# Towards Practical First-Order Model Counting

**Anonymous submission**


## Abstract

First-order model counting (FOMC) is the problem of counting the number of models of a sentence in first-order logic. Since lifted inference techniques rely on reductions to variants of FOMC, the design of scalable methods for FOMC has attracted attention from theoreticians and practitioners alike over the past decade. Recently, significant advances were achieved with the introduction of first-order knowledge compilation-based approach, called CRANE, that instead of simply providing the final count, generates definitions of (possibly recursive) functions, which can be evaluated with different arguments to compute the model count for any domain size. We introduce a compilation algorithm that transforms the function definitions into C++ code, equipped with arbitrary-precision arithmetic. These additions allow the new FOMC algorithm to scale to domain sizes over 500 000 times larger than the current state of the art, as demonstrated through experimental results.


## 1 Introduction

*First-order model counting* (FOMC) is the task of counting the number of models of a sentence in first-order logic over some given domain(s). The weighted variant of this problem, known as WFOMC, seeks to compute the total weight of the models (Van den Broeck et al. 2011). WFOMC is related to its propositional predecessor weighted model counting (Chavira and Darwiche 2008) and other attempts to unify logic and probability (Nilsson 1986; Novák, Perfilieva, and Mockor 2012; Šaletić 2024). It is also a key approach to *lifted inference*, which aims to compute probabilities more efficiently by leveraging symmetries in the problem (Kersting 2012).

Lifted inference is an active area of research, with recent work in domains such as constraint satisfaction problems (Totis et al. 2023) and probabilistic answer set programming (Azzolini and Riguzzi 2023). WFOMC has been used for inference on probabilistic databases (Gribkoff, Suciu, and Van den Broeck 2014) and probabilistic logic programs (Riguzzi et al. 2017). FOMC algorithms have been utilised for discovering new integer sequences (Svatos et al. 2023), and for conjecturing (Barvínek et al. 2021) and constructing (Dilkas and Belle 2023) recurrence relations and other recursive structures that describe these sequences. FOMC algorithms have also been extended to perform *sampling* (Wang et al. 2022, 2023).

The complexity of FOMC is typically characterised in terms of *data complexity*. If there is an algorithm that can compute the FOMC of a formula in polynomial time with respect to the domain size(s), that formula is called *liftable* (Jaeger and Van den Broeck 2012). Beame et al. (2015) demonstrated the existence of an unliftable formula with three variables. It is also known that formulas with up to two variables are liftable (Van den Broeck 2011; Van den Broeck, Meert, and Darwiche 2014). The liftable fragment of formulas with two variables has been expanded with various axioms (Tóth and Kuželka 2023; van Bremen and Kuželka 2023), counting quantifiers (Kuželka 2021) and in other ways (Kazemi et al. 2016).

There are many FOMC algorithms with different underlying principles. Perhaps the most prominent class of FOMC algorithms is based on *first-order knowledge compilation* (FOKC). In this approach, the formula is compiled into a representation (such as a circuit or graph) by applying *compilation rules*. Algorithms in this class include FORCLIFT (Van den Broeck et al. 2011) and its extension CRANE (Dilkas and Belle 2023). Another FOMC algorithm, FASTWFOMC (van Bremen and Kuželka 2021), is based on cell enumeration. Other algorithms utilise local search (Niu et al. 2011), junction trees (Venugopal, Sarkhel, and Gogate 2015), Monte Carlo sampling (Gogate and Domingos 2016), and anytime approximation via upper/lower bound construction (van Bremen and Kuželka 2020).

The recently proposed CRANE algorithm marked significant progress in handling formulas beyond the capabilities of FASTWFOMC and FORCLIFT, yet it fell short in critical aspects. CRANE was incomplete since it could only construct function definitions, requiring users to manually evaluate these functions to obtain model counts. This limitation prevented CRANE from serving as a convenient black box solution for FOMC. Furthermore, it introduced recursive functions without defining necessary base cases, adding further complexity to the users. In this work, we present CRANE2, addressing these gaps and pushing scalability to unprecedented levels. Unlike its predecessor, CRANE2 is a fully automated FOMC algorithm, capable of handling domain sizes over 500 000 times larger than previous algorithms.

Figure 1 outlines the workflow of the new algorithm. In Section 3, we describe how `CompileWithBaseCases` finds base cases for recursive functions. Section 4 explains post-processing techniques to preserve the correct model count. Section 5 elucidates how function definitions are com-

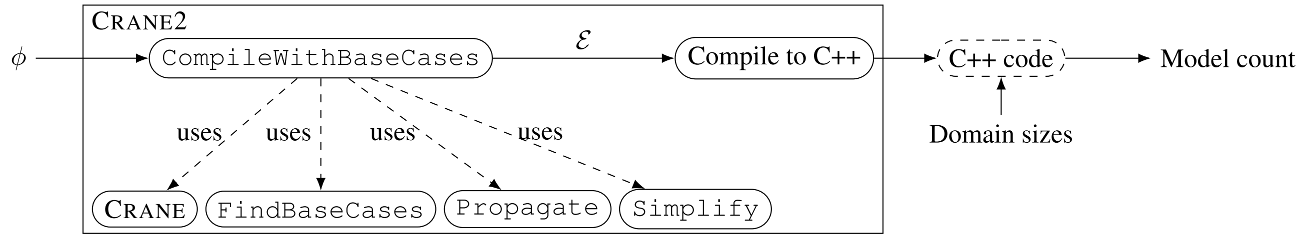


Figure 1: The outline of using CRANE2 to compute the model count of a formula $\phi$. First, the formula is compiled into a set of equations, which are then used to create a C++ program. This program can be executed with different command line arguments to calculate the model count of $\phi$ for different domain sizes. To accomplish this, the `CompileWithBaseCases` function employs several components: (i) the FOKC algorithm of CRANE, (ii) a procedure called `FindBaseCases`, which identifies a sufficient set of base cases, (iii) a procedure called `Propagate`, which constructs a formula corresponding to a given base case, and (iv) algebraic simplification techniques (denoted as `Simplify`).

piled into C++ programs. Note that a solution to FOMC that uses compilation to C++ has been considered before (Kazemi and Poole 2016), however, the extent of formulas that could be handled was limited. Finally, Section 6 presents experimental results comparing CRANE2 with other FOMC algorithms.

## 2 Preliminaries

In Section 2.1, we summarise the basic principles of first-order logic. Then, in Section 2.2, we formally define (W)FOMC and discuss the distinctions between three variations of first-order logic used for FOMC. Finally, in Section 2.3, we introduce the terminology used to describe the output of the original CRANE algorithm, i.e., functions and equations that define them.

We use $\mathbb{N}_0$ to represent the set of non-negative integers. In both algebra and logic, we write $S\sigma$ to denote the application of a *substitution* $\sigma$ to an expression $S$, where $\sigma = [x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n]$ signifies the replacement of all instances of $x_i$ with $y_i$ for all $i = 1, \ldots, n$.

### 2.1 First-Order Logic

In this section, we will review the basic concepts of first-order logic as they are used in FOKC algorithms. There are two key differences between the logic used by these algorithms and the logic supported as input. First, Skolemization (Van den Broeck, Meert, and Darwiche 2014) eliminates existential quantifiers by introducing additional predicates. Please note that Skolemization here differs from the standard Skolemization procedure that introduces function symbols (Hodges 1997). Second, the input formula is rewritten as a conjunction of clauses, each in *prenex normal form* (Hinman 2018).

A *term* can be either a variable or a constant. An *atom* can be either (i) $P(t_1, \ldots, t_m)$ for some predicate $P$ and terms $t_1, \ldots, t_m$ (written as $P(\mathbf{t})$ for short) or (ii) $x = y$ for some terms $x$ and $y$. The *arity* of a predicate is the number of arguments it takes, i.e., $m$ in the case of the predicate $P$ mentioned above. We write $P/m$ to denote a predicate along with its arity. A *literal* can be either an atom (i.e., a *positive* literal) or its negation (i.e., a *negative* literal). An atom is *ground* if it contains no variables, i.e., only constants. A *clause* is of the form $\forall x_1 \in \Delta_1.\, \forall x_2 \in \Delta_2 \ldots \forall x_n \in \Delta_n.\, \phi(x_1, x_2, \ldots, x_n)$, where $\phi$ is a disjunction of literals that only contain variables $x_1, \ldots, x_n$ (and any constants). We say that a clause is a *(positive) unit clause* if (i) there is only one literal with a predicate, and (ii) it is a positive literal. Finally, a *formula* is a conjunction of clauses. Throughout the paper, we will use set-theoretic notation, interpreting a formula as a set of clauses and a clause as a set of literals.

### 2.2 FOMC Algorithms and Their Logics

In Table 1, we outline the differences among three first-order logics commonly used in FOMC: (i) $\mathsf{FO}$ is the input format for FORCLIFT* and its extensions CRANE† and CRANE2; (ii) $\mathsf{C}^2$ is often used in the literature on FASTWFOMC and related methods (Kuželka 2021; Malhotra and Serafini 2022); (iii) $\mathsf{UFO}^2 + \mathsf{CC}$ is the input format supported by the most recent implementation of FASTWFOMC‡. The notation we use to refer to each logic is standard in the case of $\mathsf{C}^2$ and $\mathsf{UFO}^2 + \mathsf{CC}$ (Tóth and Kuželka 2024) and redefined to be more specific in the case of $\mathsf{FO}$. All three logics are function-free, and domains are always assumed to be finite. As usual, we presuppose the *unique name assumption*, which states that two constants are equal if and only if they are the same constant (Russell and Norvig 2020).

In $\mathsf{FO}$, each term is assigned to a *sort*, and each predicate $P/n$ is assigned to a sequence of $n$ sorts. Each sort has its corresponding domain. These assignments to sorts are typically left implicit and can be reconstructed from the quantifiers. For example, $\forall x, y \in \Delta.\, P(x, y)$ implies that variables $x$ and $y$ have the same sort. On the other hand, $\forall x \in \Delta.\, \forall y \in \Gamma.\, P(x, y)$ implies that $x$ and $y$ have different sorts, and it would be improper to write, for example, $\forall x \in \Delta.\, \forall y \in \Gamma.\, P(x, y) \lor x = y$. $\mathsf{FO}$ is also the only logic to support constants, formulas with more than two variables, and the equality predicate. While we do not explicitly refer to sorts in subsequent sections of this paper, the many-sorted nature of $\mathsf{FO}$ is paramount to the algorithms presented therein.

*Remark.* In the case of FORCLIFT and its extensions, support for a formula as valid input does not imply that the algorithm

*https://github.com/UCLA-StarAI/Forclift

†https://doi.org/10.5281/zenodo.8004077

‡https://github.com/jan-toth/FastWFOMC.jl

| Logic | Sorts | Constants | Variables | Quantifiers | Additional atoms |
|---|---|---|---|---|---|
| $\mathsf{FO}$ | one or more | ✓ | unlimited | $\forall$, $\exists$ | $x = y$ |
| $\mathsf{C}^2$ | one | ✗ | two | $\forall$, $\exists$, $\exists^{=k}$, $\exists^{\leq k}$, $\exists^{\geq k}$ | — |
| $\mathsf{UFO}^2 + \mathsf{CC}$ | one | ✗ | two | $\forall$ | $\lvert P\rvert = m$ |

Table 1: A comparison of the three logics used in FOMC based on the following aspects: (i) the number of sorts, (ii) support for constants, (iii) the maximum number of variables, (iv) supported quantifiers, and (v) supported atoms in addition to those of the form $P(\mathbf{t})$ for a predicate $P/n$ and an $n$-tuple of terms $\mathbf{t}$. Here: (i) $k$ and $m$ are non-negative integers, with the latter depending on the domain size, (ii) $P$ represents a predicate, and (iii) $x$ and $y$ are terms.

can compile the formula into a circuit or graph suitable for lifted model counting. However, it is known that FORCLIFT compilation is guaranteed to succeed on any $\mathsf{FO}$ formula without constants and with at most two variables (Van den Broeck 2011; Van den Broeck, Meert, and Darwiche 2014).

Compared to $\mathsf{FO}$, $\mathsf{C}^2$ and $\mathsf{UFO}^2 + \mathsf{CC}$ lack support for (i) constants, (ii) the equality predicate, (iii) multiple domains, and (iv) formulas with more than two variables. The advantage that $\mathsf{C}^2$ brings over $\mathsf{FO}$ is the inclusion of *counting quantifiers*. That is, alongside $\forall$ and $\exists$, $\mathsf{C}^2$ supports $\exists^{=k}$, $\exists^{\leq k}$, and $\exists^{\geq k}$ for any positive integer $k$. For example, $\exists^{=1}x.\ \phi(x)$ means that there exists *exactly one* $x$ such that $\phi(x)$, and $\exists^{\leq 2}x.\ \phi(x)$ means that there exist *at most two* such $x$. $\mathsf{UFO}^2 + \mathsf{CC}$, on the other hand, does not support any existential quantifiers but instead incorporates *(equality) cardinality constraints*. For example, $|P| = 3$ constrains all models to have *precisely three positive literals with the predicate* $P$.

**Definition 1** (Model)**.** Let $\phi$ be a formula in $\mathsf{FO}$. For each predicate $P/n$ in $\phi$, let $(\Delta_i^P)_{i=1}^n$ be a list of the corresponding domains. Let $\sigma$ be a map from the domains of $\phi$ to their interpretations as sets, satisfying the following conditions: (i) the sets are pairwise disjoint, and (ii) the constants in $\phi$ are included in the corresponding domains. A *structure* of $\phi$ is a set $M$ of ground literals defined by adding to $M$ either $P(\mathbf{t})$ or $\neg P(\mathbf{t})$ for every predicate $P/n$ in $\phi$ and $n$-tuple $\mathbf{t} \in \prod_{i=1}^n \sigma(\Delta_i^P)$. A structure is a *model* if it satisfies $\phi$.

*Remark.* The distinctness of domains is important in two ways. First, in terms of expressiveness, a clause such as $\forall x \in \Delta.\ P(x, x)$ is valid if predicate $P$ is defined over two copies of the same domain and invalid otherwise. Second, having more distinct domains makes the problem more decomposable for the FOKC algorithm. With distinct domains, the algorithm can make assumptions or deductions about, e.g., the first domain of predicate $P$ without worrying how (or if) they apply to the second domain.

While this work focuses on FOMC, we still define the weighted variant of the problem as Skolemization relies on weights even for unweighted FOMC.

**Definition 2** (WFOMC instance)**.** A *WFOMC instance* comprises: (i) a formula $\phi$ in $\mathsf{FO}$, (ii) two (rational) *weights* $w^+(P)$ and $w^-(P)$ assigned to each predicate $P$ in $\phi$, and (iii) $\sigma$ as described in Definition 1. Unless specified otherwise, we assume all weights to be equal to 1.

**Definition 3** (WFOMC (Van den Broeck et al. 2011))**.** Given a WFOMC instance $(\phi, w^+, w^-, \sigma)$ as in Definition 2, the *(symmetric) weighted first-order model count* (WFOMC) of $\phi$ is

$$\sum_{M \models \phi} \prod_{P(\mathbf{t}) \in M} w^+(P) \prod_{\neg P(\mathbf{t}) \in M} w^-(P), \tag{1}$$

where the sum is over all models of $\phi$.

**Example 1** (Counting functions)**.** To define predicate $P$ as a function from a domain $\Delta$ to itself, in $\mathsf{C}^2$ one would write $\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$. In $\mathsf{UFO}^2 + \mathsf{CC}$, the same could be written as

$$\begin{gathered}(\forall x, y \in \Delta.\ S(x) \lor \neg P(x, y)) \land \\ (|P| = |\Delta|),\end{gathered} \tag{2}$$

where $w^-(S) = -1$. Although Formula (2) has more models compared to its counterpart in $\mathsf{C}^2$, the negative weight $w^-(S) = -1$ makes some of the terms in Equation (1) cancel out.

Equivalently, in $\mathsf{FO}$ we would write

$$\begin{gathered}(\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y)) \land \\ (\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \land P(x, z) \Rightarrow y = z).\end{gathered} \tag{3}$$

The first clause asserts that each $x$ must have at least one corresponding $y$, while the second statement adds the condition that if $x$ is mapped to both $y$ and $z$, then $y$ must equal $z$. It is important to note that Formula (3) is written with two domains instead of just one. However, we can still determine the correct number of functions by assuming that the sizes of $\Gamma$ and $\Delta$ are equal. This formulation, as observed by Dilkas and Belle (2023), can prove beneficial in enabling FOKC algorithms to find efficient solutions.

### 2.3 Algebra

We write `expr` to represent an arbitrary algebraic expression. It is important to note that some terms have different meanings in algebra and logic. In algebra, a *constant* refers to a non-negative integer. Likewise, a *variable* can either be a parameter of a function or a variable introduced through summation, such as $i$ in the expression $\sum_{i=1}^n \texttt{expr}$. A (function) *signature* is $f(x_1, \ldots, x_n)$ (or $f(\mathbf{x})$ for short), where $f$ represents an $n$-ary function, and each $x_i$ represents a variable. An *equation* is $f(\mathbf{x}) = \texttt{expr}$, with $f(\mathbf{x})$ representing a signature.

**Definition 4** (Base case)**.** Let $f(\mathbf{x})$ be a function call where each $x_i$ is either a constant or a variable (note that signatures are included in this definition). Then function call $f(\mathbf{y})$ is considered a *base case* of $f(\mathbf{x})$ if $f(\mathbf{y}) = f(\mathbf{x})\sigma$, where $\sigma$ is a substitution that replaces one or more $x_i$ with a constant.

**Algorithm 1:** `CompileWithBaseCases(`$\phi$`)`

**Input:** formula $\phi$
**Output:** set $\mathcal{E}$ of equations

1 $(\mathcal{E}, \mathcal{F}, \mathcal{D}) \leftarrow$ CRANE($\phi$);
2 $\mathcal{E} \leftarrow$ `Simplify(`$\mathcal{E}$`)`;
3 **foreach** *base case* $f(\mathbf{x}) \in$ `FindBaseCases(`$\mathcal{E}$`)` **do**
4 $\quad\psi \leftarrow \mathcal{F}(f)$;
5 $\quad$**foreach** *index $i$ such that* $x_i \in \mathbb{N}_0$ **do**
6 $\quad\quad\psi \leftarrow$ `Propagate(`$\psi$, $\mathcal{D}(f, i)$, $x_i$`)`;
7 $\quad\mathcal{E} \leftarrow \mathcal{E} \cup$ `CompileWithBaseCases(`$\psi$`)`;

**Algorithm 2:** `FindBaseCases(`$\mathcal{E}$`)`

**Input:** set $\mathcal{E}$ of equations
**Output:** set $\mathcal{B}$ of base cases

1 $\mathcal{B} \leftarrow \emptyset$;
2 **foreach** *function call* $f(\mathbf{y})$ *on the right-hand side of an equation in* $\mathcal{E}$ **do**
3 $\quad\mathbf{x} \leftarrow$ the parameters of $f$ in its definition;
4 $\quad$**foreach** $y_i \in \mathbf{y}$ **do**
5 $\quad\quad$**if** $y_i \in \mathbb{N}_0$ **then**
6 $\quad\quad\quad\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto y_i] \}$;
7 $\quad\quad$**else if** $y_i = x_i - c_i$ *for some* $c_i \in \mathbb{N}_0$ **then**
8 $\quad\quad\quad$**for** $j \leftarrow 0$ **to** $c_i - 1$ **do**
9 $\quad\quad\quad\quad\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto j] \}$;

## 3 Completing the Definitions of Functions

Before describing the main contribution of this work, let us review the essential aspects of FOKC as realised by CRANE. The input formula is compiled into: (i) set $\mathcal{E}$ of equations, (ii) map $\mathcal{F}$ from function names to formulas, and (iii) map $\mathcal{D}$ from function names and argument indices to domains. $\mathcal{E}$ can contain any number of functions, one of which (denoted by $f$) represents the solution to the FOMC problem. To compute the FOMC for particular domain sizes, $f$ must be evaluated with those domain sizes as arguments. $\mathcal{D}$ records this correspondence between function arguments and domains.

Algorithm 1 presents our overall approach for compiling a formula into equations that include the necessary base cases. To begin, we use the FOKC algorithm of the original CRANE to compile the formula into the three components: $\mathcal{E}$, $\mathcal{F}$, and $\mathcal{D}$. After some algebraic simplification, $\mathcal{E}$ is passed to the `FindBaseCases` procedure (see Section 3.1). For each base case $f(\mathbf{x})$, we retrieve the logical formula $\mathcal{F}(f)$ associated with the function name $f$ and simplify it using the `Propagate` procedure (explained in detail in Section 3.2). We do this by iterating over all indices of $\mathbf{x}$, where $x_i$ is a constant, and using `Propagate` to simplify $\psi$ by assuming that domain $\mathcal{D}(f, i)$ has size $x_i$. Finally, on line 7, `CompileWithBaseCases` recurses on these simplified formulas and adds the resulting base case equations to $\mathcal{E}$. Example 2 below provides more detail.

*Remark.* Although `CompileWithBaseCases` starts with a call to CRANE, the proposed algorithm is not just a post-processing step for FOKC because Algorithm 1 is recursive and can issue more calls to CRANE on various derived formulas.

**Example 2** (Counting bijections)**.** Consider the following formula (previously examined by Dilkas and Belle (2023)) that defines predicate $P$ as a bijection between two sets $\Gamma$ and $\Delta$:

$$\begin{gathered}(\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y)) \land \\ (\forall y \in \Delta.\ \exists x \in \Gamma.\ P(x, y)) \land \\ (\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \land P(x, z) \Rightarrow y = z) \land \\ (\forall x, z \in \Gamma.\ \forall y \in \Delta.\ P(x, y) \land P(z, y) \Rightarrow x = z).\end{gathered}$$

We specifically examine the first solution returned by CRANE2 for this formula.

After lines 1 and 2, we have

$$\mathcal{E} = \left\{ \begin{array}{l} f(m, n) = \sum_{l=0}^{n} \binom{n}{l} (-1)^{n-l} g(l, m), \\ g(l, m) = g(l-1, m) + m g(l-1, m-1) \end{array} \right\};$$

$$\mathcal{D} = \{ (f, 1) \mapsto \Gamma, (f, 2) \mapsto \Delta, (g, 1) \mapsto \Delta^\top, (g, 2) \mapsto \Gamma \},$$

where $\Delta^\top$ is a new domain. (We omit the definition of $\mathcal{F}$ as the formulas can get a bit verbose.) Then `FindBaseCases` identifies two base cases: $g(0, m)$ and $g(l, 0)$. In both cases, `CompileWithBaseCases` recurses on the formula $\mathcal{F}(g)$ simplified by assuming that one of the domains is empty. In the first case, we recurse on the formula $\forall x \in \Gamma.\ S(x) \lor \neg S(x)$, where $S$ is a predicate introduced by Skolemization with weights $w^+(S) = 1$ and $w^-(S) = -1$. Hence, we obtain the base case $g(0, m) = 0^m$. In the case of $g(l, 0)$, `Propagate(`$\psi$, $\Gamma$, 0`)` returns an empty formula, resulting in $g(l, 0) = 1$.

It is worth noting that these base cases overlap when $l = m = 0$ but remain consistent since $0^0 = 1$. Generally, let $\phi$ be a formula with two domains $\Gamma$ and $\Delta$, and let $n, m \in \mathbb{N}_0$. Then the FOMC of `Propagate(`$\phi$, $\Delta$, $n$`)` assuming $|\Gamma| = m$ is the same as the FOMC of `Propagate(`$\phi$, $\Gamma$, $m$`)` assuming $|\Delta| = n$.

Finally, the main responsibility of the `Simplify` procedure is to handle the algebraic pattern $\sum_{m=0}^{n} [a \le m \le b] f(m)$. Here: (i) $n$ is a variable, (ii) $a, b \in \mathbb{N}_0$ are constants, (iii) $f$ is an expression that may depend on $m$, and (iv) $[a \le m \le b] = \begin{cases} 1 & \text{if } a \le m \le b \\ 0 & \text{otherwise} \end{cases}$. `Simplify` transforms this pattern into $f(a) + f(a+1) + \cdots + f(\min\{ n, b \})$. For instance, in the case of Example 2, `Simplify` transforms $g(l, m) = \sum_{k=0}^{m} [0 \le k \le 1] \binom{m}{k} g(l-1, m-k)$ into $g(l, m) = g(l-1, m) + m g(l-1, m-1)$.

### 3.1 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of `FindBaseCases`. `FindBaseCases` considers two types of arguments when a function $f$ calls itself recursively: (i) constants and (ii) arguments of the form $x_i - c_i$, where $c_i$

**Algorithm 3:** `Propagate(`$\phi$, $\Delta$, $n$`)`

**Input:** formula $\phi$, domain $\Delta$, $n \in \mathbb{N}_0$
**Output:** formula $\phi'$

1 $\phi' \leftarrow \emptyset$;
2 **if** $n = 0$ **then**
3   **foreach** *clause* $C \in \phi$ **do**
4     **if** $\Delta \notin \mathrm{Doms}(C)$ **then** $\phi' \leftarrow \phi' \cup \{ C \}$;
5     **else**
6       $C' \leftarrow \{ l \in C \mid \Delta \notin \mathrm{Doms}(l) \}$;
7       **if** $C' \neq \emptyset$ **then**
8         $l \leftarrow$ an arbitrary literal in $C'$;
9         $\phi' \leftarrow \phi' \cup \{ C' \cup \{ \neg l \} \}$;
10 **else**
11   $D \leftarrow$ a set of $n$ new constants in $\Delta$;
12   **foreach** *clause* $C \in \phi$ **do**
13     $(x_i)_{i=1}^m \leftarrow$ the variables in $C$ with domain $\Delta$;
14     **if** $m = 0$ **then** $\phi' \leftarrow \phi' \cup \{ C \}$;
15     **else**
16       $\phi' \leftarrow \phi' \cup \{ C[x_1 \mapsto c_1, \ldots, x_m \mapsto c_m] \mid (c_i)_{i=1}^m \in D^m \}$;

is a constant and $x_i$ is the $i$-th argument of the signature of $f$. When the argument is a constant $c_i$, a base case with $c_i$ is added. In the second case, a base case is added for each constant from 0 up to (but not including) $c_i$.

**Example 3.** Consider the recursive function $g$ from Example 2. `FindBaseCases` iterates over two function calls: $g(l-1, m)$ and $g(l-1, m-1)$. The former produces the base case $g(0, m)$, while the latter produces both $g(0, m)$ and $g(l, 0)$.

It can be shown that the base cases identified by `FindBaseCases` are sufficient for the algorithm to terminate.[4]

**Theorem 1** (Termination)**.** *Let $\mathcal{E}$ represent the equations returned by* `CompileWithBaseCases`*. Let $f$ be an $n$-ary function in $\mathcal{E}$ and $\mathbf{x} \in \mathbb{N}_0^n$. Then the evaluation of $f(\mathbf{x})$ terminates.*

We prove Theorem 1 using double induction. First, we apply induction to the number of functions in $\mathcal{E}$. Then, we use induction on the arity of the 'last' function in $\mathcal{E}$ according to some topological ordering. For the detailed proof, please refer to the technical appendix.

## 3.2 Propagating Domain Size Assumptions

Algorithm 3, called `Propagate`, modifies the formula $\phi$ based on the assumption that $|\Delta| = n$. When $n = 0$, some clauses become vacuously satisfied and can be removed. When $n > 0$, partial grounding is performed by replacing all variables quantified over $\Delta$ with constants. (None of the formulas examined in this work had $n > 1$.) Algorithm 3 handles these two cases separately. For a literal or a clause $C$, the set of corresponding domains is denoted as $\mathrm{Doms}(C)$.

In the case of $n = 0$, there are three types of clauses to consider: (i) those that do not mention $\Delta$, (ii) those in which every literal contains variables quantified over $\Delta$, and (iii) those that have some literals with variables quantified over $\Delta$ and some without. Clauses of Type (i) are transferred to the new formula $\phi'$ without any changes. For clauses of Type (ii), $C'$ is empty, so these clauses are filtered out. As for clauses of Type (iii), a new kind of smoothing is performed, which will be explained in Section 4.

In the case of $n > 0$, $n$ new constants are introduced. Let $C$ be an arbitrary clause in $\phi$, and let $m \in \mathbb{N}_0$ be the number of variables in $C$ quantified over $\Delta$. If $m = 0$, $C$ is added directly to $\phi'$. Otherwise, a clause is added to $\phi'$ for every possible combination of replacing the $m$ variables in $C$ with the $n$ new constants.

**Example 4.** Let $C \equiv \forall x \in \Gamma. \forall y, z \in \Delta. \neg P(x, y) \lor \neg P(x, z) \lor y = z$. Then $\mathrm{Doms}(C) = \mathrm{Doms}(\neg P(x, y)) = \mathrm{Doms}(\neg P(x, z)) = \{ \Gamma, \Delta \}$, and $\mathrm{Doms}(y = z) = \{ \Delta \}$. A call to `Propagate(`$\{ C \}$, $\Delta$, 3`)` would result in the following formula with nine clauses:

$$
\begin{gathered}
(\forall x \in \Gamma. \neg P(x, c_1) \lor \neg P(x, c_1) \lor c_1 = c_1) \land \\
(\forall x \in \Gamma. \neg P(x, c_1) \lor \neg P(x, c_2) \lor c_1 = c_2) \land \\
\vdots \\
(\forall x \in \Gamma. \neg P(x, c_3) \lor \neg P(x, c_3) \lor c_3 = c_3).
\end{gathered}
$$

Here, $c_1$, $c_2$, and $c_3$ are the new constants.

# 4 Smoothing the Base Cases

*Smoothing* modifies a circuit to reintroduce eliminated atoms, ensuring the correct model count (Darwiche 2001; Van den Broeck et al. 2011). In this section, we describe a similar process performed on lines 8 and 9 of Algorithm 3. Line 7 checks if smoothing is necessary, and lines 8 and 9 execute it. If the condition on line 7 is not satisfied, the clause is not smoothed but omitted.

Suppose `Propagate` is called with arguments $(\phi, \Delta, 0)$, i.e., we are simplifying the formula $\phi$ by assuming that the domain $\Delta$ is empty. Informally, if there is a predicate $P$ in $\phi$ unrelated to $\Delta$, smoothing preserves all occurrences of $P$ even if all clauses with $P$ become vacuously satisfied.

**Example 5.** Let $\phi$ be:

$$(\forall x \in \Delta. \forall y, z \in \Gamma. Q(x) \lor P(y, z)) \land \tag{4}$$
$$(\forall y, z \in \Gamma'. P(y, z)), \tag{5}$$

where $\Gamma' \subseteq \Gamma$ is a domain introduced by a compilation rule. It should be noted that $P$, as a relation, is a subset of $\Gamma \times \Gamma$.

Now, let us reason manually about the model count of $\phi$ when $\Delta = \emptyset$. Predicate $Q$ can only take one value, $Q = \emptyset$. The value of $P$ is fixed over $\Gamma' \times \Gamma'$ by Clause (5), but it can vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since Clause (4) is vacuously satisfied by all structures. Therefore, the correct FOMC

[4]Note that characterising the fine-grained complexity of the solutions found by CRANE2 or other FOMC algorithms is an emerging area of research. These questions have been partially addressed in previous work (Dilkas and Belle 2023; Tóth and Kuželka 2024) and are orthogonal to the goals of this section.

should be $2^{|\Gamma|^2-|\Gamma'|^2}$. However, without line 9, `Propagate` would simplify $\phi$ to $\forall y, z \in \Gamma'.\ P(y,z)$. In this case, $P$ is a subset of $\Gamma' \times \Gamma'$. This simplified formula has only one model: $\{\, P(y,z) \mid y,z \in \Gamma' \,\}$. By including line 9, `Propagate` transforms $\phi$ to:

$$(\forall y, z \in \Gamma.\ P(y,z) \lor \neg P(y,z)) \land$$
$$(\forall y, z \in \Gamma'.\ P(y,z)),$$

which retains the correct model count.

It is worth mentioning that the choice of $l$ on line 8 of Algorithm 3 is inconsequential because any choice achieves the same goal: constructing a tautological clause that retains the literals in $C'$.

## 5 Generating C++ Code

In this section, we will describe the final step of CRANE2 as outlined in Figure 1. This step involves translating the set of equations $\mathcal{E}$ into C++ code. The resulting C++ program can then be compiled and executed with different command-line arguments to compute the model count of the formula for various domain sizes.

Each equation in $\mathcal{E}$ is compiled into a C++ function, along with a separate cache for memoisation. Let us consider an arbitrary equation $e = (f(\mathbf{x}) = \mathtt{expr}) \in \mathcal{E}$, and let $\mathbf{c} \in \mathbb{N}_0^n$ represent the arguments of the corresponding C++ function. The implementation of $e$ consists of three parts. First, we check if $\mathbf{c}$ is already present in the cache of $e$. If it is, we simply return the cached value. Second, for each base case $f(\mathbf{y})$ of $f(\mathbf{x})$ (as defined in Definition 4), we check if $\mathbf{c}$ *matches* $\mathbf{y}$, i.e., $c_i = y_i$ whenever $y_i \in \mathbb{N}_0$. If this condition is satisfied, $\mathbf{c}$ is redirected to the C++ function that corresponds to the definition of the base case $f(\mathbf{y})$. Finally, if none of the above cases apply, we evaluate $\mathbf{c}$ based on the expression `expr`, store the result in the cache, and return it.

## 6 Experimental Evaluation

In this section, we present experimental results that analyse the scalability of CRANE2 in comparison to two other notable FOMC algorithms: FASTWFOMC and FORCLIFT. Our experiments involve two versions of CRANE2: CRANE2-GREEDY and CRANE2-BFS. Like its predecessor, CRANE2 has two modes for applying compilation rules to formulas: one that uses a greedy search algorithm similar to FORCLIFT and another that combines greedy and breadth-first search.

We compare these algorithms using three benchmarks from previous studies. The first benchmark is the function-counting problem from Example 1, previously examined by Dilkas and Belle (2023). The second benchmark is a variant of the well-known 'Friends and Smokers' Markov logic network (Singla and Domingos 2008; Van den Broeck, Choi, and Darwiche 2012). In $\mathsf{C}^2$, $\mathsf{FO}$, and $\mathsf{UFO}^2 + \mathsf{CC}$, this problem can be formulated as

$$(\forall x, y \in \Delta.\ S(x) \land F(x,y) \Rightarrow S(y)) \land$$
$$(\forall x \in \Delta.\ S(x) \Rightarrow C(x))$$

or, equivalently, in conjunctive normal form as

$$(\forall x, y \in \Delta.\ S(y) \lor \neg S(x) \lor \neg F(x,y)) \land$$
$$(\forall x \in \Delta.\ C(x) \lor \neg S(x)).$$

Finally, we include the bijection-counting problem previously utilised by Dilkas and Belle (2023). Its formulation in $\mathsf{FO}$ is described in Example 2. The equivalent formula in $\mathsf{C}^2$ is

$$(\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x,y)) \land$$
$$(\forall y \in \Delta.\ \exists^{=1} x \in \Delta.\ P(x,y)).$$

Similarly, in $\mathsf{UFO}^2 + \mathsf{CC}$ the same formula can be written as

$$(\forall x, y \in \Delta.\ R(x) \lor \neg P(x,y)) \land$$
$$(\forall x, y \in \Delta.\ S(x) \lor \neg P(y,x)) \land$$
$$(|P| = |\Delta|),$$

where $w^-(R) = w^-(S) = -1$.

Since FASTWFOMC does not support many-sorted logic, our experiments are limited to formulas with a single domain. However, many of the counting problems used in the experimental evaluation of CRANE (Dilkas and Belle 2023) become equivalent (in terms of generating equivalent integer sequences) when restricted to a single domain. Additionally, comparing CRANE2 and FASTWFOMC on a broader set of problems is challenging because each problem needs to be represented in two (quite different) logics: $\mathsf{FO}$ and $\mathsf{UFO}^2 + \mathsf{CC}$.

Nevertheless, the three benchmarks cover a wide range of possibilities. The 'friends' benchmark stands out as it uses multiple predicates and can be expressed in $\mathsf{FO}$ using just two variables without cardinality constraints or counting quantifiers. The 'functions' benchmark, on the other hand, can still be handled by all the algorithms, but it requires cardinality constraints, counting quantifiers, or more than two variables. Lastly, the 'bijections' benchmark is an example of a formula that FASTWFOMC can handle but FORCLIFT cannot.

A considerable difference between FORCLIFT and the other two algorithms is that FORCLIFT does not support arbitrary precision whereas the other algorithms use the GNU Multiple Precision Arithmetic Library. Thus, when the count exceeds finite precision, FORCLIFT returns $\infty$. For evaluation purposes, we ran each algorithm on each benchmark using domains of sizes $2^1, 2^2, 2^3$, and so on, until a timeout occurred after $1\,\mathrm{h}$, the algorithm ran out of memory, or FORCLIFT returned $\infty$. We separately measured compilation and inference time but primarily focused on total runtime, dominated by the latter.

The experiments were conducted using an Intel Skylake $2.4\,\mathrm{GHz}$ CPU with $188\,\mathrm{GiB}$ of memory and CentOS 7. C++ programs were compiled using the Intel C++ Compiler 2020u4. FASTWFOMC ran on Julia 1.10.4, while the other algorithms were executed on the Java Virtual Machine 1.8.0_201.

Figure 2 presents a summary of the experimental results. Only FASTWFOMC and CRANE2-BFS could handle the bijection-counting problem. For this benchmark, the largest domain sizes these algorithms could accommodate were 64 and 4096, respectively. On the other two benchmarks, FORCLIFT had the lowest runtime. However, due to its finite precision, it only scaled up to domain sizes of 16 and 128 for 'friends' and 'functions', respectively. FASTWFOMC

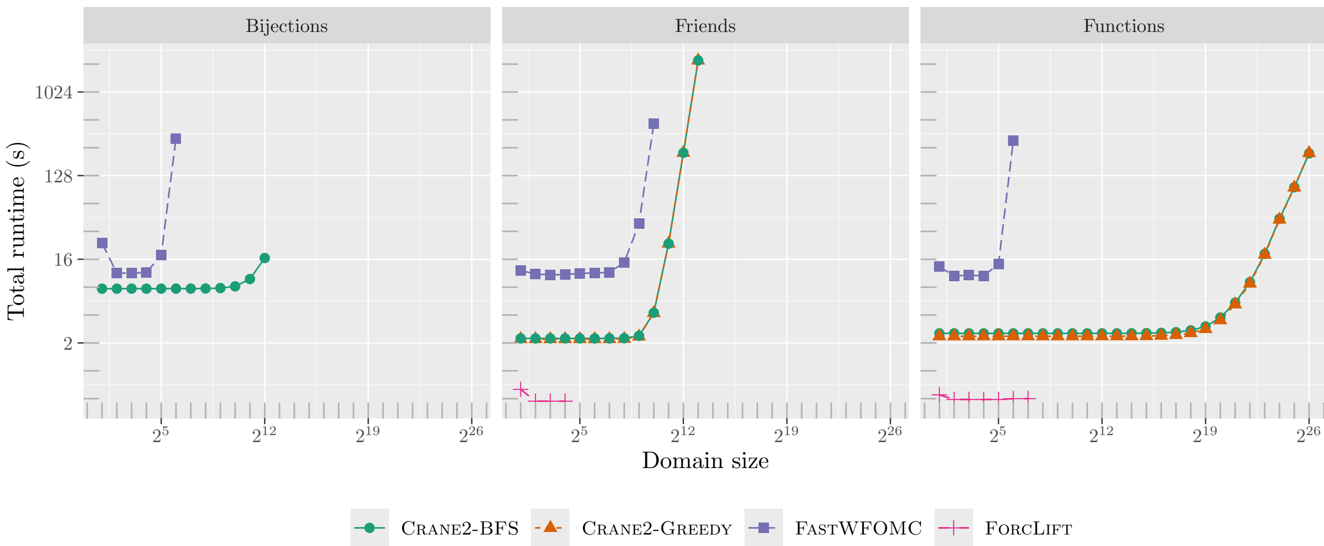


Figure 2: The runtime of the algorithms as a function of the domain size. Note that both axes are on a logarithmic scale.

outperformed ForcLift in the case of 'friends', but not 'functions', as it could handle domains of size 1024 and 64, respectively. Furthermore, both Crane2-BFS and Crane2-Greedy performed similarly on both benchmarks. Similarly to the 'bijections' benchmark, Crane2 significantly outperformed the other two algorithms, scaling up to domains of size 8192 and 67 108 864, respectively.

Another aspect of the experimental results that deserves separate discussion is compilation. Both Julia and Scala use just-in-time (JIT) compilation, which means that FastWFOMC and ForcLift take longer to run on the smallest domain size, where most JIT compilation occurs. In the case of Crane2, it is only run once per benchmark, so the JIT compilation time is included in its overall runtime across all domain sizes. Additionally, while ForcLift's compilation is generally faster than that of Crane2, neither significantly affects overall runtime. Specifically, ForcLift compilation typically takes around 0.5 s, while Crane2 compilation takes around 2.3 s.

Based on our experiments, which algorithm should be used in practice? If the formula can be handled by ForcLift and the domain sizes are reasonably small, ForcLift is likely the fastest algorithm. In other situations, Crane2 is expected to be significantly more efficient than FastWFOMC regardless of domain size, provided both algorithms can handle the formula.

## 7 Conclusion and Future Work

In this work, we have made several contributions. First, we have developed algorithmic techniques to find the base cases of recursive functions generated by Crane. Second, we have extended the smoothing procedure of ForcLift and Crane to support base case formulas. Third, we have proposed an approach to compile function definitions into C++ programs with support for arbitrary-precision arithmetic. Lastly, we have provided experimental evidence demonstrating that Crane2 can scale to much larger domain sizes than FastWFOMC while handling more formulas than ForcLift. Having FOMC algorithms that can efficiently handle large domain sizes is especially crucial in the weighted setting. For example, consider the 'friends' instance examined in Section 6, which models a social network with friendships, smoking habits, and the probability of having cancer. The utility of such a model would be significantly limited if probabilities could only be efficiently computed for networks of at most 1000 people.

There are many potential avenues for future work. Specifically, a more thorough experimental study is needed to understand how FOMC algorithms compare in terms of their ability to handle different formulas and their scalability with respect to domain size. Additionally, further characterisation of the capabilities of Crane2 can be explored. For example, *completeness* could be proven for a fragment of first-order logic such as $\mathsf{C}^2$ (using a suitable encoding of counting quantifiers). Moreover, the efficiency of a FOMC algorithm in handling a particular formula can be assessed using *fine-grained complexity*. In the case of Crane2, this can be done by analysing the equations (Dilkas and Belle 2023). By doing so, efficiency can be reasoned about in a more implementation-independent manner by making claims about the maximum degree of the polynomial that characterises any given solution.

## References

Azzolini, D.; and Riguzzi, F. 2023. Lifted inference for statistical statements in probabilistic answer set programming. *Int. J. Approx. Reason.*, 163: 109040.

Barvínek, J.; van Bremen, T.; Wang, Y.; Zelezný, F.; and Kuželka, O. 2021. Automatic Conjecturing of P-Recursions Using Lifted Inference. In *ILP*, volume 13191 of *Lecture Notes in Computer Science*, 17–25. Springer.

Beame, P.; Van den Broeck, G.; Gribkoff, E.; and Suciu, D. 2015. Symmetric Weighted First-Order Model Counting. In *PODS*, 313–328. ACM.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7): 772–799.

Darwiche, A. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2): 11–34.

Dilkas, P.; and Belle, V. 2023. Synthesising Recursive Functions for First-Order Model Counting: Challenges, Progress, and Conjectures. In *KR*, 198–207.

Gogate, V.; and Domingos, P. M. 2016. Probabilistic theorem proving. *Commun. ACM*, 59(7): 107–115.

Gribkoff, E.; Suciu, D.; and Van den Broeck, G. 2014. Lifted Probabilistic Inference: A Guide for the Database Researcher. *IEEE Data Eng. Bull.*, 37(3): 6–17.

Hinman, P. G. 2018. *Fundamentals of mathematical logic*. CRC Press.

Hodges, W. 1997. *A Shorter Model Theory*. Cambridge University Press.

Jaeger, M.; and Van den Broeck, G. 2012. Liftability of Probabilistic Inference: Upper and Lower Bounds. In *StarAI@UAI*.

Kazemi, S. M.; Kimmig, A.; Van den Broeck, G.; and Poole, D. 2016. New Liftable Classes for First-Order Probabilistic Inference. In *NIPS*, 3117–3125.

Kazemi, S. M.; and Poole, D. 2016. Knowledge Compilation for Lifted Probabilistic Inference: Compiling to a Low-Level Language. In *KR*, 561–564. AAAI Press.

Kersting, K. 2012. Lifted Probabilistic Inference. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 33–38. IOS Press.

Kuželka, O. 2021. Weighted First-Order Model Counting in the Two-Variable Fragment With Counting Quantifiers. *J. Artif. Intell. Res.*, 70: 1281–1307.

Malhotra, S.; and Serafini, L. 2022. Weighted Model Counting in FO2 with Cardinality Constraints and Counting Quantifiers: A Closed Form Formula. In *AAAI*, 5817–5824. AAAI Press.

Nilsson, N. J. 1986. Probabilistic Logic. *Artif. Intell.*, 28(1): 71–87.

Niu, F.; Ré, C.; Doan, A.; and Shavlik, J. W. 2011. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. *Proc. VLDB Endow.*, 4(6): 373–384.

Novák, V.; Perfilieva, I.; and Mockor, J. 2012. *Mathematical principles of fuzzy logic*, volume 517. Springer Science & Business Media.

Riguzzi, F.; Bellodi, E.; Zese, R.; Cota, G.; and Lamma, E. 2017. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *Int. J. Approx. Reason.*, 80: 313–333.

Russell, S.; and Norvig, P. 2020. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

Šaletić, D. Z. 2024. Graded Logics. *Interdisciplinary Description of Complex Systems: INDECS*, 22(3): 276–295.

Singla, P.; and Domingos, P. M. 2008. Lifted First-Order Belief Propagation. In *AAAI*, 1094–1099. AAAI Press.

Svatos, M.; Jung, P.; Tóth, J.; Wang, Y.; and Kuželka, O. 2023. On Discovering Interesting Combinatorial Integer Sequences. In *IJCAI*, 3338–3346. ijcai.org.

Tóth, J.; and Kuželka, O. 2023. Lifted Inference with Linear Order Axiom. In *AAAI*, 12295–12304. AAAI Press.

Totis, P.; Davis, J.; De Raedt, L.; and Kimmig, A. 2023. Lifted Reasoning for Combinatorial Counting. *J. Artif. Intell. Res.*, 76: 1–58.

Tóth, J.; and Kuželka, O. 2024. Complexity of Weighted First-Order Model Counting in the Two-Variable Fragment with Counting Quantifiers: A Bound to Beat. arXiv:2404.12905.

van Bremen, T.; and Kuželka, O. 2020. Approximate Weighted First-Order Model Counting: Exploiting Fast Approximate Model Counters and Symmetry. In *IJCAI*, 4252–4258. ijcai.org.

van Bremen, T.; and Kuželka, O. 2021. Faster lifting for two-variable logic using cell graphs. In *UAI*, volume 161 of *Proceedings of Machine Learning Research*, 1393–1402. AUAI Press.

van Bremen, T.; and Kuželka, O. 2023. Lifted inference with tree axioms. *Artif. Intell.*, 324: 103997.

Van den Broeck, G. 2011. On the Completeness of First-Order Knowledge Compilation for Lifted Probabilistic Inference. In *NIPS*, 1386–1394.

Van den Broeck, G.; Choi, A.; and Darwiche, A. 2012. Lifted Relax, Compensate and then Recover: From Approximate to Exact Lifted Probabilistic Inference. In *UAI*, 131–141. AUAI Press.

Van den Broeck, G.; Meert, W.; and Darwiche, A. 2014. Skolemization for Weighted First-Order Model Counting. In *KR*. AAAI Press.

Van den Broeck, G.; Taghipour, N.; Meert, W.; Davis, J.; and De Raedt, L. 2011. Lifted Probabilistic Inference by First-Order Knowledge Compilation. In *IJCAI*, 2178–2185. IJCAI/AAAI.

Venugopal, D.; Sarkhel, S.; and Gogate, V. 2015. Just Count the Satisfied Groundings: Scalable Local-Search and Sampling Based Inference in MLNs. In *AAAI*, 3606–3612. AAAI Press.

Wang, Y.; Pu, J.; Wang, Y.; and Kuželka, O. 2023. On Exact Sampling in the Two-Variable Fragment of First-Order Logic. In *LICS*, 1–13.

Wang, Y.; van Bremen, T.; Wang, Y.; and Kuželka, O. 2022. Domain-Lifted Sampling for Universal Two-Variable Logic and Extensions. In *AAAI*, 10070–10079. AAAI Press.