary clauses form a common subclass with good computational properties. Other subclasses are in Post's lattice [21].

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10472-024-09955-5.

Author Contributions single author did it all.

Funding Open access funding provided by Università degli Studi di Roma La Sapienza within the CRUI-CARE Agreement. This article is supported by PRIN project “From 2D to 3D, Surgical and legal medical aspects” and by the SAPIENZA project “Riduzione di formule logiche”.

Availability of data and material (data transparency) no data is used in this article

Code availability (software application or custom code) code is freely available online at https://github.com/paololiberatore

Declarations

Conflicts of interest/Competing interests the author has no conflicts of interest to declare that are relevant to the content of this article

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Antoniou, G., Eiter, T., Wang, K.: Forgetting for defeasible logic. In Proceedings of the 18th conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-18), vol. 7180 of Lecture Notes in Computer Science, pp. 77–91. Springer (2012)
2. Atserias, A., Bonet, M.L.: On the automatizability of resolution and related propositional proof systems. Inf. Comput. **189**(2), 182–201 (2004)
3. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT competition 2016: Recent developments. In Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI 2017), pp. 5061–5063. AAAI Press/The MIT Press (2017)
4. Bash reference manual (2007)
5. Baumann, R., Gabbay, D.M., Rodrigues, O.: Forgetting an argument. In Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020), pp. 2750–2757. AAAI Press/The MIT Press (2020)
6. Baumgartner, R., Gottlob, G.: On the complexity of model checking for propositional default logics: New results and tractable cases. In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI’99), pp. 64–69 (1999)
7. Beeri, C., Dowd, M., Fagin, R., Statman, R.: On the structure of armstrong relations for functional dependencies. J. ACM **31**(1), 30–46 (1984)
8. Berthold, M., Gonçalves, R., Knorr, M., Leite, J.: A syntactic operator for forgetting that satisfies strong persistence. Theory Pract. Logic Program. **19**(5–6), 1038–1055 (2019)
9. Bhattacharya, A., DasGupta, B., Mubayi, D., Turán, G.: On approximate Horn formula minimization. In Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP 2010), pp. 438–450 (2010)
10. Bílková, M.: Uniform interpolation and propositional quantifiers in modal logics. Stud. Logica. **85**(1), 1–31 (2007)
11. Boole, G.: Investigation of The Laws of Thought. On Which Are Founded the Mathematical Theories of Logic and Probabilities, Walton and Maberly (1854)
12. Borchmann, D., Hanika, T., Obiedkov, S.: Probably approximately correct learning of Horn envelopes from queries. Computing Research Repository (CoRR), abs/1807.06149 (2018)
13. Borchmann, D., Hanika, T., Obiedkov, S.: Probably approximately correct learning of horn envelopes from queries. Discret. Appl. Math. **273**, 30–42 (2020)
14. Bubeck, U., Büning, H.K.: The power of auxiliary variables for propositional and quantified Boolean formulas. Stud. Log. **3**(3), 1–23 (2010)

15. Buchfuhrer, D., Umans, C.: The complexity of Boolean formula minimization. J. Comput. Syst. Sci. **77**(1), 142–153 (2011)
16. Buss, S.R.: Polynomial size proofs of the propositional pigeonhole principle. J. Symb. Log. **52**(4), 916–927 (1987)
17. Cadoli, M., Donini, F.M., Liberatore, P., Schaerf, M.: The size of a revised knowledge base. Artif. Intell. **115**(1), 25–64 (1999)
18. Cadoli, M., Donini, F.M., Liberatore, P., Schaerf, M.: Space efficiency of propositional knowledge representation formalisms. J. Artif. Intell. Res. **13**, 1–31 (2000)
19. Cadoli, M., Donini, F.M., Liberatore, P., Schaerf, M.: Preprocessing of intractable problems. Inf. Comput. **176**(2), 89–120 (2002)
20. Čepek, O., Kučera, P.: On the complexity of minimizing the number of literals in Horn formulae. RUTCOR Research Report RRR 11-208, Rutgers University (2008)
21. Chapdelaine, P., Hermann, M., Schnoor, I.: Complexity of default logic on generalized conjunctive queries. In Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), pp. 58–70. Springer (2007)
22. Coste-Marquis, S., Lang, J., Liberatore, P., Marquis, P.: Expressive power and succinctness of propositional languages for preference representation. In Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), pp. 203–212 (2004)
23. Coudert, O.: Two-level logic minimization: an overview. Integr. **17**(2), 97–140 (1994)
24. Coudert, O., Sasao, T.: Two-level logic minimization. In Logic Synthesis and Verification, pp. 1–27. Springer (2002)
25. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
26. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002)
27. Darwiche, A., Marquis, P., Suciu, D., Szeider, S.: Recent trends in knowledge compilation (dagstuhl seminar 17381). Dagstuhl Rep. **7**(9), 62–85 (2017)
28. Delgrande, J.P.: A knowledge level account of forgetting. J.Artif. Intell. Res. **60**, 1165–1213 (2017)
29. Delgrande, J.P., Wang, K.: A syntax-independent approach to forgetting in disjunctive logic programs. In Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), pp. 1482–1488. AAAI Press (2015)
30. Delgrande, J.P., Wassermann, R.: Horn clause contraction functions. J. Artif. Intell. Res. **48**, 475–511 (2013)
31. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In International conference on theory and applications of satisfiability testing, pp. 61–75 (2005)
32. Eiter, T., Gottlob, G.: On the complexity of propositional knowledge base revision, updates and counterfactuals. Artif. Intell. **57**, 227–270 (1992)
33. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H., Wang, K.: Forgetting in managing rules and ontologies. In 2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2006), pP. 411–419. IEEE Computer Society Press (2006)
34. Eiter, T., Kern-Isberner, G.: A brief survey on forgetting from a knowledge representation and perspective. KI — Kuenstliche Intelligenz, 33(1), 9–33 (2019)
35. Eiter, T., Wang, K.: Forgetting and conflict resolving in disjunctive logic programming. In Proceedings of the 21th National Conference on Artificial Intelligence (AAAI 2006), pp. 238–243 (2006)
36. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artif. Intell. **172**(14), 1644–1672 (2008)
37. Erdem, E., Ferraris, P.: Forgetting actions in domain descriptions. In Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007), pages 409–414. AAAI Press (2007)
38. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.: Reasoning about knowledge. The MIT Press (1995)
39. Fan, W., Ma, S., Hu, Y., Liu, J., Wu, Y.: Propagating functional dependencies with conditions. Proc. VLDB Endowment **1**(1), 391–407 (2008)
40. Fang, L., Liu, Y., van Ditmarsch, H.: Forgetting in multi-agent modal logics. Artif. Intell. **266**, 51–80 (2019)
41. Fang, L., Wan, H., Liu, X., Fang, B., Lai, Z.R.: Dependence in propositional logic: Formula-formula dependence and formula forgetting - Application to belief update and conservative extension. In Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018), pp. 1835–1844 (2018)
42. Feng, R., Acar, E., Schlobach, S., Wang, Y., Liu, W.: On sufficient and necessary conditions in bounded CTL. Technical Report abs/2003.06492, Computing Research Repository (CoRR) (2020)
43. Fermé, E.L., Hansson, S.O.: Belief Change - Introduction and Overview. Springer Briefs in Intelligent Systems. Springer (2018)

44. Fischer, P.C., Jou, J.H., Tsou, D.-M.: Succinctness in dependency systems. Theoret. Comput. Sci. **24**, 323–329 (1983)
45. Flögel, A., Kleine Büning, H., Lettmann, T.: On the restricted equivalence subclasses of propositional logic. Informatique Théorique et Applications, 27(4), 327–340 (1993)
46. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. J. Autom. Reason. **36**(4), 345 (2006)
47. Gonçalves, R., Knorr, M., Leite, J.: The ultimate guide to forgetting in answer set programming. In Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR 2016), pp. 135–144. AAAI Press/The MIT Press (2016)
48. Gonçalves, R., Knorr, M., Leite, J.: Forgetting in answer set programming - A survey. Theory Pract. Logic Program. **23**(1), 111–156 (2023)
49. Gonçalves, R., Knorr, M., Leite, J., Woltran, S.: When you must forget: Beyond strong persistence when forgetting in answer set programming. Theory Pract. Logic Program. **17**(5–6), 837–854 (2017)
50. Gonçalves, R., Knorr, M., Leite, J., Woltran, S.: On the limits of forgetting in Answer Set Programming. Artif. Intell. 286 (2020)
51. Gottlob, G.: Computing covers for embedded functional dependencies. In Proceedings of the 6th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'87), pp. 58–69. ACM (1987)
52. Hammer, P.L., Kogan, A.: Optimal compression of propositional Horn knowledge bases: Complexity and approximation. Artif. Intell. **64**(1), 131–145 (1993)
53. Hammer, P.L., Kogan, A.: Quasi-acyclic propositional Horn knowledge bases: Optimal compression. IEEE Trans. Knowl. Data Eng. **7**(5), 751–762 (1995)
54. He, Q., Ling, T.W.: Extending and inferring functional dependencies in schema transformation. In Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, 2004, pp. 12–21. ACM Press (2004)
55. Hemaspaandra, E., Schnoor, H.: Minimization for generalized Boolean formulas. In Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 566–571 (2011)
56. Henkin, L.: An extension of the craig-lyndon interpolation theorem. J. Symb. Log. **28**(3), 201–216 (1963)
57. Karnaugh, M.: The map method for synthesis of combinational logic circuits. Trans. Am. Inst. Electr. Eng. Part I Commun. Electron. **72**(5), 593–599 (1953)
58. Knorr, M., Alferes, J.J.: Preserving strong equivalence while forgetting. In Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014), vol. 8761, pp. 412–425. Springer (2014)
59. Konev, B., Walther, B., Wolter, F.: Forgetting and uniform interpolation in extensions of the description logic EL. In Proceedings of the 22nd International Workshop on Description Logics (DL 2009), vol. 9 (2009)
60. Koopmann, P., Schmidt, R.A.: Uniform interpolation of ALC ontologies using fixpoints. In Proceeding of the 9th International Workshop on Frontiers of Combining Systems (FroCoS 2014), vol. 8152 of Lecture Notes in Computer Science, pp. 87–102. Springer (2013)
61. Koopmann, P., Schmidt, R.A.: Uniform interpolation and forgetting for ALC ontologies with aboxes. In Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), pp. 175–181. AAAI Press/The MIT Press (2015)
62. Lang, J., Liberatore, P., Marquis, P.: Propositional independence – formula-variable independence and forgetting. J. Artif. Intell. Res. **18**, 391–443 (2003)
63. Lang, J., Marquis, P.: Reasoning under inconsistency: A forgetting-based approach. Artif. Intell. **174**(12–13), 799–823 (2010)
64. Liberatore, P.: Complexity issues in finding succinct solutions of PSPACE-complete problems. Technical Report abs/cs/0503043, CoRR (2005)
65. Liberatore, P.: The ghosts of forgotten things: A study on size after forgetting. Computing Research Repository (CoRR), abs/2005.04123 (2020)
66. Liberatore, P.: One head is better than two: a polynomial restriction for propositional definite horn forgetting. Computing Research Repository (CoRR), abs/2009.07497 (2020)
67. Liberatore, P.: Abductive forgetting. Technical Report abs/2209.12825, Computing Research Repository (CoRR) (2022)
68. Lin, F., Reiter, R.: Forget it! In Proceedings of the AAAI Fall Symposium on Relevance, pp. 154–159 (1994)
69. Ludwig, M., Konev, B.: Practical uniform interpolation and forgetting for ALC tboxes with applications to logical difference. In Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR 2014). AAAI Press/The MIT Press (2014)
70. McCluskey, E.J.: Minimization of Boolean functions. Bell Syst. Tech. J. **35**(6), 1417–1444 (1956)

71. Mengel, S., Wallon, R.: Graph width measures for CNF-encodings with auxiliary variables. J. Artif. Intell. Res. **67**, 409–436 (2020)
72. Moinard, Y.: Forgetting literals with varying propositional symbols. J. Log. Comput. **17**(5), 955–982 (2007)
73. Nakamura, K., Maruoka, S., Kimura, S., Watanabe, K.: Multi-clock path analysis using propositional satisfiability. In Proceedings of the 2000 Asia and South Pacific Design Automation Conference, pp. 81–86 (2000)
74. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). Artif. Intell. **274**, 224–248 (2019)
75. Rajaratnam, D., Levesque, H.J., Pagnucco, M., Thielscher, M.: Forgetting in action. In Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR 2014). AAAI Press (2014)
76. Rudell, R.L., Sangiovanni-Vincentelli, A.: Multiple-valued minimization for PLA optimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **6**(5), 727–750 (1987)
77. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. **4**(2), 177–192 (1970)
78. Smith, J.M., Maguire, G.Q., Jr.: Effects of copy-on-write memory management on the response time of UNIX fork operations. Comput. Syst. **1**(3), 255–278 (1988)
79. Stockmeyer, L.J.: The polynomial-time hierarchy. Theoret. Comput. Sci. **3**, 1–22 (1976)
80. Subbarayan, S., Pradhan, D.K.: NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In International conference on theory and applications of satisfiability testing, pp. 76–291. Springer (2004)
81. Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM J. Comput. **2**(3), 211–216 (1973)
82. Theobald, M., Nowick, S.M., Wu, T.: Espresso-HF: a heuristic hazard-free minimizer for two-level logic. In Proceedings of the 33rd Design Automation Conference, pp. 71–76 (1996)
83. Umans, C.: The minimum equivalent DNF problem and shortest implicants. J. Comput. Syst. Sci. **63**(4), 597–611 (2001)
84. Umans, C., Villa, T., Sangiovanni-Vincentelli, A.L.: Complexity of two-level logic minimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **25**(7), 1230–1246 (2006)
85. van Ditmarsh, H., Herzig, A., Lang, J., Marquis, P.: Introspective forgetting. Synthese **169**, 809–827 (2009)
86. Van Rossum, G., Drake, F.L.: The Python language reference manual. Network Theory Ltd (2011)
87. Veitch, E.W.: A chart method for simplifying truth functions. In Proceedings of the 1952 ACM national meeting (Pittsburgh), pp. 127–133 (1952)
88. Wang, A., Wang, K., Topor, R.W., Pan, J.Z.: Forgetting concepts in DL-lite. In Proceedings of the 5th European Semantic Web Conference, ESWC-2008, vol. 5021 of Lecture Notes in Computer Science, pp. 245–257. Springer (2008)
89. Wang, K., Sattar, A., Su, K.: A theory of forgetting in logic programming. In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 682–688. AAAI Press/The MIT Press (2005)
90. Wang, Q., Wen, X.: Propagating dependencies under schema mappings: A graph-based approach. In Proceedings of the 19th International Database Engineering & Applications Symposium, Yokohama, Japan, July 13-15, 2015, pp. 126–135. ACM Press (2015)
91. Wang, Y.: On forgetting in tractable propositional fragments. Technical Report 1502.02799, Computing Research Repository (CoRR) (2015)
92. Wang, Y., Wang, K., Wang, Z., Zhuang, Z.: Knowledge forgetting in circumscription: A preliminary report. In Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), pp. 1649–1655. AAAI Press/The MIT Press (2015)
93. Wang, Y., Zhang, Y., Zhou, Y., Zhang, M.: Knowledge forgetting in answer set programming. J. Artif. Intell. Res. **50**, 31–70 (2014)
94. Wang, Z., Wang, K., Topor, R.W., Pan, J.Z.: Forgetting for knowledge bases in DL-lite. Ann. Math. Artif. Intell. **58**(1–2), 117–151 (2010)
95. Zengler, C., Küchlin, W.: Encoding the Linux kernel configuration in propositional logic. In Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010), Workshop on Configuration, vol. 2010, pp. 51–56 (2010)
96. Zhang, X.: Forgetting for distance-based reasoning and repair in DL-lite. Knowl.-Based Syst. **107**, 246–260 (2016)
97. Zhang, Y., Foo, N.Y.: Solving logic program conflict through strong and weak forgettings. Artif. Intell. **170**(8–9), 739–778 (2006)

98. Zhang, Y., Zhou, Y.: Knowledge forgetting: properties and applications. Artif. Intell. **173**, 1525–1537 (2009)
99. Zhao, Y., Schmidt, R.A., Wang, Y., Zhang, X., Feng, H.: A practical approach to forgetting in description logics with nominals. In Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020), pp. 3073–3079 (2020)
100. Zhou, Y.: Polynomially bounded forgetting. In Proceedings of the 13th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2014), pp. 422–434 (2014)
101. Zhou, Y., Zhang, Y.: Bounded forgetting. In Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI 2011). AAAI Press (2011)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.