# 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. Recently, a new algorithm for FOMC was proposed 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. However, the algorithm did not include base cases in the recursive definitions. This work makes three contributions. First, we demonstrate how to construct function definitions that include base cases by modifying the logical formulas used in the FOMC algorithm. Second, we extend the well-known circuit modification technique in knowledge compilation, known as smoothing, to work with the formulas corresponding to base cases and the recently proposed node types, such as domain recursion and constraint removal. Third, 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 9000 times larger than the current state of the art, as demonstrated through experimental results.

## 1 Introduction

*First-order model counting* is the task of counting the number of models of a sentence in first-order logic over some given domain(s). The (symmetric) weighted variant of this problem, known as WFOMC, seeks to calculate the sum of model weights. In WFOMC, the weight of a model is determined by predicate weights (Van den Broeck et al. 2011). WFOMC is a key approach to *lifted (probabilistic) inference*, which aims to compute probabilities more efficiently by leveraging symmetries inherent in the problem description (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). By considering domains of increasing sizes, the model count of a formula can be seen as an integer sequence. WFOMC algorithms have been utilised for discovering new sequences (Svatos et al. 2023) as well as conjecturing (Barvínek et al. 2021) and constructing (Dilkas and Belle 2023) recurrence relations and other recursive structures that describe these sequences. Additionally, WFOMC algorithms have been extended to perform *sampling* (Wang et al. 2022, 2023).

The complexity of WFOMC is typically characterised in terms of *data complexity*. This involves fixing the formula and determining whether an algorithm exists that can compute the WFOMC in time polynomial with respect to the domain size(s). If such an algorithm exists, the 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 all 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 various WFOMC algorithms with different underlying principles. Perhaps the most prominent class of WFOMC algorithms is based on *first-order knowledge compilation*. In this approach, the formula is compiled into a representation (such as a circuit or graph) by iteratively applying *compilation rules*. This representation can then be used to compute the WFOMC for any combination of domain sizes (or weights). Algorithms in this class include FORCLIFT (Van den Broeck et al. 2011) and its recent extension CRANE (Dilkas and Belle 2023). The former compiles formulas into circuits, while the latter compiles them first to *first-order computational graphs* (FCGs) and then to (algebraic) equations. Another WFOMC 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, while capable of handling formulas beyond the reach of FORCLIFT, was incomplete as it could only construct function definitions, which would then need to be evaluated to compute the WFOMC. Additionally, recursive functions were presented without base cases. In this work, we introduce CRANE2, an extension of CRANE that addresses these two weaknesses.

Figure 1 outlines the workflow of the new algorithm. In

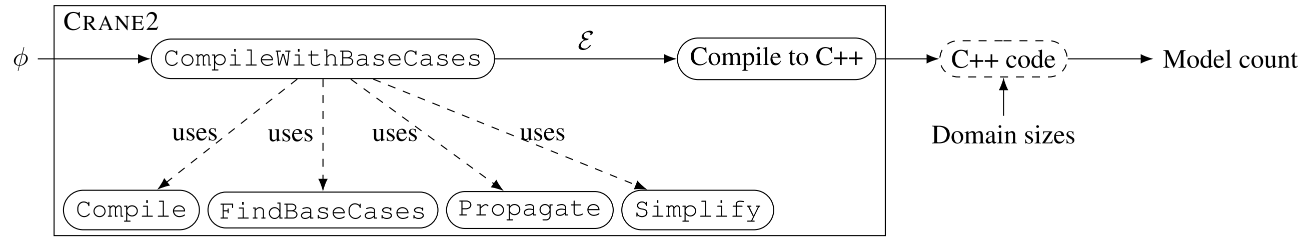


Figure 1: The outline of using CRANE2 to compute the model count of a formula $\phi$. First, the formula is compiled into a set of equations, which are then used to create a C++ program. This program can be executed with different command line arguments to calculate the model count of $\phi$ for different domain sizes. To accomplish this, the `CompileWithBaseCases` function employs several components: (i) the knowledge compilation algorithm of CRANE, referred to as `Compile`, (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`).

Section 3, we describe how `CompileWithBaseCases` finds the base cases for recursive functions by: (i) identifying a sufficient set of base cases for each function, (ii) constructing formulas corresponding to these base cases, and (iii) recursing on these new formulas. Then, Section 4 explains post-processing techniques for FCGs and the formulas from Step (ii) above required to preserve the correct model count. Next, Section 5 elucidates how function definitions encoding a solution to a WFOMC problem are compiled into C++ programs.[1] Finally, Section 6 presents two experiments comparing CRANE2 with other state-of-the-art WFOMC algorithms.

## 2 Preliminaries

In Section 2.1, we provide a summary of the basic principles of first-order logic. Then, in Section 2.2, we formally define WFOMC and discuss the distinctions between three variations of first-order logic that are utilised for WFOMC. 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 first-order knowledge compilation algorithms. There are two key differences between the logic used within such algorithms and the logic supported as input. First, Skolemization (Van den Broeck, Meert, and Darwiche 2014) is employed to eliminate existential quantifiers by introducing additional predicates. Second, the input formula is rewritten as a conjunction of clauses, each of which is in *prenex normal form* (Hinman 2018).

A *formula* is a conjunction of clauses. 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. A *literal* is either an atom (i.e., a *positive* literal) or its negation (i.e., a *negative* literal). An *atom* is 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 $y$. An atom is *ground* if it contains no variables, i.e., only constants. The *arity* of a predicate is the number of arguments it takes, i.e., $m$ in the case of predicate $P$ mentioned above. When we want to denote a predicate together with its arity, we write $P/m$. A *term* is either a variable or a constant. 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. Moreover, for readability, clauses written on separate lines are implicitly conjoined.

### 2.2 WFOMC Algorithms and Their Logics

In Table 1, we outline the differences among three first-order logics commonly used in WFOMC: (i) $\mathsf{FO}$ is the input format for FORCLIFT[2] and its extensions CRANE[3] and CRANE2; (ii) $\mathsf{C}^2$ is often used in the literature on FASTWFOMC and related methods (Kuželka 2021; Malhotra and Serafini 2022); (iii) $\mathsf{UFO}^2 + \mathsf{EQ}$ is the input format supported by a private version of FASTWFOMC obtained directly from the authors. Note that the publicly available version[4] of FASTWFOMC does not support any cardinality constraints. The notation we use to refer to each logic is standard in the case of $\mathsf{C}^2$, new in the case of $\mathsf{UFO}^2 + \mathsf{EQ}$, 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.

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. Most of these assignments to sorts are typically left implicit and can be reconstructed from the

[1]The possibility of compiling WFOMC solutions to C++ programs has been previously considered (Kazemi and Poole 2016). However, the extent of formulas that could be handled was limited.

[2]https://github.com/UCLA-StarAI/Forclift

[3]https://doi.org/10.5281/zenodo.8004077

[4]https://comp.nus.edu.sg/~tvanbr/software/fastwfomc.tar.gz

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

Table 1: Comparison of three logics used in WFOMC based on the following aspects: (i) number of sorts, (ii) support for constants, (iii) 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 $n$-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.

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.

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

$\mathsf{C}^2$ and $\mathsf{UFO}^2 + \mathsf{EQ}$ are single-sorted (i.e., all variables are quantified over the same domain) and only include formulas with at most two variables and no constants. Unlike $\mathsf{UFO}^2 + \mathsf{EQ}$, $\mathsf{C}^2$ supports existential quantifiers, including *counting quantifiers*. For example, $\exists^{=1}x.\ \phi(x)$ means that there exists *exactly one* $x$ such that $\phi(x)$, and $\exists^{\le 2}x.\ \phi(x)$ means that there exist *at most two* such $x$. The advantage $\mathsf{UFO}^2 + \mathsf{EQ}$ brings over $\mathsf{C}^2$ is the support for *(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 (which may not be distinct). 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. Then a *structure* of $\phi$ (with respect to $\sigma$) is a set $M$ of ground literals defined by adding 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 $\phi$.

**Definition 2** (WFOMC (Van den Broeck et al. 2011))**.** Continuing from Definition 1, for each predicate $P$ in $\phi$, let $w^+(P)$ and $w^-(P)$ be its *weights* in $\mathbb{Q}$. Unless specified otherwise, we assume all weights are equal to 1. The *(symmetric) weighted first-order model count* (WFOMC) of $\phi$ (with respect to $\sigma$, $w^+$, and $w^-$) is:

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

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

**Example 1** (Counting functions)**.** To define predicate $P$ as an endofunction on $\Delta$, in $\mathsf{C}^2$ one would write $\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$. In $\mathsf{UFO}^2 + \mathsf{EQ}$, the same could be written as

$$\begin{gathered} \forall x, y \in \Delta.\ S(x) \lor \neg P(x, y) \\ |P| = |\Delta|, \end{gathered} \tag{1}$$

where $w^-(S) = -1$. Although Formula (1) has more models compared to its counterpart in $\mathsf{C}^2$, the weight function makes the *weighted* model count of Formula (1) equal to the number of endofunctions on $\Delta$.

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

$$\begin{gathered} \forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y) \\ \forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \land P(x, z) \Rightarrow y = z. \end{gathered} \tag{2}$$

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 (2) is written with two domains instead of just one. However, we can still determine the number of endofunctions 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 first-order knowledge compilation 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 compared to 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 3** (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.

## 3 Completing the Definitions of Recursive Functions

Algorithm 1 presents our overall approach for compiling a formula into equations that include the necessary base cases.

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

```
Input: formula $\phi$
Output: set $\mathcal{E}$ of equations
1 $(\mathcal{E}, \mathcal{F}, \mathcal{D}) \leftarrow$ Compile($\phi$);
2 $\mathcal{E} \leftarrow$ Simplify($\mathcal{E}$);
3 foreach base case $f(\mathbf{x}) \in$ FindBaseCases($\mathcal{E}$) do
4     $\psi \leftarrow \mathcal{F}(f)$;
5     foreach $i$ such that $x_i \in \mathbb{N}_0$ do
6         $\psi \leftarrow$ Propagate($\psi$, $\mathcal{D}(f, i)$, $x_i$);
7     $\mathcal{E} \leftarrow \mathcal{E} \cup$ CompileWithBaseCases($\psi$);
```

To begin, we utilise the knowledge compilation algorithm from the original CRANE to compile the formula into three components: (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. After some algebraic simplification, $\mathcal{E}$ is then passed to the `FindBaseCases` procedure (explained in Section 3.1), which identifies the base cases that need to be defined. For each base case $f(\mathbf{x})$, we determine the formula associated with the function name $f$ and simplify it using the `Propagate` procedure (explained in Section 3.2). The algorithm then calls itself on these simplified formulas and adds the resulting base case equations to $\mathcal{E}$. Example 2 provides a more detailed explanation of Algorithm 1.

**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 $\Delta$:

$$\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y)$$
$$\forall y \in \Delta.\ \exists x \in \Gamma.\ P(x, y)$$
$$\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \land P(x, z) \Rightarrow y = z$$
$$\forall x, z \in \Gamma.\ \forall y \in \Delta.\ P(x, y) \land P(z, y) \Rightarrow x = z.$$

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 introduced by `Compile`. 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

**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 equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$ do
3     foreach function call $f(\mathbf{y}) \in \texttt{expr}$ do
4         foreach $y_i \in \mathbf{y}$ do
5             if $y_i \in \mathbb{N}_0$ then
6                 $\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto y_i] \}$;
7             else if $y_i = x_i - c_i$ for some $c_i \in \mathbb{N}_0$ then
8                 for $j \leftarrow 0$ to $c_i - 1$ do
9                     $\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto j] \}$;
```

$\phi$ be a formula with two domains $\Gamma$ and $\Delta$, and let $n, m \in \mathbb{N}_0$. Then the WFOMC of `Propagate(`$\phi$, $\Delta$, $n$`)` assuming $|\Gamma| = m$ is the same as the WFOMC of `Propagate(`$\phi$, $\Gamma$, $m$`)` assuming $|\Delta| = n$.

Finally, we highlight the crucial role of the `Simplify` procedure in simplifying 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) $[a \le m \le b] = \begin{cases} 1 & \text{if } a \le m \le b \\ 0 & \text{otherwise} \end{cases}$ represents the Iverson bracket. `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 the simplified form mentioned above.

## 3.1 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of the `FindBaseCases` function. When a function $f$ calls itself recursively, `FindBaseCases` considers two types of arguments: (i) constants and (ii) arguments of the form $x_i - c_i$, where $c_i$ is a constant and $x_i$ is the $i$-th argument of the signature of $f$. If 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 zero up to (but not including) $c_i$. The following discussion explains the reasoning behind this approach.

**Example 3.** Let us consider the recursive function $g$ from Example 2. In this case, `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 $g(l, 0)$.

For the rest of this section, let $\mathcal{E}$ represent the equations returned by `CompileWithBaseCases`. To demonstrate that the base cases identified by `FindBaseCases` are sufficient, we begin with a few observations that stem from the details of previous work (Van den Broeck et al. 2011; Dilkas and Belle 2023) and this work.

**Observation 1.** *For each function $f$, there is precisely one equation $e \in \mathcal{E}$ with $f(\mathbf{x})$ on the left-hand side where all $x_i$'s are variables (i.e., $e$ is not a base case). We refer to $e$ as the* definition *of $f$.*

**Observation 2.** *There is a* topological ordering *of all functions $(f_i)_i$ in $\mathcal{E}$ such that equations in $\mathcal{E}$ with $f_i$ on the left-hand side do not contain function calls to $f_j$ with $j > i$. This condition prevents mutual recursion and other cyclic scenarios.*

**Observation 3.** *For every equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, the evaluation of* `expr` *terminates when provided with the values of all relevant function calls.*

**Corollary 1.** *If $f$ is a non-recursive function with no function calls on the right-hand side of its definition, then the evaluation of any function call $f(\mathbf{x})$ terminates.*

**Observation 4.** *For any equation $f(\mathbf{x}) = \texttt{expr}$, if $\mathbf{x}$ contains only constants, then* `expr` *cannot include any function calls to $f$.*

Additionally, we introduce an assumption about the structure of recursion.

**Assumption 1.** *For every equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, every recursive function call $f(\mathbf{y}) \in \texttt{expr}$ satisfies the following:*

- *Each $y_i$ is either $x_i - c_i$ or $c_i$ for some constant $c_i$.*
- *There exists $i$ such that $y_i = x_i - c_i$ for some $c_i > 0$.*

Finally, we assume a particular order of evaluation for function calls using the equations in $\mathcal{E}$. Specifically, we assume that base cases are considered before the recursive definition. The exact order in which base cases are considered is not relevant.

**Assumption 2.** *When multiple equations in $\mathcal{E}$ match a function call $f(\mathbf{x})$, preference is given to an equation with the most constants on its left-hand side.*

With the facts and assumptions mentioned above, we prove the following theorem.

**Theorem 1** (Termination)**.** *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 on 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 (as defined in Observation 2). For readability, we divide the proof into several lemmas of increasing generality.

**Lemma 1.** *Assume that $\mathcal{E}$ consists of just* one unary *function $f$. Then the evaluation of a function call $f(x)$ terminates for any $x \in \mathbb{N}_0$.*

*Proof.* If $f(x)$ is captured by a base case, then its evaluation terminates by Corollary 1 and Observation 4. If $f$ is not recursive, the evaluation of $f(x)$ terminates by Corollary 1.

Otherwise, let $f(y)$ be an arbitrary function call on the right-hand side of the definition of $f(x)$. If $y$ is a constant, there is a base case for $f(y)$. Otherwise, let $y = x - c$ for some $c > 0$. Then there exists $k \in \mathbb{N}_0$ such that $0 \leq x - kc \leq c - 1$. So, after $k$ iterations, the sequence of function calls $f(x), f(x-c), f(x-2c), \ldots$ will be captured by the base case $f(x \mod c)$. □

**Lemma 2.** *Generalising Lemma 1, let $\mathcal{E}$ be a set of equations for* one *$n$-ary function $f$ for some $n \geq 1$. Then the evaluation of $f(\mathbf{x})$ terminates for any $\mathbf{x} \in \mathbb{N}_0^n$.*

**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 $\Delta \not\in \mathrm{Doms}(C)$ then $\phi' \leftarrow \phi' \cup \{ C \}$;
5          else
6              $C' \leftarrow \{ l \in C \mid \Delta \not\in \mathrm{Doms}(l) \}$;
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             $\phi' \leftarrow \phi' \cup \{ C[x_1 \mapsto c_1, \ldots, x_m \mapsto c_m] \mid$
                   $(c_i)_{i=1}^m \in D^m \}$;
```

*Proof.* If $f$ is non-recursive, the evaluation of $f(\mathbf{x})$ terminates by previous arguments. We proceed by induction on $n$, with the base case of $n = 1$ handled by Lemma 1. Assume that $n > 1$. Any base case of $f$ can be seen as a function of arity $n - 1$, since one of the parameters is fixed. Thus, the evaluation of any base case terminates by the inductive hypothesis. It remains to show that the evaluation of the recursive equation for $f$ terminates, but that follows from Observation 3. □

*Proof of Theorem 1.* We proceed by induction on the number of functions $n$. The base case of $n = 1$ is handled by Lemma 2. Let $(f_i)_{i=1}^n$ be some topological ordering of these $n > 1$ functions. If $f = f_j$ for $j < n$, then the evaluation of $f(\mathbf{x})$ terminates by the inductive hypothesis since $f_j$ cannot call $f_n$ by Observation 2. Using the inductive hypothesis that all function calls to $f_j$ (with $j < n$) terminate, the proof proceeds similarly to the Proof of Lemma 2. □

## 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 $n > 0$[5], partial grounding is performed by replacing all variables quantified over $\Delta$ with constants. 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

[5]None of the formulas examined in this study had $n > 1$.

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 4.1.

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:

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

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

## 4 Smoothing

*Smoothness* is a property originating in propositional knowledge compilation, where it refers to the situation where all disjuncts in a disjunction node contain the same atoms (Darwiche 2001). Van den Broeck et al. (2011) extend this concept to first-order logic, introducing set-disjunction and inclusion-exclusion nodes in addition to disjunction.

The purpose of smoothing is as follows. When compilation rules such as unit propagation and inclusion-exclusion simplify a formula, certain ground atoms may be eliminated (see Example 5 below). To properly account for these atoms during counting, smoothing nodes (i.e., tautological clauses such as $P(c) \lor \neg P(c)$) are added to the FCG in the appropriate location.

The remainder of this section presents an extension to the smoothing algorithm of Van den Broeck et al. (2011). Section 4.1 explains the role of smoothing in the base-case-finding algorithm described in Section 3. Section 4.2 demonstrates how to adapt smoothing to the compilation rules introduced by Dilkas and Belle (2023).

### 4.1 Smoothing for Base Cases

In this section, we motivate and describe lines 8 and 9 of Algorithm 3. Suppose that `Propagate` is called with arguments $(\phi, \Delta, 0)$, which means we are simplifying the formula $\phi$ by assuming that the domain $\Delta$ is empty. Informally, if there is a predicate $P$ in $\phi$ that has nothing to do with the domain $\Delta$, smoothing preserves all occurrences of $P$ even if all clauses with $P$ become vacuously satisfied. It is important to note that the approach presented in this section is not unique. We explain it via the example below.

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

$$\forall x \in \Delta.\, \forall y, z \in \Gamma.\, Q(x) \lor P(y, z) \tag{3}$$
$$\forall y, z \in \Gamma'.\, P(y, z), \tag{4}$$

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 (4), but it is allowed to vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since Clause (3) is vacuously satisfied by all structures. Therefore, the correct WFOMC 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:

$$\begin{gathered}
\forall y, z \in \Gamma.\, P(y, z) \lor \neg P(y, z) \\
\forall y, z \in \Gamma'.\, P(y, z),
\end{gathered}$$

which retains the correct model count.

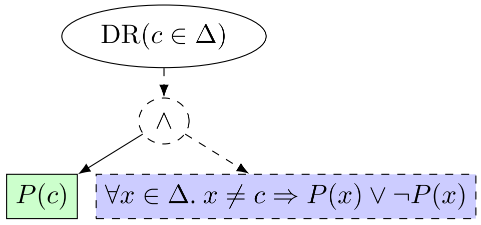
It is worth mentioning that the choice of $l$ on line 8 of Algorithm 3 is inconsequential because any choice achieves the same goal: constructing a tautological clause that retains the literals in $C'$.

### 4.2 Smoothing the FCG

Smoothing is a two-step process that involves propagating unit clauses upwards (i.e., in the opposite direction of FCG arcs) and adding smoothing nodes to account for missing atoms. In this section, we will (i) describe the relevant node types from previous work, (ii) explain how smoothing should work for these node types, and (iii) demonstrate the new smoothing techniques using two example FCGs.

Before discussing the proposed changes to smoothing, let us briefly review the compilation rules and their corresponding node types introduced by Dilkas and Belle (2023). *Domain recursion* involves selecting a domain $\Delta$, introducing a new constant $c \in \Delta$, and modifying the formula based on two possibilities for each variable $x$ quantified over $\Delta$: $x = c$ or $x \neq c$. The resulting node is denoted as $\mathrm{DR}(c \in \Delta)$. *Constraint removal* applies to formulas where each variable $x$ quantified over a domain $\Delta$ is followed by the inequality constraint $x \neq c$. These formulas can be rewritten so that each clause begins with $\forall x \in \Delta.\, x \neq c \Rightarrow \ldots$ Constraint removal replaces $\Delta$ with a new domain $\Delta'$ and removes the $x \neq c$ constraints. The resulting node is denoted as $\mathrm{CR}(\Delta' \leftarrow \Delta \setminus \{\, c \,\})$. Finally, *caching* detects when the input formula $\phi$ is equal to a previously encountered formula $\psi$ except for having different domains. The resulting node is denoted as $\mathrm{Ref}(\sigma)$, with $\sigma$ representing the substitution mapping the domains of $\psi$ to their corresponding domains in $\phi$.

**Stage 1 for Domain Recursion** When visiting a $\mathrm{DR}(c \in \Delta)$ node, we replace each occurrence of $\phi(c)$ or $\forall x \in \Delta.\, x \neq c \Rightarrow \phi(x)$ with $\forall x \in \Delta.\, \phi(x)$ for each unit clause received from the child node. This way, we ensure that if the subgraph below the domain recursion node covers some of the ground



DR($c \in \Delta$)
$\wedge$
$P(c)$
$\forall x \in \Delta.\, x \neq c \Rightarrow P(x) \vee \neg P(x)$

Figure 2: An artificial example of an FCG where a smoothing node needs to be added below a domain recursion node. The dashed nodes and arcs are added during Stage 2 of smoothing.

atoms affected by domain recursion, it covers all of them. If the relevant subgraph already covers those ground atoms, Stage 2 will not make any changes. Otherwise, smoothing nodes will be added below the domain recursion node to account for the difference in the sets of ground atoms assigned to the domain recursion node and its child node.

**Stage 1 for Constraint Removal and Caching** When visiting a CR($\Delta' \leftarrow \Delta \setminus \{ c \}$) node, we reverse constraint removal by replacing each $\forall x \in \Delta'.\, \phi(x)$ with $\forall x \in \Delta.\, x \neq c \Rightarrow \phi(x)$. When visiting a $\mathrm{Ref}(\sigma)$ node, we apply the substitution $\sigma$ to the unit clauses from the child node.

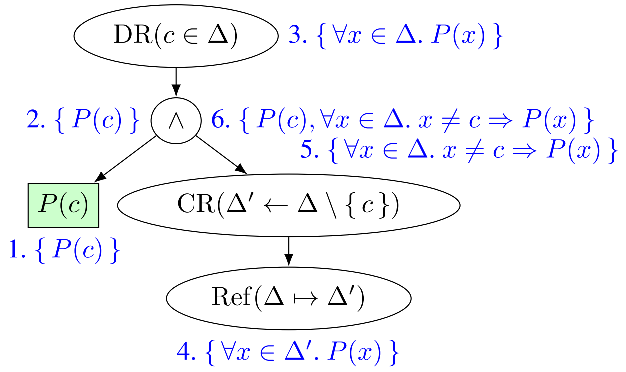
**Stage 2 for Domain Recursion** We do not need to add smoothing nodes immediately below constraint removal or caching nodes. However, for domain recursion, we follow a specific process.

1. If the set of unit clauses assigned to the child node during Stage 1 contains both $\phi(c)$ and $\forall x \in \Delta.\, x \neq c \Rightarrow \phi(x)$, we merge the two clauses into $\forall x \in \Delta.\, \phi(x)$. The only difference between these two clauses is that one has the constant $c$ while the other has a variable $x \neq c$.
2. If necessary, we add smoothing nodes below the domain recursion node to account for the difference between the unit clauses assigned to the domain recursion node during Stage 1 and the unit clauses of the child node post-processed by the step above.

**Example 6** (An FCG that requires smoothing)**.** Figure 2 depicts an FCG with a domain recursion node DR($c \in \Delta$) followed directly by a single $P(c)$ node. In this scenario, Stage 1 assigns $\{ \forall x \in \Delta.\, P(x) \}$ to the former node and $\{ P(c) \}$ to the latter. As these two sets of unit clauses cover different ground atoms, Stage 2 adds a smoothing node to address $P(x)$ for all $x \in \Delta \setminus \{ c \}$.

**Example 7** (A smooth FCG)**.** Stage 1 of the smoothing process is more complex in the case of the FCG shown in Figure 3. The unit clause $P(c)$ propagates to the conjunction node and is then generalised to $\forall x \in \Delta.\, P(x)$ by the domain recursion node. It continues to propagate to the caching node, changing its form to $(\forall x \in \Delta.\, P(x))[\Delta \mapsto \Delta'] \equiv \forall x \in \Delta'.\, P(x)$. The constraint removal node re-introduces the constraints, transforming the clause to $\forall x \in \Delta.\, x \neq c \Rightarrow P(x)$, which then propagates to the conjunction node, joining $P(c)$.

During Stage 2, $P(c)$ and $\forall x \in \Delta.\, x \neq c \Rightarrow P(x)$ are combined into $\forall x \in \Delta.\, P(x)$. Since the resulting clause



DR($c \in \Delta$)
3. $\{ \forall x \in \Delta.\, P(x) \}$
2. $\{ P(c) \}$
$\wedge$
6. $\{ P(c), \forall x \in \Delta.\, x \neq c \Rightarrow P(x) \}$
5. $\{ \forall x \in \Delta.\, x \neq c \Rightarrow P(x) \}$
$P(c)$
CR($\Delta' \leftarrow \Delta \setminus \{ c \}$)
1. $\{ P(c) \}$
Ref($\Delta \mapsto \Delta'$)
4. $\{ \forall x \in \Delta'.\, P(x) \}$

Figure 3: A smooth FCG based on Example 2. For compactness, the arc between the caching node and the domain recursion node is not shown. The labels next to the nodes indicate the sets of unit clauses assigned to each node during Stage 1 and the order in which these assignments were made. Empty assignments that replace a set $S$ with $S$ itself are not included.

matches the clause assigned to the domain recursion node, the FCG is already smooth.

The algebraic interpretation of Figure 3 is an equation $e$ that defines a recursive function. The right-hand side of $e$ already encompasses $P(c)$, and the smoothing algorithm correctly recognises that the recursive call also encompasses $P(x)$ for all $x \in \Delta \setminus \{ c \}$.

## 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 3), 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 $\mathtt{expr}$, store the result in the cache, and return it.

## 6 Experimental Evaluation

We compared CRANE2[6] with FASTWFOMC and FORCLIFT on two problems previously considered by Dilkas

[6]CRANE has two modes that decide how compilation rules are applied to formulas: one that uses greedy search and another that combines greedy and breadth-first search. In our experiments, we refer to these modes as CRANE2-GREEDY and CRANE2-BFS,

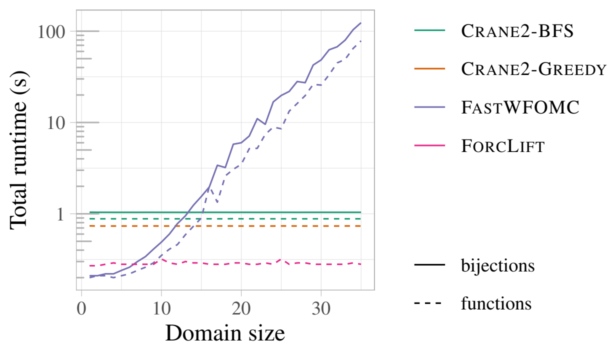


Figure 4: The running time of WFOMC algorithms. Note that the $y$-axis is on a logarithmic scale.

and Belle (2023). The first problem is the function-counting problem from Example 1, and the second problem is the bijection-counting problem described below. It is important to note that comparing CRANE2 and FASTWFOMC on a more substantial set of benchmarks is challenging because there is no automated way to translate a formula in FO or $\mathsf{C}^2$ into $\mathsf{UFO}^2 + \mathsf{EQ}$, or even check if such an encoding is possible. We ran the experiments on an AMD Ryzen 7 5800H processor with 16 GiB of memory and Arch Linux 6.8.2-arch2-1 operating system. FASTWFOMC was executed using Python 3.8.19 with Python-FLINT 0.5.0.

The FO formula for bijections is described in Example 2. The equivalent formula in $\mathsf{C}^2$ is

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

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

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

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

In the first experiment, we ran each of the four algorithms once on each combination of benchmark and domain size, ranging from one up to and including 35. Figure 4 shows the total runtime values as a measure of the overall performance of each algorithm. We also separately tracked compilation and inference times and commented on them where relevant. It is worth noting that CRANE2-BFS can handle more instances than either FORCLIFT or CRANE2-GREEDY, including the bijection-counting problem in our experiments as well as similar formulas examined previously (Dilkas and Belle 2023).

As shown in Figure 4, the runtimes of all compilation-based algorithms remained practically constant, in contrast to the rapidly increasing runtimes of FASTWFOMC. Although the search/compilation part is slower in CRANE2 than in FORCLIFT, the difference is negligible. The runtimes

respectively.

| Algorithm | Bijections | Functions |
|---|---|---|
| CRANE2-BFS | $10^4$ | $3 \times 10^5$ |
| CRANE2-GREEDY | — | $3 \times 10^5$ |
| FASTWFOMC | 28 | 32 |
| FORCLIFT | — | 143 |

Table 2: The maximum domain sizes that each algorithm can handle within 45 s. A dash symbol '—' indicates that the algorithm was unable to find a solution. The domain sizes for CRANE2 are accurate up to the first digit, while the domain sizes for the other domain sizes are exact.

of the knowledge compilation algorithms appear constant because, for these counting problems and domain sizes, compilation time dominates inference time (remember that compilation time is independent of domain sizes). Indeed, the maximum inference time of CRANE2-BFS and CRANE2-GREEDY across these experiments is only 4 ms. The runtimes of CRANE2 have lower variation than those of FORCLIFT because we compile the formula anew for each domain size with FORCLIFT, whereas with CRANE2, we compile it once and reuse the resulting C++ program for all domain sizes.

For the second experiment, we examined the maximum domain size the algorithms can handle in at most 45 s of total runtime. As shown in Table 2, CRANE2-BFS can scale to domain sizes 357 and 9375 times larger than FASTWFOMC for the bijection-counting and function-counting problems, respectively. It is important to note that while FORCLIFT may have runtime performance comparable to CRANE2, its finite-precision arithmetic cannot represent model counts for domain sizes greater than 143 and returns $\infty$ instead.

## 7 Conclusion and Future Work

In this work, we have presented several contributions. First, we have developed algorithmic techniques to find the base cases of recursive functions generated by the original CRANE algorithm. Second, we have extended the smoothing procedure of FORCLIFT and CRANE to support base case formulas and the compilation rules introduced by Dilkas and Belle (2023). 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.

There are many potential avenues for future work. Specifically, a more thorough experimental study is needed to understand how WFOMC 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 WFOMC algorithm in handling a particular formula can be assessed using *fine-grained complexity*. In the case of CRANE and 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.; Zeleznỳ, 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.

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.

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.

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.

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.

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.

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.; 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.