FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

A Comparison of Methods and Operators for Multiple Variable Forgetting

Bachelor's Thesis

in partial fulfilment of the requirements for
the degree of *Bachelor of Science (B.Sc.)*
in Informatik

submitted by
Florian Lerch

First examiner:	Prof. Dr. Kai Sauerwald Artificial Intelligence Group
Advisor:	Prof. Dr. Kai Sauerwald Artificial Intelligence Group

Statement

I declare that I have written the bachelor´s thesis independently and without unauthorized use of third parties. I have only used the indicated resources and I have clearly marked the passages taken verbatim or in the sense of these resources as such. The assurance of independent work also applies to any drawings, sketches or graphical representations. The work has not previously been submitted in the same or similar form to the same or another examination authority and has not been published. By submitting the electronic version of the final version of the bachelor´s thesis, I acknowledge that it will be checked by a plagiarism detection service to check for plagiarism and that it will be stored exclusively for examination purposes.

I explicitly agree to have this thesis published on the webpage of the artificial intelligence group and endorse its public availability.

Software created for this work has been made available as open source; a corresponding link to the sources is included in this work. The same applies to any research data.

Hochheim am Main, 14.10.2025 Florian Lerch

(Place, Date) (Signature)

Zusammenfassung

Variablenvergessen in KI spiegelt die menschliche Fähigkeit wieder, bekannte Informationen temporär oder permanent auszublenden. Die bekannteste Methode basiert auf der Arbeit von Lin und Reiter aus 1994 und wurde für das Entfernen einzelner Variablen und für das Bestimmen der Relevanz einer Variable in einer logischen Formel konzipiert. Beim Vergessen mehrerer Variablen hintereinander mit dieser Methode wird aber schnell auf Probleme der Skalierbarkeit gestoßen, die diese Arbeit aufgezeigt werden und mögliche alternativen präsentiert werden. Variablenvergessen in KI spiegelt die menschliche Fähigkeit wieder, vorhandenes Wissen vorübergehend oder dauerhaft zu ignorieren. Die gängigste Methode basiert auf der Arbeit von Lin und Reiter aus dem Jahr 1994 und wurde entwickelt, um eine kleine Anzahl von Variablen zu entfernen und die Relevanz einer Variable in einer logischen Formel zu bestimmen. Beim sukzessiven Vergessen mehrerer Variablen treten jedoch schnell Skalierbarkeitsprobleme auf. Diese Arbeit zeigt diese Probleme in der klassischen Definition des Variablenvergessens auf, stellt mögliche alternative Ansätze vor und implementiert sie in einem grafischen Tool, um die weitere Forschung zum Vergessen mehrerer Variablen zu unterstützen.

Abstract

Variable forgetting in AI mirrors the human ability to temporarily or permanently ignore knowledge. The most common method is based on the work of Lin and Reiter from 1994 and was designed for the removal of small numbers of variables and for determining the relevance of a variable in a logical formula. However, when forgetting multiple variables in a row using this method, scalability issues quickly become apparent. This work showcases these issues in the classic definition of variable forgetting, presents possible alternative approaches and implements them in a graphical tool to support further research into multiple variable forgetting.

Contents

1 Introduction

Within the field of knowledge representation in artificial intelligence, an agent is a system that autonomously gathers information through observations from its environment and uses it to derive logical conclusions. While an agent can be human, the term usually refers to robotic AI systems. The knowledge of an agent consists of the observations together with the respective logical conclusions and is kept in the agent's knowledge base. An agent modifies its knowledge base through the fundamental actions learning and forgetting. Simplified, learning means adding logical statements to the knowledge base, and forgetting is the removal of statements from the knowledge base. Since the knowledge base always contains all possible logical consequences of the contained information, a single observation can lead to the addition of many logical statements to the knowledge base.

As an agent accumulates more and more knowledge, the amount of information to be considered when making decisions also increases. However, not all available information is necessary to make decisions in every case. For example, if an agent was to determine whether an observed object is a bird, the current weather should not be a factor. Humans can simply ignore redundant information for decision making without permanently forgetting the respective knowledge, but AI agents cannot simply restore forgotten information. If the knowledge of the agent is represented using some logical system, the agent can temporarily or permanently remove the respective variables from their formulas using a method called variable forgetting.

The most common method for variable forgetting is based on the work of Lin and Reiter from 1994 [6] and subsequent research by Lang et al. in 2003 [4]. However, this method quickly runs into complexity issues when forgetting multiple variables successively. While Lin and Reiters' method was transferred to many different logical systems, the underlying core methodology remains the same, which means that the concept of multiple variable forgetting has not yet been explored.

The goal of this work is to explore possible approaches for multiple variable forgetting, to characterize and compare the approaches, and finally to implement them in a simple tool to enable further experimentation.

This work is structured in 6 sections. After introducing the topic and its theoretical background, we will explore the terms and concepts which will be used throughout this work. Building upon this, we define some approaches for variable forgetting and illustrate them with some examples. We then show how these approaches can be applied to sets of variables and demonstrate the approaches for multiple variable forgetting again using some examples. In section 5 we present the implementation of the approaches. Finally, we conclude with an evaluation and future improvements.

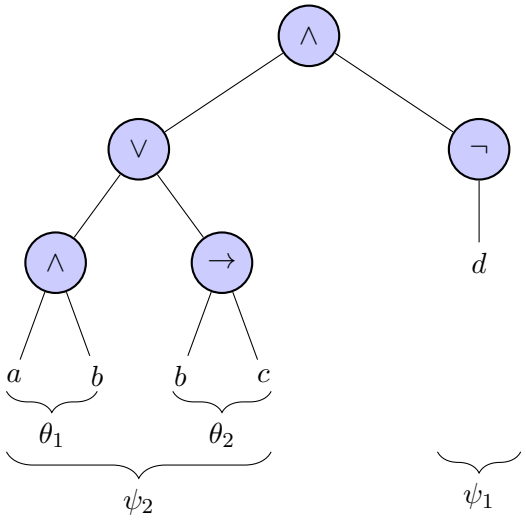

Figure 1: Syntax tree of $\varphi_{st} = ((a \wedge b) \vee (b \rightarrow c)) \wedge \neg d$. The subformulas $\psi_1 = \neg d$ and $\psi_2 = (a \wedge b) \vee (b \rightarrow c)$ are highlighted, where ψ_2 further decomposes into $\theta_1 = (a \wedge b)$ and $\theta_2 = (b \rightarrow c)$.

2 Preliminaries

2.1 Propositional Logic

Although Variable Forgetting extends to other logical systems [2], this work focuses on propositional logic to keep the treatment of multiple variable forgetting and its implementation tractable.

In this work, propositional logic operates on an alphabet $\Sigma_{\{\top,\bot\}}$, which consists of a set of propositional variables Σ and the set of boolean truth values BOOL$= \{\top, \bot\}$. A propositional formula is constructed using elements of $\Sigma_{\{\top,\bot\}}$ in concatenation with parentheses, the set of binary logical operators OP$= \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ and the unary logical operator $\neg$.

We call the set of all variables that occur in a formula φ the $Signature$ of the formula, written $Sig(\varphi)$. A $literal$ is an occurrence of a variable in its negated or non-negated form. We write $Lit(\varphi)$ for the set of all literals that occur in a formula φ. For example, for the formula $\varphi = ((a \wedge b) \vee (\neg b \rightarrow c)) \wedge \neg d$, it is $Sig(\varphi) = \{a, b, c, d\}$ and $Lit(\varphi) = \{a, b, \neg b, c, \neg d\}$.

A formula is in negation normal form (NNF), if the only binary operators in the formula are conjunction and disjunction, and negation operators only occur directly in front of variables. We call a formula an *atomic formula*, if it only consists of a single literal.

We can represent a formula as a *syntax tree*. In a syntax tree, the leaf nodes (and only the leaf nodes) contain the variables of the formula and the inter-

nal nodes contain the operators, specifically $\neg$ and elements of OP. As an example, the syntax tree of the formula $\varphi_{st} = (a \wedge b \vee b \rightarrow c) \wedge \neg d$ is presented in Figure 1. The figure also highlights the *subformulas* $\psi_1 = \neg d$ and $\psi_2 = a \wedge b \vee b \rightarrow c$, which itself split into the subformulas $\theta_1 = a \wedge b$ and $\theta_2 = b \rightarrow c$.

Variables in the leaf nodes are also atomic formulas, but we represent them simply as variables to focus on the compound subformulas. Each subformula has a signature that is a subset of its respective parent formula's signature. In this example, $Sig(\psi_1) \subseteq Sig(\varphi)$, $Sig(\psi_2) \subseteq Sig(\varphi)$, $Sig(\theta_1) \subseteq Sig(\psi_2)$, and $Sig(\theta_2) \subseteq Sig(\psi_2)$.

A model ω over an alphabet Σ is an assignment for all variables $v \in \Sigma$ to $\top$ or $\bot$. We call Ω_Σ the set of all models over Σ. If the evaluation of a formula φ yields $\top$ under the assignment ω, we say φ holds for ω, denoted as $\omega \models \varphi$. We write $Mod(\varphi)$ for the set of all models of φ. We represent models as sequences of variables, where $\overline{v}$ indicates that variable v is assigned $\bot$. For example, let $\varphi = a \vee (b \wedge \neg c)$, then $Mod(\varphi) = \{abc, ab\overline{c}, a\overline{b}c, a\overline{bc}, \overline{a}b\overline{c}\}$.

2.2 Variable Forgetting

Variable Forgetting is the process of removing all occurrences of a variable from a formula and its models, effectively removing the variable from the signature of the formula without a replacement. Syntactically, this is usually achieved by substituting the variable that we try to forget from the formula with truth values.

Definition 1 (Substitution) *For an expression (a variable or a formula) x, a formula φ and a replacement (variable, formula or truth value) v, the substitution of all occurrences of x in φ with v is denoted as $\varphi[x/v]$.*
For a set of expressions X, the substitution of all occurrences of all elements of X

Example 1 (Substitution) *Let $\varphi = ((a \wedge b) \rightarrow c) \vee (a \wedge b \wedge c)$, $S = \{a, c\}$ and $F = \{a \wedge b\}$ We can form different substitutions as follows:*

$$\varphi[a/\top] = ((\top \wedge b) \rightarrow c) \vee (\top \wedge b \wedge c) \quad (1)$$
$$\varphi[S/d] = ((d \wedge b) \rightarrow d) \vee (d \wedge b \wedge d) \quad (2)$$
$$\varphi[F/(a \vee b)] = ((a \vee b) \rightarrow c) \vee (a \vee b \wedge c) \quad (3)$$

Since Variable Forgetting is ultimately a optimization method, we require terms to determine the effectiveness of the different methods.

Definition 2 *Propositional Logic complexity statistics We define $size$ as the amount of variable occurrences within a formula and $depth$ as the maximum depth of the syntax tree.*

- $size(\varphi) := |Lit(\varphi)|$ *where* $Lit(\varphi)$ *is the multiset of all literal occurrences in* φ*.*
- $depth(\varphi) := \max\{d(n) \mid n$ *is a leaf node in* $T(\varphi)\}$ *where* $d(n)$ *denotes the distance from the root to note* n *in the syntax tree* $T(\varphi)$

3 Variable Forgetting operators

The operators for variable forgetting we will introduce in this chapter can all operate using sets of variables; however, for an introduction we will focus on how they remove single variables first.

3.1 Classic Variable Forgetting

This method of variable forgetting was introduced by Lin and Reiter in 1994 [6] and is the basis for most other variable forgetting methods. The key idea behind it is to consider both possibilities for the assignment of a truth value to a variable, so that the actual value is irrelevant in the result.

Definition 3 (Classic Variable Forgetting $fgClassic$) *For a formula φ and propositional variable v, classic variable Forgetting is defined as*
$fgClassic(\varphi, v) = \varphi[x/\top] \vee \varphi[x/\bot]$

Example 2 *Let $\varphi_1 = b \vee (a \wedge c) \vee (\neg a \wedge c)$, $\varphi_2 = a \wedge b \wedge c$ and $\varphi_3 = (a \vee b) \wedge c$. To forget a in each of the formulas, we derive:*

$$\begin{aligned}
fgClassic(\varphi_1, a) &= (b \vee (a \wedge c) \vee (\neg a \wedge c))[a/\top] \vee (b \vee (a \wedge c) \vee (\neg a \wedge c))[a/\bot] \\
&= (b \vee (\top \wedge c) \vee (\neg \top \wedge c)) \vee (b \vee (\bot \wedge c) \vee (\neg \bot \wedge c)) \\
&= (b \vee c \vee \bot) \vee (b \vee \bot \vee c) \\
&= b \vee c \\
fgClassic(\varphi_2, a) &= (\top \wedge b \wedge c) \vee (\bot \wedge b \wedge c) \\
&= b \wedge c \\
fgClassic(\varphi_3, a) &= ((\top \vee b) \wedge c) \vee ((\bot \vee b) \wedge c) \\
&= c \vee (b \wedge c) \\
&= c
\end{aligned}$$

The evaluations of φ_1, φ_2 and φ_3 and the respective results of forgetting a from the formulas are presented in Table 1.

We observe that for every formula φ, $fgClassic(\varphi)$ has double the length of the original formula and depth $depth(\varphi) + 1$. In the truth table 1, we can see that in φ_1, the removal of a has no semantic effect since $Mod_\Sigma(\varphi_1) = Mod_\Sigma(forget(\varphi_1, a))$. $forget(\varphi_3, a)$ shows that, depending on the syntactical structure of the formula, variable forgetting can also result in the removal of additional variables from the formula. These notions of *formula-variable dependency* and *variable interdependency* are strongly related to variable forgetting and are further explored in [4] and other related works.

Variables			Example 1		Example 2		Example 3	
a	b	c	φ_1	$fgClassic(\varphi_1, a)$	φ_2	$fgClassic(\varphi_2, a)$	φ_3	$fgClassic(\varphi_3, a)$
0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	1
0	1	0	1	1	0	0	0	0
0	1	1	1	1	0	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	1	1
1	1	0	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1

Table 1: Truth table evaluation of variable forgetting examples. Green cells indicate preserved truth values, gray cells show false evaluations, and red cells highlight changes in truth value after forgetting.

Based only on the definition, we can see some basic properties of classic variable forgetting using a formula φ and variable v:

$$\begin{aligned}
\text{Variable Elimination} \quad & \forall v \in Sig(\varphi) : v \notin Sig(fgClassic(\varphi, v)) \\
\text{Invariance} \quad & \forall v \notin Sig(\varphi) : fgClassic(\varphi, v) = \varphi \\
\text{Idempotency} \quad & \forall v \in Sig(\varphi) : fgClassic(fgClassic(\varphi, v), v) \\
& \qquad = fgClassic(\varphi, v) \\
\text{Weakness} \quad & \forall v \in Sig(\varphi) : \varphi \models fgClassic(\varphi, v) \\
\text{Model expansion} \quad & \forall v \in \Sigma : Mod_\Sigma(\varphi) \subseteq Mod_\Sigma(\varphi, v)
\end{aligned}$$

As expected, a forgotten variable is removed from the signature of the formula (variable elimination) and attempting to forget a variable that is not part of the formula produces the same unmodified formula (invariance). Accordingly, forgetting the same variable multiple times has no further effect beyond the first iteration (idempotency). Semantically, a formula entails any formula that results from forgetting a variable from the formula using classic variable forgetting (weakness), but the resulting formula may have more models than the original formula (model expansion).

Formally, for a formula φ over alphabet Σ and $v \in \Sigma$, let $\Gamma = \Sigma \backslash \{v\}$. For any model $\omega \in Mod(\varphi)$, let ω^Γ denote the Γ-part of ω (the model ω with variable v removed). Then $\{\omega^\Gamma \mid \omega \in Mod(\varphi)\} \subseteq Mod(fgClassic(\varphi, v))$.

We can also determine the distributivity of conjunction and disjunction for classic variable forgetting:

Let $\varphi_1 = \psi \vee \theta$. Then

$$\begin{aligned} fgClassic(\varphi_1, v) &= fgClassic(\psi \vee \theta, v) \\ &= (\psi \vee \theta)[v/\top] \vee (\psi \vee \theta)[v/\bot] \\ &= \psi[v/\top] \vee \theta[v/\top] \vee \psi[v/\bot] \vee \theta[v/\bot] \\ &= \psi[v/\top] \vee \psi[v/\bot] \vee \theta[v/\top] \vee \theta[v/\bot] \\ &= fgClassic(\psi, v) \vee fgClassic(\theta, v). \end{aligned}$$

However, this does not hold for conjunction. Let $\varphi_2 = \psi \wedge \theta$. Then

$$\begin{aligned} fgClassic(\varphi_2, v) &= fgClassic(\psi \wedge \theta, v) \\ &= (\psi \wedge \theta)[v/\top] \vee (\psi \wedge \theta)[v/\bot] \\ &= \psi[v/\top] \wedge \theta[v/\top] \vee \psi[v/\bot] \wedge \theta[v/\bot] \\ &\neq fgClassic(\psi, v) \wedge fgClassic(\theta, v) \end{aligned}$$

Therefore, we observe

$$\begin{aligned} &fgClassic(\varphi \vee \psi, v) = fgClassic(\varphi, v) \vee fgClassic(\psi, v) \text{ and} \\ &fgClassic(\varphi \wedge \psi, v) \neq fgClassic(\varphi, v) \wedge fgClassic(\psi, v). \end{aligned}$$

$fgClassic$ is a combination of substitution of all occurrences of a variable in a formula and duplication of the formula. As both of these operations are of linear complexity, a single application of $fgClassic$ also exhibits linear complexity. However, forgetting multiple variables by successive application of $fgClassic$ results in exponential complexity, since the length of the resulting formula doubles with each forgotten variable. We will examine this in more detail in Section 4.

3.2 Truth value assumption

We can forget a variable by assuming the truth value of the variable for all models of a formula. Syntactically, this equals the substitution of the variable with a truth value.

Definition 4 (Truth value assumption $forgetTrue(\varphi, v)/forgetFalse(\varphi, v)$) *For a variable v and formula φ, we define $forgetTrue(\varphi, v) = \varphi[x/\top]$ and $forgetFalse(\varphi, v) = \varphi[x/\bot]$.*

Variables			Truth Value Assumption Examples			
a	b	c	φ	$forgetFalse(\varphi, a)$	$forgetTrue(\varphi, b)$	$forgetTrue(\varphi, c)$
0	0	0	0	0	1	0
0	0	1	0	0	1	0
0	1	0	1	1	1	0
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	1	1	0	1	0
1	1	0	0	1	0	0
1	1	1	1	1	1	1

Table 2: Truth table for example 3. Green cells indicate preserved truth values, gray cells show false evaluations for both φ and the result, and red cells highlight where the truth value changes after forgetting, demonstrating information loss or falsification.

Example 3 *Let* $\varphi = (\neg a \wedge b) \vee (a \wedge c)$*. Then*

$$\begin{aligned} forgetFalse(\varphi, a) &= (\neg\bot \wedge b) \vee (\bot \wedge c) \\ &= b \\ forgetTrue(\varphi, b) &= (\neg a \wedge \top) \vee (a \wedge c) \\ &= \neg a \vee (a \wedge c) \\ forgetTrue(\varphi, c) &= (\neg a \wedge b) \vee (a \wedge \top) \\ &= (\neg a \wedge b) \vee a \end{aligned}$$

The evaluations of φ *and the respective results of forgetting variables using truth value assumption can be seen in the truth table Table 2.*

Truth value assumption is a trivialization of classic variable forgetting. Replacing all occurrences of a variable in a formula has strict linear complexity, regardless of whether the variable is in the signature of the formula or not. As already stated, the basic properties of variable forgetting all hold for truth value assumption as well. The distribution over conjunction and disjunction can also be shown trivially.

The semantic trade-off for the reduced syntactic complexity is a greater loss of information in the resulting formula. Although we remove a single variable and achieve the same loss of semantic complexity as $fgClassic$, truth value assumption neglects to consider a possible truth value of the forgotten variable. Consequently, we lose all information about the models where the variable has the opposite value. For example, consider $forgetTrue(\varphi, b)$ from Table 2. Although $ab\bar{c}$ is a model of φ, the corresponding assignment

$a\overline{c}$ is not a model of $forgetTrue(\varphi, b)$, demonstrating information loss. Conversely, $forgetTrue(\varphi, b)$ includes the models $\overline{a}\overline{c}$ and $\overline{a}c$, whereas the corresponding assignments $\overline{ab}\overline{c}$ and $\overline{ab}c$ are not models of φ. Both types of discrepancies are shown as red cells in the truth table, demonstrating how truth value assumption can not only lose valid information but also introduce falsifications by accepting assignments that should evaluate to false.

Additionally, truth value assumption increases the risk of unintentionally removing additional variables. For example, if we have $\varphi = a \wedge b$ and forget a by assuming $a = \bot$, we get $\bot \wedge b \equiv \bot$, thus all information about b is lost as well, whereas $fgClassic(\varphi, a) = (\top \wedge b) \vee (\bot \wedge b) = b$ preserves the relevance of b.

For these reasons, truth value assumption is best applied when the truth value of a variable is constant throughout all models of a formula.

3.3 Skeptical Variable Forgetting

Skeptical variable forgetting is a variant of classic variable forgetting that aims to minimize truth values. The approach was explored by Cristoph Kaplan in [3].

Definition 5 (Skeptical Variable Forgetting $fgSkep$) *For a formula φ and a propositional variable v, skeptical variable forgetting is defined as $fgSkep(\varphi, v) = \varphi[x/\top] \wedge \varphi[x/\bot]$.*

Example 4 *Let $\varphi_1 = b \vee (a \wedge c) \vee (\neg a \wedge c)$, $\varphi_2 = a \wedge b \wedge c$ and $\varphi_3 = (a \vee b) \wedge c$. Forgetting a from each formula yields:*

$$\begin{aligned}
fgSkep(\varphi_1, a) &= (b \vee (a \wedge c) \vee (\neg a \wedge c))[a/\top] \wedge (b \vee (a \wedge c) \vee (\neg a \wedge c))[a/\bot] \\
&= (b \vee (\top \wedge c) \vee (\neg\top \wedge c)) \wedge (b \vee (\bot \wedge c) \vee (\neg\bot \wedge c)) \\
&= (b \vee c \vee \bot) \wedge (b \vee \bot \vee c) \\
&= b \vee c \\
fgSkep(\varphi_2, a) &= (\top \wedge b \wedge c) \wedge (\bot \wedge b \wedge c) \\
&= \bot \\
fgSkep(\varphi_3, a) &= ((\top \vee b) \wedge c) \wedge ((\bot \vee b) \wedge c) \\
&= c \wedge (b \wedge c) \\
&= c \wedge b
\end{aligned}$$

The evaluations of φ_1, φ_2 and φ_3 and the respective results of removing a from the formulas using skeptical forgetting are shown in Table 3.

Due to their syntactic similarity, $fgClassic$ and $fgSkep$ show similar properties that depend on their syntax. Variable elimination, invariance and

Variables			Example 1		Example 2		Example 3	
a	b	c	φ_1	$fgSkep(\varphi_1, a)$	φ_2	$fgSkep(\varphi_2, a)$	φ_3	$fgSkep(\varphi_3, a)$
0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0
0	1	0	1	1	0	0	0	0
0	1	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	1	0
1	1	0	1	1	0	0	0	0
1	1	1	1	1	1	0	1	1

Table 3: Truth table evaluation of skeptical variable forgetting examples. Green cells indicate preserved truth values, gray cells show false evaluations, and red cells highlight changes in truth value after forgetting.

weakness also hold for $fgSkep$ and forgetting a variable using skeptical variable forgetting also results in a formula with double the length of the original formula. Distributivity over $\wedge$ and $\vee$ is inverted for $fgSkep$ compared to $fgClassic$, formally $fgSkep(\varphi \vee \psi, v) \neq fgSkep(\varphi, v) \vee fgSkep(\psi, v)$ and $fgSkep(\varphi \wedge \psi, v) = fgSkep(\varphi, v) \wedge fgSkep(\psi, v)$.

For negation, Kaplan [3] establishes the following relation between $fgClassic$ and $fgSkep$:

$$\begin{aligned} fgClassic(\neg\varphi, x) &= (\neg\varphi)[x/\top] \vee (\neg\varphi)[x/\bot] \\ &= \neg(\varphi[x/\top] \wedge \varphi[x/\bot]) \\ &= \neg fgSkep(\varphi, x) \\ \neg fgClassic(\varphi, x) &= \neg(\varphi[x/\top] \vee \varphi[x/\bot]) \\ &= \neg(\varphi[x/\top]) \wedge \neg(\varphi[x/\bot]) \\ &= fgSkep(\neg\varphi, x) \end{aligned}$$

The semantic properties weakness and model expansion are also inverted for $fgSkep$ compared to $fgClassic$. Whereas $fgClassic$ tends to include more models and the original formula entails the result of forgetting, $fgSkep$ tends to falsify models and the result entails the original formula [3].

3.4 Recursive Classic Variable Forgetting

A key disadvantage of $fgClassic$ is the doubled length of the result formula. We can minimize the additional syntactic complexity by minimizing the affected portion of the formula by exploiting the distributivity of $fgClassic$. Depending on the occurrence of the variable we are trying to forget in the signatures of the subformulas and the operator with the highest precedence,

we can forget from the deepest possible subformula instead of the whole formula. More specifically, for a formula in the form of $\varphi = \psi \wedge \theta$, if the variable we are trying to forget occurs in the signature of only one of ψ or θ, we only need to forget from the relevant part of the formula.

Formally, for a formula $\varphi_1 = \psi \wedge \theta$, with $x \notin Sig(\psi)$ and $x \in Sig(\theta)$, we have

$$\begin{aligned} fgClassic(\varphi_1, x) &= fgClassic(\psi \wedge \theta, x) \\ &= (\psi \wedge \theta)[x/\top] \vee (\psi \wedge \theta)[x/\bot] \\ &= (\psi \wedge \theta[x/\top]) \vee (\psi \wedge \theta[x/\bot]) \\ &= \psi \wedge (\theta[x/\top] \vee \theta[x/\bot]) \\ &= \psi \wedge fgClassic(\theta, x) \end{aligned}$$

We can combine this with the distributivity of $fgClassic$ over $\vee$ to minimize the number of affected subformuals.

Definition 6 (Recursive classic variable forgetting $fgClRec$) *For any formulas $varphi_1 = v$, $\varphi_2 = \neg\psi$, $\varphi_3 = \psi \vee \theta$, $\varphi_4 = \psi \wedge \theta$ and variables $v, x \in \Sigma$ (with not necessarily $v \neq x$), we define*
$fgClRec(\varphi_1, x) = fgClRec(v, x) = fgClassic(v, x)$,
$fgClRec(\varphi_2, x) = fgClRec(\neg\psi, x) = fgClassic(\neg\psi, x)$,
$fgClRec(\varphi_3, x) = fgClRec(\psi, x) \vee fgClRec(\theta, x)$ *and*
$fgClRec(\varphi_4, x) = fgClRec(\psi \wedge \theta, x)$

$$= \begin{cases} fgClassic(\varphi_4, x), & iff\ x \in Sig(\psi)\ and\ x \in Sig(\theta) \\ \psi \wedge fgClRec(\theta, x), & iff\ x \notin Sig(\psi)\ and\ x \in Sig(\theta) \\ fgClRec(\psi, x) \wedge \theta, & iff\ x \in Sig(\psi)\ and\ x \notin Sig(\theta) \\ \varphi_4, & iff\ x \notin Sig(\psi)\ and\ x \notin Sig(\theta) \end{cases}$$

Example 5 *Let $\varphi = (a \wedge b \vee a \wedge c) \wedge (b \oplus c)$ and $\theta = \neg(a \vee \neg b) \vee (a \wedge c) \vee (b \wedge c)$.*

Then

$$
\begin{aligned}
fgClassic(\varphi, a) &= ((a \wedge b \vee a \wedge c) \wedge (b \oplus c))[a/\top] \vee ((a \wedge b \vee a \wedge c) \wedge (b \oplus c))[a/\bot] \\
&= ((\top \wedge b \vee \top \wedge c) \wedge (b \oplus c)) \vee ((\bot \wedge b \vee \bot \wedge c) \wedge (b \oplus c)) \\
&= ((b \vee c) \wedge (b \oplus c)) \vee ((\bot \vee \bot) \wedge (b \oplus c)) \\
&= b \oplus c \\
fgClRec(\varphi, a) &= fgClRec((a \wedge b \vee a \wedge c), a) \wedge (b \oplus c) \\
&= (fgClRec((a \wedge b), a) \vee fgClRec(a \wedge c), a)) \wedge (b \oplus c) \\
&= (fgClRec(a, a) \wedge b \vee fgClRec(a, a) \wedge c) \wedge (b \oplus c) \\
&= (\top \wedge b \vee \top \wedge c) \wedge (b \oplus c) \\
&= (b \vee c) \wedge (b \oplus c) \\
&= b \oplus c \\
fgClassic(\theta, a) &= (\neg(a \vee \neg b) \vee (a \wedge c) \\
&\quad \vee (b \wedge c))[a/\top] \vee (\neg(a \vee \neg b) \vee (a \wedge c) \vee (b \wedge c))[a/\bot] \\
&= (\neg(\top \vee \neg b) \vee (\top \wedge c) \vee (b \wedge c)) \vee (\neg(\bot \vee \neg b) \vee (\bot \wedge c) \vee (b \wedge c)) \\
&= (\neg\top \vee c \vee (b \wedge c)) \vee (\neg(\neg b) \vee \bot \vee (b \wedge c)) \\
&= (c \vee (b \wedge c)) \vee (b \vee (b \wedge c)) \\
&= c \vee b \\
fgClRec(\theta, a) &= fgClRec(\neg(a \vee \neg b), a) \vee fgClRec((a \wedge c), a) \\
&\quad \vee fgClRec((b \wedge c), a) \\
&= ((\neg(a \vee \neg b))[a/\top] \vee (\neg(a \vee \neg b))[a/\bot]) \\
&\quad \vee (fgClRec(a, a) \wedge c) \vee (b \wedge c) \\
&= (\neg(\top \vee \neg b)) \vee (\neg(\bot \vee \neg b) \vee (\top \wedge c) \vee (b \wedge c) \\
&= \bot \vee b \vee c \vee (b \wedge c) \\
&= b \vee c
\end{aligned}
$$

The evaluations of φ, θ, and their results after forgetting a are shown in Table 4.

$fgClRec$ is a result from restructuring $fgClassic$. As such, all properties of $fgClassic$ also hold here, and both methods achieve the same semantic result. A distinct advantage of $fgClRec$ over $fgClassic$ is of course the reduced length of the immediate result after forgetting a variable, with the potential difference in the length of the result increasing further with the length of the original formula. However, this requires more calculation time for the signatures of the subformulas with each recursive execution of $fgClRec$. Although the signature of a formula can be constructed in linear time, deeper subformulas will be analyzed multiple times for one execution of $fgClassic$, further increasing the complexity of the operation.

Variables			Example 1			Example 2		
a	b	c	φ	$fgClassic(\varphi, a)$	$fgClRec(\varphi, a)$	θ	$fgClassic(\theta, a)$	$fgClRec(\theta, a)$
0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	1	1	1
0	1	1	0	0	0	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

Table 4: Truth table evaluation of recursive classical variable forgetting examples. Green cells indicate preserved truth values, gray cells show false evaluations, and red cells highlight changes in truth value after forgetting.

3.5 Irrelevance assumption

Looking at the syntax tree of a formula, we could remove a variable by removing its respective leaf nodes and replacing the operator parent node with the remaining variable. We can achieve this formally by replacing the variable depending on the operator it is bound to, respecting negations, to remove it from the formula without impacting the formula on the other side of the operator. For example, to forget a in a formula $(\neg a \wedge b) \vee (a \wedge c)$, we would replace a with $\bot$ in the subformula $\neg a \wedge b$ to receive $\neg\bot \wedge b = b$, and we would replace it with $\top$ in the other subformula to receive c.

However, forgetting a variable from atomic formulas or formulas using the logical equivalence operator $\Leftrightarrow$ using the method we just described poses a problem. A formula $a \Leftrightarrow b$ has the two possible interpretations $(a \wedge b) \vee (\neg a \wedge \neg b)$ and $(a \vee \neg b) \wedge (\neg a \vee b)$. Forgetting a by assuming irrelevancy would result in $(\top \wedge b) \vee (\neg\bot \wedge \neg b) = b \vee \neg b = \top$ and $(\bot \vee \neg b) \wedge (\neg\top \vee b) = \neg b \wedge b = \bot$, a contradiction that cannot be solved using only propositional logic. Furthermore, we can not simply define irrelevancy on an atomic formula.

To avoid these issues, we introduce an *irrelevance placeholder* χ and define how the operators of propositional logic interact with χ. Our variable forgetting function would then require two steps: First, we replace all occurrences of the variable we want to forget with χ. Then, we immediately resolve every occurrence of χ, since it is not an element of propositional logic. We will call this operation $fgTrim$, as we "trim" the leaf nodes of the syntax tree.

The purpose of χ is to be removed from a binary formula without impacting the other side of a binary operator. Consequently, for any formula φ, we define $\chi \wedge \varphi = \varphi$ and $\chi \vee \varphi = \varphi$. This transfers to implication as $\chi \rightarrow \varphi = \varphi$ and $\varphi \rightarrow \chi = \neg\varphi$. For the issue with the logical equivalence operator, we

assume that if one side of an equivalence is irrelevant, the other side must be equivalently irrelevant. Therefore, we define $\chi \Leftrightarrow \varphi = \chi$ and we also receive $\chi \oplus \varphi = \neg(\chi \Leftrightarrow \varphi) = \neg\chi$. For negation, we assume the negation of an irrelevant variable (or formula) to also be irrelevant. We define $\neg\chi = \chi$, and therefore $\chi \oplus \varphi = \chi$.

This leaves the interpretation of atomic formulas. As χ can only be removed in concatenation with another variable, χ as the only remaining element of using $forgetTrim$ must be interpreted by the specific implementation. For example, one could argue that irrelevance as the result of a forget operation indicates that the result is obsolete and could be fully retracted from the respective knowledge base. In this work, we will interpret it as the absence of models, which would be equal to logical contradiction $\bot$.

We can now formally define $fgTrim$.

Definition 7 (Irrelevance assumption $fgTrim$) *Let φ be a well-formed propositional formula over an alphabet Σ and $v \in \Sigma$. The assumption of irrelevance for a variable v in φ is denoted as $fgTrim(\varphi, v) = \varphi[v/\chi]$. As the irrelevance placeholder χ is not an element of propositional logic, all occurrences of χ must be resolved immediately after the substitution of v in φ according to the following rules.*

Let θ be any well-formed propositional formula. χ interacts with propositional logic as follows:

1. $\neg\chi = \chi$
2. $\chi \wedge \theta = \theta$
3. $\chi \vee \theta = \theta$
4. $\chi \rightarrow \theta = \theta$
5. $\theta \rightarrow \chi = \neg\theta$
6. $\chi \Leftrightarrow \theta = \chi$
7. $\chi \oplus \theta = \chi$
8. $\chi \wedge \chi = \chi$
9. $\chi \vee \chi = \chi$
10. $\chi \rightarrow \chi = \chi$
11. $\chi \Leftrightarrow \chi = \chi$
12. $\chi \oplus \chi = \chi$

Variables			Example 1		Example 2		Example 3	
a	b	c	φ_1	$fgTrim(\varphi_1, a)$	φ_2	$fgTrim(\varphi_2, a)$	φ_3	$fgTrim(\varphi_3, a)$
0	0	0	0	0	0	0	1	1
0	0	1	1	1	0	1	1	0
0	1	0	1	1	1	1	0	1
0	1	1	1	1	1	1	1	0
1	0	0	0	0	0	0	0	1
1	0	1	1	1	1	1	1	0
1	1	0	1	1	0	1	0	1
1	1	1	1	1	1	1	0	0

Table 5: Truth table evaluation of irrelevance assumption examples. Green cells indicate preserved truth values, gray cells show false evaluations, and red cells highlight changes in truth value after forgetting.

Example 6 *Let* $\varphi_1 = b \vee (a \wedge c) \vee (\neg a \wedge c)$, $\varphi_2 = \neg(a \vee \neg b) \vee (a \wedge c) \vee (b \wedge c)$ *and* $\varphi_3 = ((a \rightarrow c) \wedge (b \rightarrow a)) \Leftrightarrow (b \wedge c)$.
To forget a *in each of the formulas, we obtain:*

$$\begin{aligned} fgTrim(\varphi_1, a) &= (b \vee (a \wedge c) \vee (\neg a \wedge c))[a/\chi] \\ &= b \vee (\chi \wedge c) \vee (\neg \chi \wedge c) \\ &= b \vee c \vee c \\ &= b \vee c \\ fgTrim(\varphi_2, a) &= (\neg(a \vee \neg b) \vee (a \wedge c) \vee (b \wedge c))[a/\chi] \\ &= \neg(\chi \vee \neg b) \vee (\chi \wedge c) \vee (b \wedge c) \\ &= \neg(\neg b) \vee c \vee (b \wedge c) \\ &= b \vee c \\ fgTrim(\varphi_3, a) &= (((a \rightarrow c) \wedge (b \rightarrow a)) \Leftrightarrow (b \wedge c))[a/\chi] \\ &= ((\chi \rightarrow c) \wedge (b \rightarrow \chi)) \Leftrightarrow (b \wedge c) \\ &= (c \wedge \neg b) \Leftrightarrow (b \wedge c) \end{aligned}$$

The evaluations of φ_1, φ_2 *and* φ_3 *and the respective results of forgetting* a *from the formulas are shown in Table 5.*

The properties variable elimination, invariance and idempotency hold for $fgTrim$. However, as semantics of the resulting formula are dependent on the syntactical structure of the original formula, no general relationship between $Mod(\varphi)$ and
$Mod(fgTrim(\varphi, v))$ can be guaranteed.

In contrast to $fgClassic$ and $fgSkep$, if a variable is successfully forgetten through $fgTrim$, the resulting formula will always have reduced size com-

Property	$fgClassic$	$fgSkep$	truth value assumption	$fgClRec$	$fgTrim$
Variable Elimination	y	y	y	y	y
Invariance	y	y	y	y	y
Idempotency	y	y	y	y	y

Table 6: Comparison of the properties of the variable forgetting methods

pared to the original formula, whereas results from the former methods will always have double the size of the original formula, before resolving any $\top$ or $\bot$.

3.6 Properties and comparison

So far, all methods for variable forgetting we named operate only on the syntax of the formula. $fgClassic$, $fgSkep$ and truth value assumption are global operations on the whole formula, $fgClRec$ attempts to localize $fgClassic$ as much as possible, and $fgTrim$ replaces the variable locally depending on context.

Generally, if φ is formed over a language Σ, forgetting a variable v forms a new formula, which is formed over a language $\Gamma = \Sigma \backslash \{v\}$. The number of possible models Ω_Γ is reduced by half compared to Ω_Σ, which reduces the semantic complexity, but also limits the amount of information that can be derived from the formula.

The operators we presented are all in a way syntactical modifications of each other; Truth value assumption, or more generally substitution, is the basis of all other methods for variable forgetting. $fgClassic$ was the first of these operators to be formally defined and uses substitution of variables. $fgSkep$ replaces the disjunction of $fgClassic$ with a conjunction in an attempt to minimize the models of the formula that results from forgetting a variable. $fgClRec$ attempts to reduce the resulting complexity of $fgClassic$ by applying localized operations to subformulas rather than replacing variables globally and duplicating the whole formula. Finally, $fgTrim$ attempts to forget variables as locally as possible by taking into account only the specific binary subformula of which the forgotten variable is part.

Regarding the semantics of the result of all forget operators we presented, only a few general statements can be made. Forgetting a variable from a formula using any of the introduced methods removes it from the syntax and also from all models of the formula. If the removed variable is irrelevant to the models of the formula, forgetting the variable will not modify the remaining part of all models of the formula, which holds for all variable forgetting methods. To describe this formally, let φ be a formula over an alphabet Σ, $v \in Sig(\varphi)$ and $\Gamma = \sigma \backslash \{v\}$. If the sets $M_\Gamma^+ = \{m_\Gamma^+ | m_\Gamma^+$ is the Γ-part of a model

$\omega \in Mod(\varphi)$ that includes the positive literal $v\}$ and $M_\Gamma^- = \{m_\Gamma^- | m_\Gamma^-$ is the Γ-part of a model $\omega \in Mod(\varphi)$ that includes the negative literal $\overline{v}\}$ are equal, v is irrelevant in φ and can be forgotten by at least $fgClassic$ (and therefore $fgClRec$), such that $Mod_\Sigma(fgClassic(\varphi, v) = Mod_\Sigma(\varphi)$.

4 Multiple variable forgetting

We will now analyze how the different methods perform when forgetting sets of variables. Although the focus of this work is on syntactic operators for multiple variable forgetting, we will also introduce model marginalization as a semantic approach to variable forgetting, together with a syntactic adaptation of model marginalization [7].

4.1 Classic/Skeptical Variable Forgetting

Because $fgClassic$ and $fgSkep$ are syntactically similar, we can discuss their properties regarding multiple variable forgetting at the same time.

Definition 8 (Multiple classic/skeptic variable forgetting) *Let φ be a formula over an alphabet Σ and $V \subseteq \Sigma$ with $V = \{v_1, v_2, ..., v_n\}$. Forgetting all variables $v \in V$ from φ using classic variable forgetting is denoted as*

$$\begin{aligned} fgClassic(\varphi, V) =& fgClassic(fgClassic(\varphi, v_n), V \backslash \{v_n\}) \\ =& fgClassic(...fgClassic(fgClassic(\varphi, v_1), v_2)..., v_n). \end{aligned}$$

Symmetrically, forgetting all variables $v \in V$ from φ using skeptical variable forgetting is denoted as

$$\begin{aligned} fgSkep(\varphi, V) =& fgSkep(fgSkep(\varphi, v_n), V \backslash \{v_n\}) \\ =& fgSkep(...fgSkep(fgSkep(\varphi, v_1), v_2)..., v_n). \end{aligned}$$

Example 7 *Let $\varphi = (a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d))$. Then $Mod(\varphi) = \{abcd, abc\overline{d}, ab\overline{c}d, ab\overline{c}\overline{d}, a\overline{b}cd, a\overline{b}c\overline{d}, \overline{a}bcd, \overline{a}bc\overline{d}\}$. To forget a and b from φ using classic or skeptical variable*

forgetting, we form

$$\begin{aligned}
fgClassic(\varphi, \{a, b\}) =& fgClassic((((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\top] \\
& \vee ((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\bot]), \{b\}) \\
=& (((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\top] \\
& \vee ((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\bot])[b/\top] \\
& \vee (((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\top] \\
& \vee ((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\bot])[b/\bot] \\
=& (((\top \wedge \top) \vee ((\top \wedge c) \oplus (\top \wedge d))) \\
& \vee ((\bot \wedge \top) \vee ((\top \wedge c) \oplus (\bot \wedge d)))) \\
& \vee (((\top \wedge \bot) \vee ((\bot \wedge c) \oplus (\top \wedge d))) \\
& \vee ((\bot \wedge \bot) \vee ((\bot \wedge c) \oplus (\bot \wedge d)))) \\
=& (\top \vee (c \oplus d)) \vee (\bot \vee (c \oplus \bot)) \\
& \vee (\bot \vee (\bot \oplus d)) \vee (\bot \vee (\bot \oplus \bot)) \\
=& \top
\end{aligned}$$

and

$$\begin{aligned}
fgSkep(\varphi, \{a, b\}) =& fgSkep((((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\top] \\
& \wedge ((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\bot]), \{b\}) \\
=& (((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\top] \\
& \wedge ((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\bot])[b/\top] \\
& \wedge (((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\top] \\
& \wedge ((a \wedge b) \vee ((b \wedge c) \oplus (a \wedge d)))[a/\bot])[b/\bot] \\
=& (((\top \wedge \top) \vee ((\top \wedge c) \oplus (\top \wedge d))) \\
& \wedge ((\bot \wedge \top) \vee ((\top \wedge c) \oplus (\bot \wedge d)))) \\
& \wedge (((\top \wedge \bot) \vee ((\bot \wedge c) \oplus (\top \wedge d))) \\
& \wedge ((\bot \wedge \bot) \vee ((\bot \wedge c) \oplus (\bot \wedge d)))) \\
=& ((\top \vee (c \oplus d)) \wedge (\bot \vee (c \oplus \bot))) \\
& \wedge ((\bot \vee (\bot \oplus d)) \wedge (\bot \vee ((\bot \oplus \bot))) \\
=& (c \oplus d) \wedge (c \oplus \bot) \wedge (\bot \oplus d) \wedge \bot \\
=& \bot
\end{aligned}$$

This example illustrates the tendency for $fgClassic$ to maximize models, and for $fgSkep$ to minimize models. Most importantly, it shows the quickly increasing complexity of both methods when forgetting sets of variables. With every iteration of $fgClassic$ or $fgSkep$, the length of the resulting formula

doubles, and its depth increases by 1 because of the additional conjunction or disjunction. This exponentially increasing length means that after n iterations of $fgClassic$ of $fgSkep$, there are 2^n variations of the original formula, and $2^n - 1$ additional conjunctions or disjunctions in the resulting formula.

The syntactic properties of singular variable forgetting can be transferred to multiple variable forgetting. Also, we note that the order in which the variables of a set are forgotten does not matter [4].

- Set Variable Elimination: $\forall v \in S : v \notin Sig(forget(\varphi, S))$
- Set Idempotency: $\forall S' \subseteq S : \ forget(forget(\varphi, S), S') = forget(forget(\varphi, S'), S) = forget(\varphi, S)$
- Set Element Irrelevance: $\forall v \in S : v \notin Sig(\varphi) \Leftrightarrow forget(\varphi, S) = forget(\varphi, \{S \backslash v\})$
- Set Irrelevance: $S \cap Sig(\varphi) = \emptyset \Leftrightarrow forget(\varphi, S) = \varphi$
- Order independence: $\forall S' = \sigma(S) : forget(\varphi, S) = forget(\varphi, S')$.

Forgetting multiple variables using $fgClassic$ or $fgSkep$ shows no semantic difference from forgetting single variables, since we forget the variables of the set one by one iteratively. Therefore, all properties of single variable forgetting still hold when forgetting sets of variables.

4.2 Truth value assumption

Truth value assumption stands out from other variable forgetting methods because it maintains linear complexity when forgetting sets of variables. For sets, $fgTrue$ and $fgFalse$ could even be combined into a single operator. For a variable φ over an alphabet Σ, two sets $T \subseteq \Sigma$ and $F \subseteq \Sigma$ with $T \cap F = \emptyset$, we can denote the assumption of $\top$ for all $t \in T$ and of $\bot$ for all $f \in F$ with $fgAssumeTruth(\varphi, T, F) = \varphi[T/\top][F/\bot]$.

Similarly to $fgClassic$ and $fgSkep$, there are no further semantical differences when forgetting sets of variables over forgetting variables iteratively using truth value assumption, and all properties, including the properties we established for multiple variable forgetting for $fgClassic$ and $fgSkep$, still hold here. As such, the advantages and disadvantages of the method remain the same in multiple variable forgetting.

4.3 Recursive Variable Forgetting

Compared to the previous operators, $fgClRec$ behaves slightly different when forgetting multiple variables. While variables are still processed individually, the forget operations are localized to the smallest relevant subformulas. Specifically, if a variable appears only in one branch of a conjunction,

only that branch needs to be modified. This exploits the distributivity of $fgClassic$ by analyzing which variables occur in which signatures of subformulas, we can apply forget operations locally rather than globally, reducing the syntactic overhead.

Definition 9 (Recursive Classic Multiple Variable Forgetting $fgClRec$) *Let $\varphi_1 = v$, $\varphi_2 = \neg\psi$, $\varphi_3 = \psi \vee \theta$, $\varphi_4 = \psi \wedge \theta$ and $S \subseteq \Sigma$. Then*

$$\begin{aligned}
fgClRec(\varphi_1, S) =& fgClassic(\varphi_1, S)\\
fgClRec(\varphi_2, S) =& fgClassic(\varphi_1, S)\\
fgClRec(\varphi_3, S) =& fgClRec(\psi, S) \vee fgClRec(\theta, S) and\\
fgClRec(\varphi_4, S) =& fgClassic(\\
& fgClRec(\psi, \{(S \cap Sig(\psi))\backslash(Sig(\psi) \cap Sig(\theta))\})\\
& \wedge fgClRec(\theta, \{(S \cap Sig(\theta))\backslash(Sig(\psi) \cap Sig(\theta))\}),\\
& S \cap (Sig(\psi) \cap Sig(\theta)))
\end{aligned}$$

The case φ_1 is simple. If we have an atomic formula, we only need to use classic variable forgetting. In case of $v \in S$, we can abbreviate to $fgClRec(v, S) = \top$, as $fgClRec(\varphi_1, S) = fgClRec(v, S) = forgetClassic(v, S) = \top \vee \bot = \top$. For a negated formula in the form of φ_2, we cannot recurse any further, since $forgetClassic(\neg\varphi, S) \neq \neg forgetClassic(\varphi, S)$ does not hold for all formulas φ. At this point, we can introduce an operator $fgSkRec$ as the $fgSkep$ counterpart of $fgClRec$, in which case we could form $forgetClassic(\neg\varphi, S) = \neg fgSkRec(\varphi, S)$.

For φ_3, we again use the distributivity of $fgClassic$ over disjunction. Through combination of the cases φ_2 and φ_3, we also receive $fgClRec((\psi \rightarrow \theta), S) = fgClRec(\neg\psi, S) \vee fgClRec(\theta, S) = fgClassic(\neg\psi, S) \vee fgClRec(\theta, S)$.

The more complex case is a conjunction such as $fgClRec(\varphi_4, S)$. We split the $fgClRec$-operation into three parts. First, we distribute all variables in $S\backslash(Sig(\psi) \cap Sig(\theta))$ over two additional operations of $fgClRec$ on ψ and θ by respectively forgetting the variables which are in exactly one of the two signatures. Formally, we use the subset of S which contains all the variables of S that occur in $Sig(\psi)$ and not in $Sig(\theta)$ to form $fgClRec(\psi, \{(S \cap Sig(\psi))\backslash(Sig(\psi) \cap Sig(\theta))\})$, and vice versa for $Sig(\theta)$ to form $fgClRec(\theta, \{(S \cap Sig(\theta))\backslash(Sig(\psi) \cap Sig(\theta))\})$. Then, we form a disjunction of the two $forget$-formulas and use $fgClassic$ on the resulting formula to forget the set of all variables from S that are elements of $Sig(\psi)$ and also elements of $Sig(\theta)$, formally $S \cap Sig(\psi) \cap Sig(\theta)$. We demonstrate how $fgClRec$ performs over $fgClassic$ in an example.

Example 8 *Let $\varphi = ((a \vee b) \rightarrow (b \wedge c)) \wedge ((e \wedge \neg d) \vee \neg(e \wedge a))$ and $F = \{a, b, d\}$. To forget F from φ using $fgClassic$ and $fgClRec$, we form*

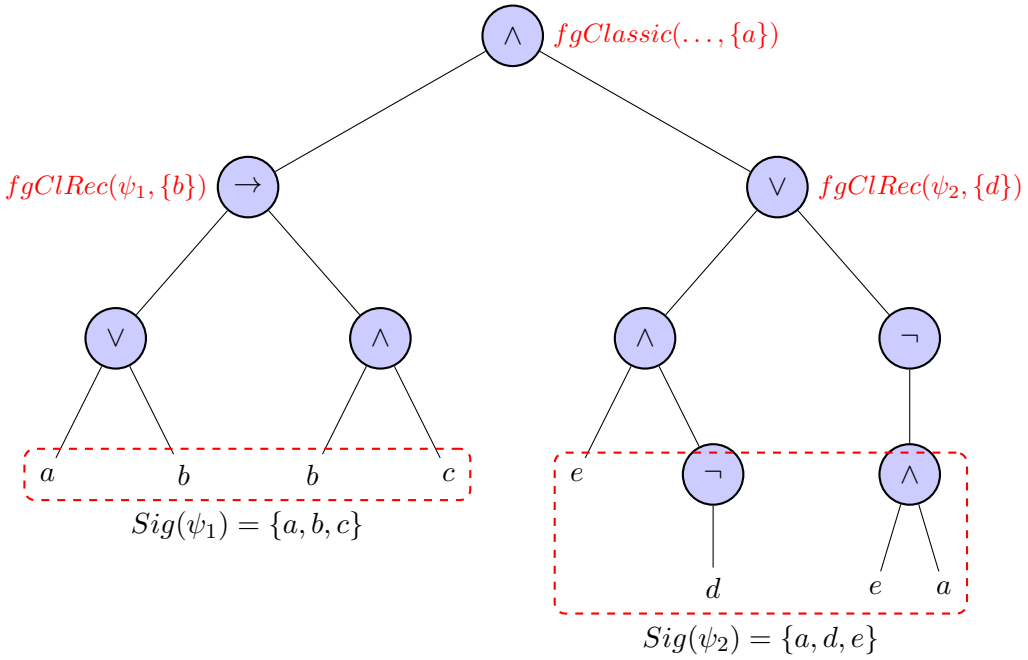

Figure 2: Syntax tree with signatures for recursive classic variable forgetting of $\{a, b, d\}$ from $\varphi = \psi_1 \wedge \psi_2$ where $\psi_1 = (a \vee b) \rightarrow (b \wedge c)$, $\psi_2 = (e \wedge \neg d) \vee \neg(e \wedge a)$.

$$
\begin{aligned}
fgClassic(\varphi, F) =& ((((((\top \vee \top) \rightarrow (\top \wedge c)) \wedge ((e \wedge \neg\top) \vee \neg(e \wedge \top)))\\
&\vee (((\bot \vee \top) \rightarrow (\top \wedge c)) \wedge ((e \wedge \neg\top) \vee \neg(e \wedge \bot))))\\
&\vee ((((\top \vee \bot) \rightarrow (\bot \wedge c)) \wedge ((e \wedge \neg\top) \vee \neg(e \wedge \top)))\\
&\vee (((\bot \vee \bot) \rightarrow (\bot \wedge c)) \wedge ((e \wedge \neg\top) \vee \neg(e \wedge \bot)))))\\
&\vee ((((((\top \vee \top) \rightarrow (\top \wedge c)) \wedge ((e \wedge \neg\bot) \vee \neg(e \wedge \top)))\\
&\vee (((\bot \vee \top) \rightarrow (\top \wedge c)) \wedge ((e \wedge \neg\bot) \vee \neg(e \wedge \bot))))\\
&\vee ((((\top \vee \bot) \rightarrow (\bot \wedge c)) \wedge ((e \wedge \neg\bot) \vee \neg(e \wedge \top)))\\
&\vee (((\bot \vee \bot) \rightarrow (\bot \wedge c)) \wedge ((e \wedge \neg\bot) \vee \neg(e \wedge \bot)))))\\
=& \top\\
fgClRec(\varphi, F) =& fgClassic((fgClRec(((a \vee b) \rightarrow (b \wedge c)), b)\\
&\wedge fgClRec(((e \wedge \neg d) \vee \neg(e \wedge a)), d)), a)\\
=& fgClassic(((fgClassic((\neg(a \vee b)), b) \vee fgClRec((b \wedge c), b))\\
&\wedge (fgClRec((e \wedge \neg d), d) \vee \neg(e \wedge a))), a)\\
=& fgClassic(((fgClRec((\neg(a \vee b)), b) \vee (fgClassic(b, b) \wedge c))\\
&\wedge ((e \wedge fgClassic(\neg d, d)) \vee \neg(e \wedge a))), a)\\
=& (((\neg((\top \vee \top))) \vee (\neg((\top \vee \bot))) \rightarrow (\top \wedge c))\\
&\wedge ((e \wedge (\neg\top) \vee (\neg\bot)) \vee \neg(e \wedge \top)))\\
&\vee (((\neg((\bot \vee \top))) \vee (\neg((\bot \vee \bot))) \rightarrow (\top \wedge c))\\
&\wedge ((e \wedge (\neg\top) \vee (\neg\bot)) \vee \neg(e \wedge \bot)))\\
=& \top
\end{aligned}
$$

For the sake of brevity, we omit some transformation steps in this example. Figure 2 illustrates the distribution of the different $fgClRec$ formulas through $fgClRec(\varphi, F)$.

This example again demonstrates the reduced increment in size of $fgClRec$ over $fgClassic$, but it also demonstrates the reduction of operations required to form the $forget$-formula for both methods. To form $fgClassic(\varphi, F)$, the original formula φ is duplicated a total of seven times, requiring 8 substitutions in variants of φ for each variable of F, adding up to a total of 24 substitutions on φ and increasing the size of the formula from 8 to 64 and the depth from 3 to 6.

On the other hand, $fgClRed(\varphi, F)$ requires only one duplication of the whole formula, and distributes the $fgClassic$-formulas over the subformulas $\neg(a \vee b)$, b, and $\neg d$, for a total size of 22, and also a depth of 6.

As $fgClRec$ is semantically the same as $fgClassic$ when forgetting single variables, all semantic properties of $fgClRec$ for multiple variable forgetting hold here as well. As such, for a formula φ over a language Σ with $\Gamma = \Sigma \backslash S$, for all $m \in M$ with $M = \{\omega^\Gamma | \omega^\Gamma \; is \; the \; \Gamma - part \; of \; \omega \in Mod(\varphi)\}$, the relation $M \subseteq Mod(fgClassic(\varphi, S)$ holds.

4.4 Irrelevance assumption

Similar to truth value assumption, the substitution of the variables we want to forget from a formula with χ is linear performance. The following resolution of χ remains the same as for single variable forgetting.

For multiple variable forgetting, this resolution of χ maintains the complexity of single variable forgetting, as the size of the immediate result from substitution is the same as the original formula. If $\Leftrightarrow$ or $\oplus$ are part of the modified formula, forgetting multiple variables could even reduce the complexity of the resolution, as larger subformulas could be erased through cases like $\chi \Leftrightarrow \psi$.

However, in this case, depending on the syntactical structure of the modified formula, the risk of removing the whole formula and receiving only χ as the result of $fgTrim$ increases with the number of forgotten variables. Consider the formula $\varphi = \chi \Leftrightarrow \theta$. If we form $fgTrim(\varphi, S)$ and $Sig(\chi) \cap S = Sig(\chi)$ or $Sig(\theta) \cap S = Sig(\theta)$, the respective subformula will be reduced to χ, and thus $fgTrim(\varphi, S) = \chi$.

We demonstrate multiple variable forgetting with $fgTrim$ by considering φ from example 8.

Example 9 *Let $\varphi = ((a \vee b) \rightarrow (b \wedge c)) \wedge ((e \wedge \neg d) \vee \neg(e \wedge a))$ and $F = \{a, b, d\}$.*

To forget F from φ using $fgTrim$, we form

$$\begin{aligned} fgTrim(\varphi, F) =& (((a \vee b) \rightarrow (b \wedge c)) \wedge ((e \wedge \neg d) \vee \neg(e \wedge a)))[\{a, b, d\}/\chi] \\ =& ((\chi \vee \chi) \rightarrow (\chi \wedge c)) \wedge ((e \wedge \neg\chi) \vee \neg(e \wedge \chi)) \\ =& c \wedge (e \vee \neg e) \end{aligned}$$

The example demonstrates the reduction in size of the result formula, and also how other variables we did not explicitly remove through $fgTrim$ for the most part remain in the formula. We can see that the syntactical structure of the formula can lead to additional variables becoming irrelevant after $fgTrim$, like e in $fgTrim(\varphi, F)$. As with the other methods for multiple variable forgetting, the properties from single variable forgetting still hold here, as well as the set properties from subsection 4.1.

4.5 Model marginalization and syntactic model marginalization

This method for variable forgetting was defined by Sauerwald [7]. Model marginalization is an operation originating from probability theory. In propositional logic, the marginalization of a model ω over an alphabet Σ to an alphabet $\Gamma \subseteq \Sigma$ is the removal of all variables $v \in (\Sigma \backslash \Gamma)$ from ω.

Definition 10 (Model Marginalization [7] $ModMg$) *Let $\omega \in \Omega_\Sigma$, $S \subseteq \Omega_\Sigma$ and $\Gamma \subseteq \Sigma$. The marginalization of ω from Σ to Γ is $ModMg_\Sigma(\omega, \Gamma) = \omega^\Gamma$. The element-wise marginalization for all $\omega \in S$ from Σ to Γ is $ModMg_\Sigma(S, \Gamma) = \{ModMg(\omega, \Gamma | \omega \in S\}$.*

Example 10 *Let φ be a formula over $\Sigma = \{a, b, c, d\}$ with $\Gamma = \{a, b, c\}$, $Mod(\varphi) = \{abcd, abc\overline{d}, ab\overline{c}d, a\overline{b}cd, a\overline{b}c\overline{d}, \overline{a}bcd, \overline{a}bc\overline{d}, \overline{a}b\overline{c}d\}$ and $\omega = \overline{a}bc\overline{d}$. The marginalization of ω from Σ to Γ is $ModMg_\Sigma(\omega, \Gamma) = \overline{a}bc$. The element-wise marginalization of $Mod(\varphi)$ to Γ is $ModMg_\Sigma(Mod(\varphi), \Gamma) = \{abc, ab\overline{c}, a\overline{b}c, \overline{a}bc, \overline{a}b\overline{c}\}$.*

As a semantic operation, the syntax of the formula is not relevant for the result of model marginalization, as we operate only on the signature and model of the formula. Of course, this means that in order to perform marginalization on the models of a formula, we need to construct the models first.

Sauerwald et al. proposed a syntactic interpretation of model marginalization, which they named *syntactic marginalization*.

Definition 11 (Syntactic Marginalization [7] $SynMg$) *Let φ be a formula over Σ and let $\Gamma \subseteq \Sigma$. The syntactic marginalization of φ from Σ to Γ, written $SynMg_\Sigma(\varphi, \Gamma)$ is $fgClassic(\varphi, \Sigma \backslash \Gamma)$.*

Operator	Best Use Case	Reasoning
$fgClassic$	General purpose, small formulas	Standard method, preserves all semantic relations
$fgSkep$	Knowledge base minimization	Minimizes truth values, useful for conservative reasoning
Truth Value Assumption	Known variable values, real-time systems	Fastest method, minimal complexity increase
$fgTrim$	Large formulas, memory constraints	Minimal syntactic overhead, local operations
$fgClRec$	Formulas with localized variables	Reduces complexity when variables appear in few subformulas
Model Marginalization	Complete semantic accuracy needed	Semantic approach, computationally expensive

Table 7: Recommended use cases for different variable forgetting operators based on formula characteristics and application requirements.

4.6 Comparison

Most operators presented in this work are syntactical, with marginalization as the only semantic approach. This distinction is significant, as semantic methods require construction of all models of a formula. For a formula with 20 variables, this means enumerating over one million possible assignments, and therefore making semantic approaches impractical for large signatures.

The remaining syntactic methods can be split between global and local operations. Classic forget, skeptical forget, and truth value assumption are global operations that affect the whole formula without regard to specific syntax. Their performance depends only on the length of the formula and is independent from their syntactical structure. This uniformity makes them predictable in terms of computational cost, but potentially inefficient when variables appear in isolated subformulas.

The simplest and fastest method is truth value assumption, since it scales linearly, regardless of how many variables are forgotten. A single iteration through the token sequence suffices to replace all target variables with the chosen truth value. This efficiency is achieved by considering only one possible truth value per variable, but therefore results in greater information loss compared to $fgClassic$. As shown in the examples, truth value assumption can discard valid models or even introduce false ones, making it in particular suitable when the truth value of the forgotten variable is actually constant across all relevant scenarios.

$fgClassic$ and $fgSkep$ reach exponential syntactical complexity, as the formula size doubles with each forgotten variable. Forgetting n variables se-

quentially results in a formula of length $2^n \cdot |\varphi|$. This growth could be counteracted by simplifying the formula through resolving truth values and use of De Morgan's laws between iterations. For instance, after forgetting one variable, subformulas like $\top \wedge \psi$ can be reduced to ψ before forgetting the next variable. However, this adds additional computational overhead and does not always eliminate exponential growth.

$fgClRec$ tries to minimize the exponential complexity of $fgClassic$ by limiting the affected portion to the deepest possible subformulas. Instead of duplicating the entire formula, only the subformulas that actually contain the target variable are modified. For a formula $\psi \wedge \theta$ where the variable v occurs only in ψ, we compute $fgClassic(\psi, v) \wedge \theta$ rather than $fgClassic(\psi \wedge \theta, v)$. In contrast, this operation requires computing signatures for all subformulas to determine variable containment, which increases overhead that scales with formula depth and structure.

$fgTrim$ is a local method, as the exact operation performed for each occurrence of a forgotten variable differs depending on the operators the variable is bound to. A variable in a conjunction is handled differently than one in a disjunction or equivalence. Unlike the other methods, $fgTrim$ operates locally on subformulas without formula duplication by replacing variables with the irrelevance placeholder χ and immediately resolving it according to operator-specific rules. This keeps syntactic complexity low, but provides no general guarantees about the relation between $Mod(\varphi)$ and $Mod(fgTrim(\varphi, v))$, as the semantic effect depends on the syntactic structure of the modified formula.

The choice of operator ultimately depends on the specific requirements of the application. If semantic precision is essential and formulas are small, model marginalization of $fgClassic$ is appropriate despite the computational cost. For large formulas where efficiency is prioritized over complete sematic preservation, $fgTrim$ or truth value assumption offer practical alternatives with understood trade-offs.

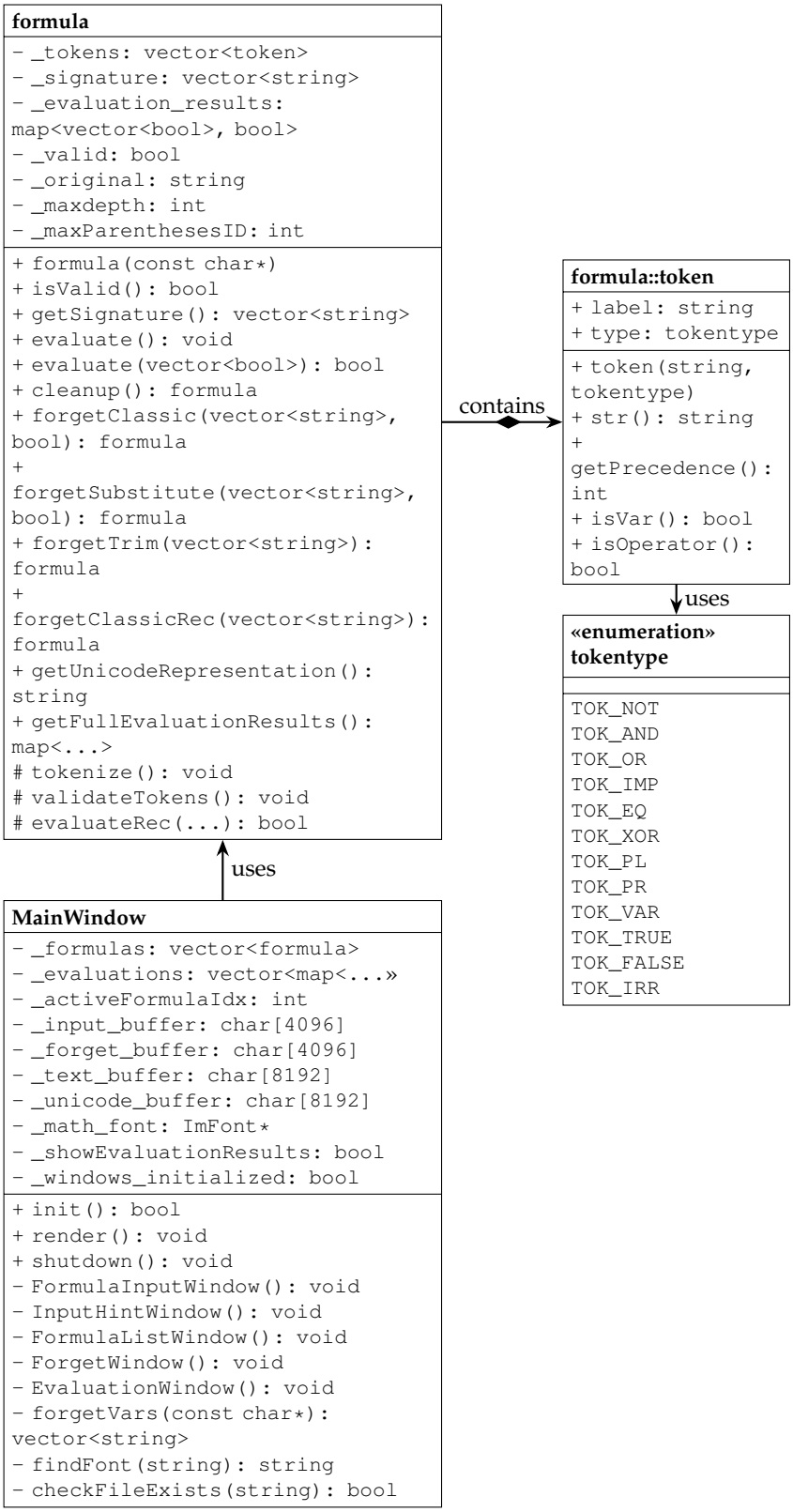

Figure 3: UML Class Diagram of PLVF Tool

5 Implementation - PLVF Tool

PLVF Tool stands for Propositional Logic and Variable Forgetting Tool and is a prototypical implementation of the syntactic variable forgetting operators which were presented in this work. We decided to use C++ as our programming language for this because of the precise control over memory management and data structures, but also because of the used user interface (UI) library, *Dear ImGui* [1], which we will call ImGui for short.

ImGui is an open-source lightweight graphical UI library for C++ and was specifically designed for real-time applications. Unlike traditional graphical UI libraries, ImGui operates in immediate mode, that is, UI elements are drawn and updated every frame without maintaining an overall persistent state, resulting in simpler integration and a reduction of memory overhead. Its simplicity, lack of external dependencies, and flexibility make it especially ideal for prototyping and tooling, making it a solid choice for the implementation of PLVF Tool.

5.1 Structure of the implementation

The implementation consists of the three classes `main.cpp`, `formula.cpp` and `mainwindow.cpp`, together with two header files `formula.h` and `mainwindow.h` for the respective classes. The general structure of the classes and their purpose is illustrated in the UML diagram Figure 3.

The obligatory class `main` contains the main loop and initialization of the other classes. Initialization includes the setup of the used graphics device for ImGui, which depends on the used operating system, loading the used fonts and special characters, and initialization of the input maps for propositional formulas, which connect different input strings to the respective propositional symbol.

The class `mainwindow` handles the UI, which means display and input. The UI consists of four ImGui windows, which is illustrated in Figure 4

A formula can be entered using the input field of the window titled “Formula Input & Analysis” and is then added to the formula list, where the formula can be selected for further analysis and modification. Using the window titled “Forget operations and cleanup”, the syntactical variable forgetting methods of this work can be applied to the selected formula, and a truth table of the selected formula can be displayed in the remaining window.

All propositional and forgetting logic is handled in the class `formula`. To represent propositional formulas, we use a `struct` called `token`, which has a `std::string` label and uses an `enum tokentype` to determine the specific element of propositional logic a token represents, for example `TOK_TRUE`

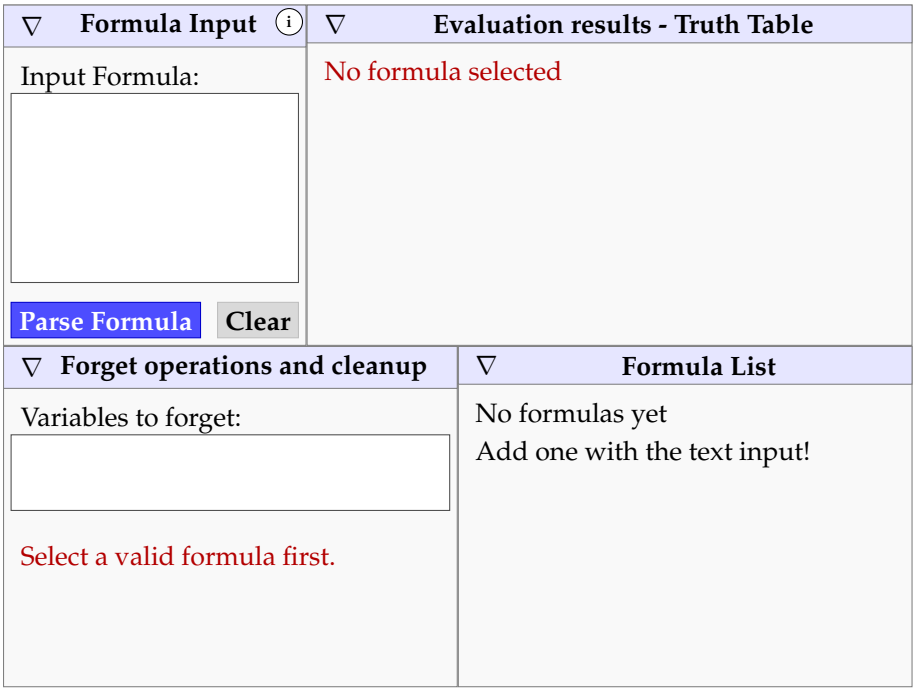

Figure 4: Conceptual graphic of PLVF Tool

for $\top$, `TOK_AND` for logical conjunction $\wedge$, `TOK_VAR` for variables or `TOK_PL` and `TOK_PR` for parentheses. The label of a token is used to preserve the textual representation of the formula by saving the names of the variables, but also to match parentheses by assigning IDs during syntax validation of a formula.

An instance of `formula` is normally created through the public constructor `formula(const char *input)`, which uses a pointer to an array of type `char` that represents the text form of a propositional formula. From the input, a `std::vector<formula::token>` is created using `formula::tokenize()`, which is then syntactically analyzed in `formula::validateTokens()` to determine whether the constructed token sequence represents a valid propositional formula. During syntactical analysis, parentheses are assigned IDs instead of a textual label to make pair matching easy.

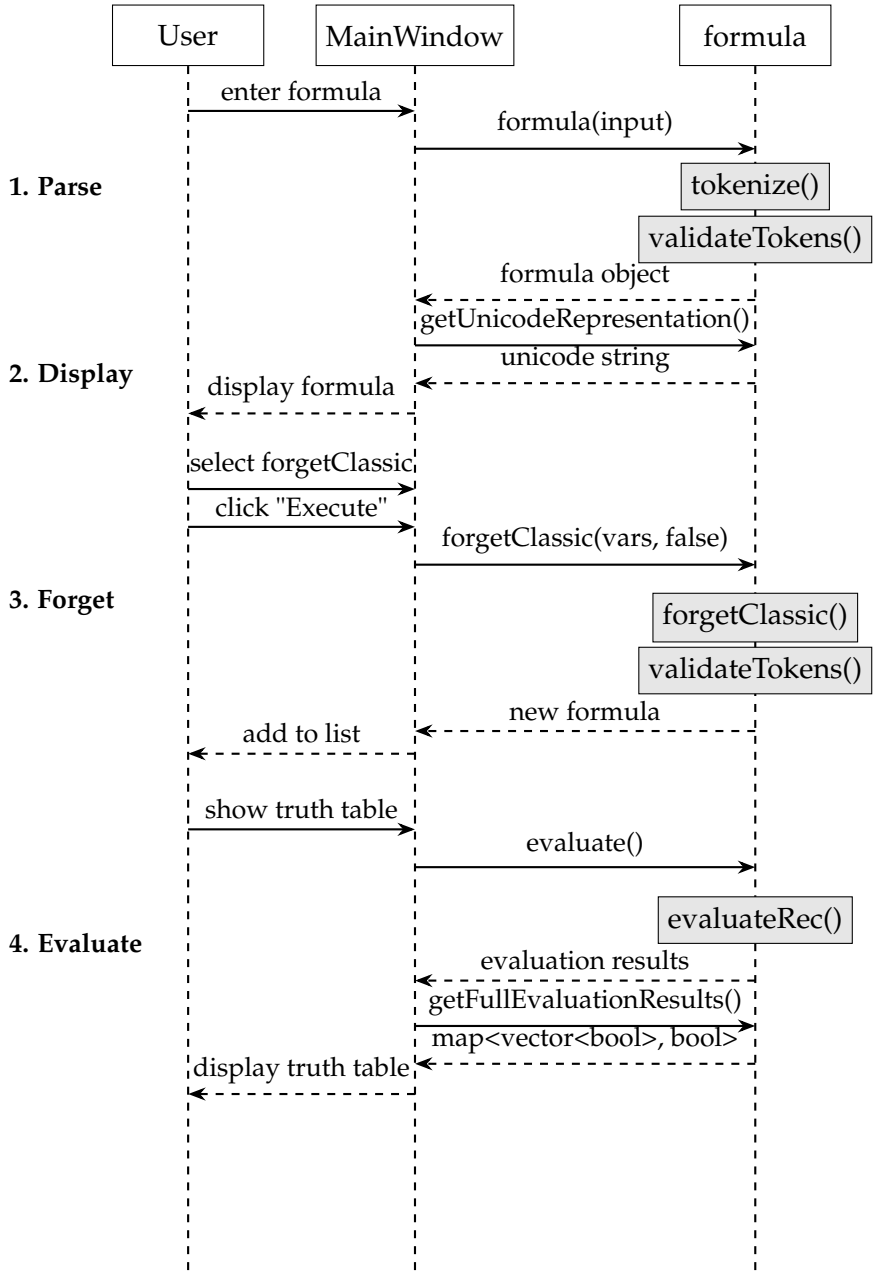

Figure 5: Sequence Diagram: Complete Workflow from Formula Input to Evaluation

The forget functions instead use the private constructor `formula(const std::vector<token> &tokens)`, which skips the lexical analysis of the text input. This constructor does not skip the syntactical analysis, because parentheses resulting from forgetting variables need to be assigned IDs which

are unique within the formula.

Figure 5 shows an example program flow after full program initialization for entering a formula, forgetting variables from the formula, and displaying the truth table.

5.2 implementation of forget operators

We will describe the implementation of the forget functions in pseudo code, for which we require some more preliminaries.

The functions for variable forgetting are implemented as members of the class `formula`. Accordingly, they are called on an instance of `formula` which we want to modify. For example, we have a `formula` f from which we want to forget a variable v using classic variable forgetting, we call `f.forgetClassic({"v"},false)`. For skeptical variable forgetting, we would call `f.forgetClassic({"v"},true)` instead.

The first parameter in all forget functions is a `std::set<std::string> vars` and is the set of variables we want to forget from the formula. `forgetClassic` and `forgetSubstitute` are called with an additional `boolean` parameter. In `forgetClassic`, passing `true` for this parameter indicates to use skeptical variable forgetting instead of classic variable forgetting. In `forgetSubstitute`, the boolean value indicates whether to replace the variables of `vars` with $\top$ or $\bot$.

All forget functions return a new `formula` object, which uses the private constructor `formula(const std::vector<token> &tokens)` instead of the public constructor `formula(const char *input)`, as there is no textual input that needs to be converted to tokens in lexical analysis.

Objects of the type `formula` have the attributes `std::vector<formula::token> _tokens`, which contains the token sequence, and `std::set<std::string> _signature`, which is a set of the labels of all variable tokens that occur in `tokens`. Note that a vector is an ordered sequence of objects and may contain duplicates, but a set is not ordered and does not contain duplicates.

We use the type `std::vector` for our token sequence, because a vector uses arrays, which allows access to elements in constant time using numbers for positions. A disadvantage of vectors is that stored numerical positions are invalidated if elements are inserted into or erased from the vector. For these cases we use the type `std::list`, which operate using iterators instead of numerical positions. Note that a `std::list` has an extra empty element after the last object of the list. The function `std::list::begin()` returns an iterator to the first element of the list, and `std::list::end()` returns an iterator to the end of the list, which is the mentioned empty element.

For the pseudo code, we assume the function `copy(source, dest)` to copy all elements of a container `source` to a container of the same type `dest`. The functions `toList(container)` and `toVector(container)` return a `list`- or `vector`-copy of `container`, respectively. `next()` and `prev()` are functions of iterators that return an iterator pointing to the next or previous element in the container. To access the object an iterator points to, we use an arrow `->`, for example, if the iterator `it` points to a `token`, we access the label of the token with the expression `it->label`.

The function `findMainOp(tokens, start, end)` searches for the operator with the currently highest precedence in `tokens` within the range of `start` to `end`, and returns the numerical position, if `tokens` is a vector, or an iterator that points to the token of the operator, if `tokens` is a list.

The signature of a function includes the name of the function, the parameters in parentheses, and the return type of the function. For example, `function forgetClassic(set<string> vars, boolean skepForget): formula` describes a function called `forgetClassic` which requires a pareme-ter of the type
`set<string>` called `vars` and a `boolean` value called `skepForget` and returns an object of the type `formula`.

We begin with the analysis of the implementation of truth value assumption.

```
1  function forgetSubstitute(set<string> vars, boolean replacement):formula
2  {
3      vector<token> resultVector = copy(_tokens);
4      for(token t:resultVector)
5          if(t.type == TOK_VAR && vars.contains(t.label))
6              if(replacement)
7                  t = token("true", TOK_TRUE);
8              else
9                  t = token("false", TOK_FALSE);
10
11     return formula(resultVector);
12 }
```

`forgetSubstitute` is most simple implementation. At first, the vector `_tokens` of the formula object is copied (line 3), and then iterated through using a `for` loop. When a token of the type `TOK_VAR` is encountered and `vars` contains the label of the token (line 5), we replace it with a token of type `TOK_TRUE` or `TOK_FALSE`, depending on the parameter `replacement` (lines 6-9). After iterating over the token sequence once, a new formula using the modified vector is returned(line 11), which guarantees linear complexity.

The function `forgetClassic` implements both $fgClassic$ and $fgSkep$ because of their syntactic similarity. Whether $fgClassic$ of $fgSkep$ is used is determined by the boolean `skepForget`.

```
1  function forgetClassic(set<string> vars, boolean skepForget):formula
```

```
2 {
3     vector<token> resultVector = _tokens;
4
5     for(string var:vars)
6     {
7         if(NOT _signature.contains(var))
8             continue;
9
10        vector<token> currentTokens=copy(resultVector);
11        vector<int> positions;
12        vector<token> newTokens;
13
14        for(int i = 0; i<currentTokens.size(); i++)
15            if (currentTokens[i].label == var)
16            {
17                positions.insert(i);
18                currentTokens[i] = token("true", TOK_TRUE);
19            }
20
21        newTokens.insert(token("(",TOK_PL));
22        newTokens.insert(currentTokens);
23        newTokens.insert(")",TOK_PR);
24        if (skepForget)
25            newTokens.insert("and", TOK_AND);
26        else
27            newTokens.insert("or", TOK_OR);
28
29        for(int pos : positions)
30            currentTokens[i] = token("false", TOK_FALSE);
31
32        newTokens.insert(token("(",TOK_PL));
33        newTokens.insert(currentTokens);
34        newTokens.insert(")",TOK_PR);
35
36        resultVector = newTokens;
37    }
38
39    return formula(resultVector);
40 }
```

At the start of function, we set a variable `resultVector` to the current token sequence of the formula. The main part of the function is a `for` loop, which is iterated once for every string `var` in the parameter `vars`, similar to how $fgClassic$ iterates through a set of variables. If the signature of the formula does not contain `var`, the current token sequence is not modified, and we skip the current iteration (lines 7-8). Otherwise, we copy the current result token sequence `resultVector` into a vector `currentTokens` (line 10), save the positions of all occurrences of `var` in `currentTokens` into a vector `positions`(lines 11, 14-17) and substitute all occurrences of tokens with the label `var` in `currentTokens` with tokens of the type `TOK_TRUE` (line 18). After that, we construct the $forget$-formula into the vector `newTokens`; We

add a opening parentheses token `TOK_PL`, copy the sequence `currentTokens`, where our target variable was replaced with `TOK_TRUE` into `newTokens` and close the left subformula with a closing parentheses token `TOK_PR` (lines 21-23). We insert the operator token `TOK_AND` or `TOK_OR` depending on the parameter `skepForget` (lines 24-27) and substitute the tokens in `currentTokens` at the positions from `positions` with tokens of the type `TOK_FALSE` (lines 29-30), before constructing the second half of the $forget$-formula the same way we constructed the first half (lines 32-34). At the end of an iteration, we set the variable `resultVector` to `newTokens`, which now contains the token sequence that results of forgetting `var`. After all elements of `vars` are iterated through, we construct a new `formula` using `resultVector` and return it.

As a recursive method, $fgClRec$ is implemented in two functions. The first function `forgetClRecInit` initializes the recursion and returns the `formula`, and `forgetClRec` modifies the token sequence according to the definition of $fgClRec$.

```
1 function forgetClRecInit(set<string> vars):formula
2 {
3     list<token> resultTokens = toList(_tokens);
4     forgetClRec(resultTokens, resultTokens.begin(),
5                 resultTokens.end().prev(), vars);
6     return formula( toVector(resultTokens));
7 }
```

The initialization through `forgetClRecInit` is simple. The current token vector is copied into a list, modified through `forgetClRec`, and then turned back into a vector, which is used to return the new `formula`. In this function, we use lists instead of vectors, because we partially modify the token vector in different calls of `forgetClRec`. If we were to use a vector, numeric target positions would be invalidated through the removal of insertion of new tokens into the vector through recursive calls of `forgetClRec`, but a `listIterator` points to the same element, even if the list around it is modified, as long as the element the iterator points to is not removed from the list.

We call the recursive function `forgetClRec` with the list `tokens` we want to modify, the range we want to modify, indicated by the `listIterators` `start` and `end`, and the set `vars` of variables, we want to forget. Note the return type `void`, which indicates that no value is returned; we modify `tokens` directly.

```
1 function forgetClRec(list<token> tokens, listIterator start,
2                      listIterator end, set<string> vars):void
3 {
4     if (vars.empty())
5         return;
6
```

```
7     if(start==end)
8     {
9         if(start->type == TOK_VAR && vars.contains(start->label))
10        {
11            tokens.insert(start,token("true",TOK_TRUE));
12            tokens.erase(start);
13        }
14        return;
15    }
16
17    if (start->type==TOK_PL && end->type==TOK_PR
18        && start->label == end->label)
19    {
20        forgetClRec(tokens, start.next(), end.prev(), vars)
21        return;
22    }
23
24    listIterator op = findMainOp(tokens, start, end);
25
26    if(op->type == TOK_NOT || op->type == TOK_EQ || op->type == TOK_XOR)
27    {
28        vector temp = forgetClassic(vector(start, end.next()),
29                                    vars, false);
30        tokens.insert(end.next(), temp);
31        tokens.erase(start, end);
32        return;
33    }
34
35    if (op->type == TOK_OR)
36    {
37        forgetClRec(tokens, start, op.prev(), vars);
38        forgetClRec(tokens, op.next(), end, vars);
39        return;
40    }
41
42    if (op->type == TOK_IMP)
43    {
44        tokens.insert(start, token("not", TOK_NOT));
45        tokens.insert(start, token("(", TOK_PL));
46        tokens.insert(op, token(")", TOK_PR));
47        forgetClassicRec(tokens, start.prev(2), op.prev()), vars);
48        forgetClassicRec(tokens, op.next(), end, vars);
49        return;
50    }
51
52    set<string> signatureL;
53    for (auto it = start; it != op; advance(it, 1))
54        if (it->type == TOK_VAR)
55        {
56            signatureL.insert(it->label);
57        }
58
59    set<string> signatureR;
```

```
60      for (auto it = op; it != end.next(); advance(it, 1))
61          if (it->type == TOK_VAR)
62          {
63              signatureR.emplace(it->label);
64          }
65
66      set<string> removeVarsL;
67      set<string> removeVarsR;
68      set<string> removeVarsBoth;
69
70      for (string var: vars)
71      {
72          const bool l = signatureL.contains(var);
73          const bool r = signatureR.contains(var);
74          if (l && r)
75          {
76              removeVarsBoth.emplace(var);
77              continue;
78          }
79          if (l)
80          {
81              removeVarsL.emplace(var);
82              continue;
83          }
84          if (r)
85              removeVarsR.emplace(var);
86      }
87
88      listIterator backupStart = start.prev;
89      listIterator backupEnd = end.prev;
90
91      forgetClassicRecRec(tokens, start, op.prev, removeVarsL);
92      forgetClassicRecRec(tokens, op.next, end, removeVarsR);
93
94      std::advance(backupStart, 1);
95      std::advance(backupEnd, -1);
96
97      if (!removeVarsBoth.empty())
98      {
99          vector temp = forgetClassic(vector(backupStart, backupEnd.next),
100                                      removeVarsBoth, false);
101         tokens.insert(end.next, temp);
102         tokens.erase(start, end);
103         return;
104     }
105 }
```

We first catch some special cases. If there are no variables to remove, we abort the modification of the current token range. If `start` and `end` point to the same token, and the token has the type `TOK_VAR` and `vars` contains the label of the token, we replace it with a `TOK_TRUE` token. This mirrors the behavior of $fgClassic$ on atomic formulas, since $\top \vee \bot = \top$, we can insert a

representation of $\top$ directly. After that, we check if `start` and `end` point to a matching parentheses pair, in which case we modify the formula inside the parentheses through a recursive call of `forgetClRec` and stop the current execution of the function.

Having handled the special cases, we determine the operator of the current formula, and continue according to the definition of $fgClRec$:

If the operator is a negation `TOK_NOT`, a logical equivalence `TOK_EQ` or logical inequivalence `TOK_XOR`, we use `forgetClassic` to forget the set `vars` from the current token range. Note that this implementation of `forgetClassic` is not a function on a `formula`, but uses a `vector` instead. We assume this function to work the same way as the function above, but to return a `vector` instead of a `formula`.

If the operator is a disjunction `TOK_OR`, we recursively call `forgetClRec` on the subformulas on both halves of `op` and abort the current call.

For implication `TOK_IMP`, we first negate the subformula left of `op`, and proceed as in the case of `TOK_OR` with recursive calls on both sides of `op`.

The last remaining case is conjunction `TOK_AND`, for which we first construct the signatures of the subformulas as the sets `signatureL` and `signatureR` (lines 49-61). Using the signatures, we construct the sets `removeVarsL` and `removeVarsR` which contain the variables we want to forget recursively from the left and right subformulas, or if the variable occurs in both formulas, we place it in the corresponding set `removeVarsBoth` instead (lines 67-83). We then save listIterators for the elements before `start` and after `end`, as the listIterators `start` and `end` might be modified through the following recursive calls. Note that `start.prev()` and `end.next()` always exist, as `start.prev()` loops back to the end of the list, if `start` is the first element of the list and vice versa. We intentionally initialized the first call of `forgetClRec` with `resultTokens.end().prev()` as the parameter `end`.

Finally, we shift the iterators `backupStart` and `backupEnd` back to their correct positions and forget the variables of `tokens` in the range `backupStart` to `backupEnd` using `forgetClassic`.

The implementation of $fgTrim$ consists of two parts, like the original method. In the first function `forgetTrim`, we replace all tokens of the variables we want to forget with tokens of the type `TOK_IRR`, which represent χ. Then, we initiate a recursive function `forgetTrimRec` to resolve all occurrences of `TOK_IRR`.

```
1 function forgetTrim(set<string> vars):formula
2 {
3     list tokensList = toList(_tokens);
4     for (token &t: tokensList)
5         if (t.type == TOK_VAR && vars.contains(t.type))
6                       t = token("", TOK_IRR);
```

```
7
8       forgetTrimCleanup(tokensList, tokensList.begin(),
9                         tokensList.end().prev());
10
11      vector resultVector = toVector(tokensList);
12      if (resultVector.size() == 1 && resultVector[0].type == TOK_IRR)
13          resultVector[0] = token("false", TOK_FALSE);
14
15      return formula(resultVector);
16  }
```

First, we transform our token sequence into a list, because the positions of tokens within the container can be modified in recursive calls of `fgTrimCleanup`. After that, we replace all occurrences of all variables of `vars` in `tokensList` with tokens of the type `TOK_IRR`. Then we initialize the recursive resolution of `TOK_IRR` with the function `fgTrimCleanup`. After the resolution of `TOK_IRR` tokens is complete, we copy the remaining token list into a variable `resultVector` and check whether the only remaining element is a `TOK_IRR` token, in which case we replace is with a token of the type `TOK_FALSE`, to represent an absence of models for the new formula, as we defined in $fgTrim$. Finally, we return a formula using `resultVector`.

The resolution of `TOK_IRR` is implemented as a recursive method and called similarly to `fgClRec` with a `list` of tokens and a range given by the parameters `start` and `end`. Because the substitution of variables was already done in `fgTrim`, we do not need a parameter `vars`.

```
1   function forgetTrimCleanup(list<token> tokens, listIterator start,
2                              listIterator end):void
3   {
4       if (start == end)
5           return;
6
7       if (start->type == TOK_PL && end->type == TOK_PR
8           && start->label == end->label)
9       {
10          while (start.next()->type == TOK_PL && end.prev()->type == TOK_PR
11                 && start.next()->label == end.prev()->label)
12          {
13              tokens.erase(start.next());
14              tokens.erase(end.prev());
15          }
16
17          forgetTrimCleanup(tokens, start.next(), end.prev());
18
19          if (start.next() == end.prev())
20          {
21              tokens.erase(start);
22              tokens.erase(end);
23          }
```

```
24          return;
25      }
26
27      listIterator op = findMainOp(tokens, start, end);
28
29      if (op->type == TOK_NOT)
30      {
31          forgetTrimCleanup(tokens, op.next();, end);
32          if (op.next()->type == TOK_IRR)
33              tokens.erase(op);
34          return;
35      }
36
37      listIterator backupStart = start.prev();
38      listIterator backupEnd = end.next();
39
40      forgetTrimCleanup(tokens, start, op.prev());
41      forgetTrimCleanup(tokens, op.next();, end);
42
43      std::advance(backupEnd, -1);
44      std::advance(backupStart, 1);
45
46      if (prev->type == TOK_IRR)
47      {
48          switch (op->type)
49          {
50              case TOK_IMP:
51              case TOK_AND:
52              case TOK_OR:
53                  tokens.erase(op.prev(), op);
54                  break;
55              case TOK_XOR:
56              case TOK_EQ:
57                  tokens.erase(op, backupEnd);
58          }
59      } else if (next->type == TOK_IRR)
60      {
61          switch (op->type)
62          {
63              case TOK_IMP:
64                  tokens.insert(backupStart, token("not", TOK_NOT));
65                  tokens.insert(backupStart, token("(", TOK_PL));
66                  tokens.insert(op, token(")", TOK_PR));
67              case TOK_AND:
68              case TOK_OR:
69                  tokens.erase(op, op.next());
70                  break;
71              case TOK_XOR:
72              case TOK_EQ:
73                  tokens.erase(backupStart, op.next());
74          }
75      }
76  }
```

Similarly to `forgetClRec`, the special cases are treated first in `forgetTrimCleanup`, too. If `start` and `end` point to the same element, there is nothing to resolve and the function is aborted. If the tokens the iterators `start` and `end` point to are matching parentheses, any additional matching parentheses are removed through a `while` loop, and `forgetTrimCleanup` is called recursively on the token sequence between `start` and `end`. In the case that after this cleanup only a single token remains between `start` and `end`, the parentheses tokens `start` and `end` point are not needed and removed. After that, the cleanup of the given token sequence is complete and we abort the function.

For any other case, the position of the main operator `op` is detected using `findMainOp`. In case of a negation, `forgetTrimCleanup` is first called recursively in the range from `op.next()` to `end`. If another negation then follows after `op`, both negation tokens are removed. After this, the resolution is complete and the function call is stopped.

For any binary operator, we first save the positions of the start and end of the current token range in the listIterators `backupStart` and `backupEnd` and recursively call `forgetTrimCleanup` on the subformulas in the range `start` to `op.prev()` and `op.next()` to `end`. In case one of `op.prev()` or `op.next()` point to a token of type `TOK_IRR` after the cleanup of the subformulas is complete, the token is resolved according to the definition of $fgTrim$; if the operator is a `TOK_EQ` or `TOK_XOR`, the subformula opposite of the `TOK_IRR` token is removed along with the operator, and if `op` is a token of the types `TOK_IMP`, `TOK_AND` or `TOK_OR`, the `TOK_IRR` token is removed along with the operator. In the special case that the `TOK_IRR` token is on the right side of an implication, the subformula on the left side of the operator is also negated.

6 Conclusion

In this work, we characterized five different syntactic methods for multiple variable forgetting and implemented them using C++ in the prototypical program PLVFTool. Aside from presenting the existing methods classic variable forgetting, skeptical variable forgetting and model marginalization, we also specialized classic variable forgetting to work more efficiently when forgetting multiple variables, and introduced irrelevance assumption as a possible alternative to quickly remove variables from formulas while mostly keeping other variables intact.

6.1 Future Work

For future discussion, the newly introduced operators for recursive variable forgetting and irrelevance assumption could be explored more deeply and ported to other logical systems, such as first-order logic. As research into multiple variable forgetting is almost nonexistent, other possible approaches which may be more complex than those introduced in this work could be developed and explored as well. As the concepts of irrelevancy and dependency are deeply connected to variable forgetting, exploring the connection with multiple variable forgetting is another probable point of research.

The implementation of propositional logic and variable forgetting operators we produced as part of this work builds a solid foundation to support any research into multiple variable forgetting. Since the program is open source, more features can be developed easily and existing features can be optimized by more experienced developers.

References

[1] Omar Cornut and various contributors. Dear ImGui, https://github.com/ocornut/imgui.git, 2025.

[2] Thomas Eiter and Gabriele Kern-Isberner. A brief survey on forgetting from a knowledge representation and reasoning perspective. *KI-Künstliche Intelligenz*, 33(1):9–33, 2019.

[3] Christoph Kaplan. An investigation of a notion of „variable forgetting" that minimizes truth values, Jul 2024.

[4] Jérôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence-formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[5] Florian Lerch. Propositional Logic and Variable Forgetting Tool https://github.com/flolrch-9677216/PLVFTool, 2025.

[6] Fangzhen Lin and Ray Reiter. Forget it. In *Working Notes of AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

[7] Kai Sauerwald, Christoph Beierle, and Gabriele Kern-Isberner. Propositional variable forgetting and marginalization: Semantically, two sides of the same coin. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 144–162. Springer, 2024.