Towards Practical First-Order Model Counting

Ananth K. Kidambi[1*] , **Guramrit Singh**[1] , **Paulius Dilkas**[2] , **Kuldeep S. Meel**[3]
[1]Indian Institute of Technology Bombay, Mumbai, India
[2]National University of Singapore, Singapore, Singapore
[3]University of Toronto, Toronto, Canada
{210051002, 210050061}@iitb.ac.in, paulius.dilkas@nus.edu.sg, meel@cs.toronto.edu

Abstract

TODO: at most 200 words

1 Introduction

- 9 pages!!!
- Add some papers mentioned in:
  - very recent work
  - my previous paper, including:
    - other liftable fragments
    - some more theory papers, e.g., LICS 2018

Papers To Cite

- overviews
  - lifted probabilistic inference (Kersting 2012)
  - recent overview paper (Kuželka 2023)
- Alternative definition (Gogate and Domingos 2016)
- relevant theoretical work (Malhotra and Serafini 2022)
- original domain recursion (Van den Broeck 2011)
- algorithms
  - FORCLIFT (Van den Broeck et al. 2011)
  - CRANE (Dilkas and Belle 2023a)
  - FASTWFOMC (van Bremen and Kuželka 2021)
  - L2C (Kazemi and Poole 2016) (similarly to us compiles to C++ code, but (probably) doesn't work on as many formulas)
  - approximate (van Bremen and Kuželka 2020)
  - for Markov logic networks (Richardson and Domingos 2006)
    - MAGICIAN (Venugopal, Sarkhel, and Gogate 2015)
    - TUFFY (Niu et al. 2011)
    - ALCHEMY (Gogate and Domingos 2016) (same as the alternative definition)
- complexity
  - liftability (Jaeger and Van den Broeck 2012)
  - hardness for three variables (Beame et al. 2015)
  - liftable fragments
    - C^2 (Kuželka 2021)
    - tree axioms (van Bremen and Kuželka 2023)
    - linear order axioms (Tóth and Kuželka 2023)
    - some liftable fragments (Kazemi et al. 2016)
- applications
  - extensions to sampling (Wang et al. 2022; Wang et al. 2023)
  - discovery of combinatorial sequences (Svatos et al. 2023)
  - conjecturing recurrence relations (Barvínek et al. 2021)
  - probabilistic logic programming (Riguzzi et al. 2017) (WFOMC was shown to be supreme)
  - probabilistic databases (Gribkoff, Suciu, and Van den Broeck 2014)
- lifted inference elsewhere
  - constraint satisfaction (Totis et al. 2023)
  - answer set programming (Azzolini and Riguzzi 2023)

By pointing to the outline in Figure 1, make forward references to the sections of the paper.

Contributions

- Completing the definitions of recursive functions by:
  - identifying a sufficient set of base cases (Section 3.1)
  - constructing formulas that correspond to these base cases (Section 3.2)
  - and recursing on these subproblems
- Compiling these function definitions into a C++ program that can be executed independently for any domain size values Section 5
  - including support for infinite precision arithmetic via GNU Multiple Precision Arithmetic Library
- Experiments comparing CRANE2 with the main alternative approach demonstrate the ability of CRANE2 to scale to domain sizes…

[*]The first and second authors contributed equally and were affiliated with the National University of Singapore during the completion of this work.

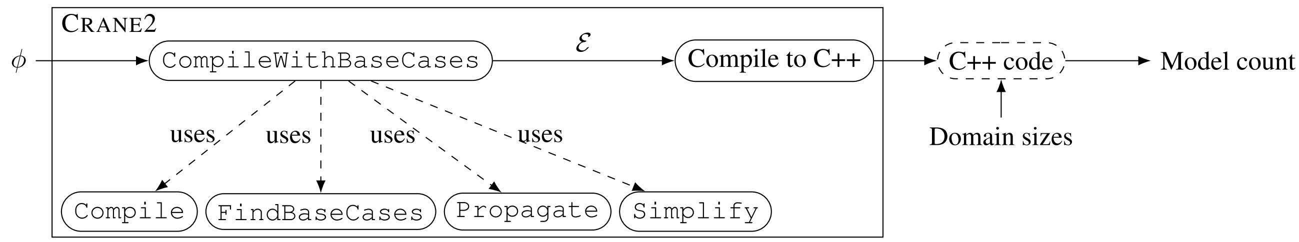

Figure 1: The outline of using CRANE2 to compute the model count of a formula ϕ. The formula is compiled into a set of equations $\mathcal{E}$ that are then compiled to a C++ program. This program can then be run with different command line arguments to compute the model count of ϕ for various domain sizes. `CompileWithBaseCases` makes use of: (i) the knowledge compilation algorithm of CRANE (denoted by `Compile`), (ii) a procedure for identifying a sufficient set of base cases (denoted by `FindBaseCases`), (iii) a procedure for constructing a formula that corresponds to a given base case (denoted by `Propagate`), and (iv) algebraic simplification techniques (denoted by `Simplify`).

Sections 3.1 and 5 deal with algebraic constructs whereas Section 3.2 deals with logic.

2 Preliminaries

Section 2.1 introduces the fundamental ideas of first-order logic and WFOMC, and discusses the different logics used in the latter. Section 2.2 introduces the terminology we use to describe the output of the original CRANE algorithm (Dilkas and Belle 2023a), i.e., functions and equations that define them.

We write $\mathbb{N}_0$ for the set of non-negative integers. In the context of both algebra and logic, we write $C[x \mapsto y]$ for C with all occurrences of x replaced with y.

2.1 Logic

Clarify the differences between the input format and the internal format

The FO Logic (After Skolemization and in a special (prenex, CNF) form)

- Although existential quantifiers are supported, here we describe the format used internally. During preprocessing, all existential quantifiers are eliminated using Skolemization (Van den Broeck, Meert, and Darwiche 2014), and the formula is rewritten into a conjunction of clauses, each of which is in *prenex normal form* (Hinman 2018).
- In the spirit of keeping sorts implicit, we always assume formulas 'type check' with respect to sorts. For example, if $P(x)$, $P(y)$, and $x \neq c$ are all part of the formula (for some predicate P, variables x and y, and constant c), then x, y, and c all have the same sort.
- 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 ϕ 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:
  - there is only one literal with a predicate, and
  - 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:
  - $P(t_1, \ldots, t_m)$ for some predicate P/m and terms $t_1, \ldots, t_m$ or
  - $x = y$ 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.
- 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 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.

Definition 1 (Model)**.** Let ϕ be a formula in FO. For every predicate p/n in ϕ, let $(\Delta_i^p)_{i=1}^n$ be a list of the corresponding domains (not necessarily distinct). Let σ be a map from the domains of ϕ to their interpretations as sets such that:

- the sets are pairwise disjoint, and
- the constants in ϕ are included in the corresponding domains.

Then a *structure* of ϕ (with respect to σ) 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 ϕ and n-tuple $\mathbf{t} \in \prod_{i=1}^n \sigma(\Delta_i^p)$. A structure is a *model* if it satisfies ϕ (see Appendix A of Dilkas and Belle (2023b) for more details).

Definition 2 (WFOMC)**.** Continuing from Definition 1, for every predicate p/n in ϕ, let $w^+(p), w^-(p) \in \mathbb{R}$ be its (positive and negative) *weights*. Unless explicitly specified otherwise, we assume weights to be equal to one. The *(symmetric) weighted first-order model count* (WFOMC) of ϕ (with respect to σ, w^+, and w^-) is the quantity

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

where the sum is over all models of ϕ.

Logic	Sorts	Constants	Variables	Quantifiers	Additional atoms
FO	one or more	✓	unlimited	$\forall$, $\exists$	$x = y$
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: A comparison of the three logics used in WFOMC in terms of: (i) the number of sorts, (ii) support for constants, (iii) the maximum number of variables, (iv) allowed quantifiers, and (v) supported atoms in addition to those of the form $P(\mathbf{t})$ for some predicate P/n and n-tuple of terms $\mathbf{t}$. Here: (i) $k, m \in \mathbb{N}_0$, the latter of which can depend on the domain size, (ii) P is a predicate, and (iii) x and y are terms.

Three Types of Logics

- See Table 1 for a detailed comparison. The notation introduced in the table is standard for C^2, new for $\mathsf{UFO}^2 + \mathsf{EQ}$, and redefined to be more specific for FO.
- All three logics are function-free.
- Domains are always assumed to be finite.
- In many-sorted logic, 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. In the input formula, all domains are assumed to be pairwise disjoint Most of these assignments are typically left implicit and can be reconstructed from the quantifiers. For instance, $\forall x, y \in \Delta.\ P(x, y)$ 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 have $x = y$ as part of a formula.
- – FO is used as the input format for FORCLIFT[1] (Van den Broeck et al. 2011) and its extensions CRANE[2] (Dilkas and Belle 2023a) and CRANE2.
  - C^2 is discussed in the literature on FASTWFOMC (van Bremen and Kuželka 2021) and related methods (Kuželka 2021; Malhotra and Serafini 2022)
  - $\mathsf{UFO}^2 + \mathsf{EQ}$ is the input format supported by a version of FASTWFOMC obtained directly from the authors. Note that the publicly available version[3] does not support any cardinality constraints.
- Note that, in the case of FORCLIFT and its extensions, support for a formula as valid input does not imply that the algorithm will be able to 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 FO formula without constants and with at most two variables (Van den Broeck 2011).

Example 1. Functions

- In C^2: $\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$
- In $\mathsf{UFO}^2 + \mathsf{EQ}$:

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

- In FO:

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

[1] https://github.com/UCLA-StarAI/Forclift
[2] https://doi.org/10.5281/zenodo.8004077
[3] https://comp.nus.edu.sg/~tvanbr/software/fastwfomc.tar.gz

2.2 Algebra

- say something about what `expr` can contain (function calls, various arithmetic operations, summations, non-negative integer constants, variables introduces by the function signature or summations)
- The definition of a function call contradicts the base case finding algorithm. Maybe rephrase it to only look at function calls from f to itself and assume:
  - no mutual recursion,
  - these calls in particular have this form

- We write `expr` for an arbitrary algebraic expression.
- Some terms have different meanings in the context of algebra as compared to logic. Here, a *constant* is a non-negative integer. Similarly, a *variable* is either a parameter of a function or a variable introduced by a summation, e.g., i in $\sum_{i=1}^{n}$ `expr`.

Definition 3. A (function) *signature* is $f(x_1, \ldots, x_n)$ (written $f(\mathbf{x})$ for short), where f is an n-ary function, and each x_i is a variable.

Definition 4. Given a function f with signature $f(\mathbf{x})$, a *function call* of f is $f(\mathbf{y})$, where each y_i is either $x_i - c_i$ or c_i for some constant $c_i \in \mathbb{N}_0$.

Definition 5. An *equation* is $f(\mathbf{x}) =$ `expr`, where $f(\mathbf{x})$ is a signature. We refer to $f(\mathbf{x})$ and `expr` as the *left-hand side* (LHS) and the *right-hand side* (RHS) of the equation, respectively.

Definition 6. Let f be a function with signature $f(\mathbf{x})$. Then a *base case* of f is a function call $f(\mathbf{y})$ such that either $y_i = x_i$ or $y_i \in \mathbb{N}_0$ for all i, and the latter case applies at least once.

3 Completing the Definitions of Recursive Functions

Algorithm 1 outlines our overall approach for compiling a formula into a set of equation that include the required base cases. In short, we first use the knowledge compilation algorithm of the original CRANE (Dilkas and Belle 2023a) to compile the formula 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. After some algebraic simplification, $\mathcal{E}$ is passed to the `FindBaseCases` procedure that returns a set of base cases that we need to find solutions for (described in Section 3.1). For each base case $f(\mathbf{x})$, we identify the formula associated with f and

Algorithm 1: `CompileWithBaseCases(`ϕ`)`

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

1 $(\mathcal{E}, \mathcal{F}, \mathcal{D}) \leftarrow$ `Compile(`ϕ`)`;
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(`ψ, $\mathcal{D}(f, i)$, x_i`)`;
7     $\mathcal{E} \leftarrow \mathcal{E} \cup$ `CompileWithBaseCases(`ψ`)`;

simplify it using the `Propagate` procedure (described in Section 3.2). The algorithm then recurses on these simplified formulas and adds the resulting base case equations to $\mathcal{E}$. Example 2 explains Algorithm 1 in more detail.

Example 2. Let us consider the following formula (previously examined by Dilkas and Belle (2023a)) that defines predicate P to be a bijection between two sets Γ and Δ:

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

In particular, we examine the first solution that CRANE2-BFS returns 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 get the base case $g(0, m) = 0^m$. In the case of $g(l, 0)$, `Propagate(`ψ, Γ, 0`)` returns an empty formula, giving us $g(l, 0) = 1$.

Note that these base cases overlap when $l = m = 0$ but are consistent with each other since $0^0 = 1$. More generally, let ϕ be a formula with two domains Γ and Δ, and let $n, m \in \mathbb{N}_0$. Then the model count of `Propagate(`ϕ, Δ, n`)` assuming $|\Gamma| = m$ is the same as the model count of `Propagate(`ϕ, Γ, m`)` assuming $|\Delta| = n$.

This last claim sounds like something that should be proven.

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

3.1 Identifying a Sufficient Set of Base Cases

- introduce the Iverson bracket (it's only used in this subsection)
- Algorithm 2 is now fixed and supersedes the description below (and the code!)
- NOTE: exactly one of the if and else-if branches always applies

The algorithm is described as Algorithm 2. We know that if, say, on the RHS of all equations, the domain size appears as $m - c_1, m - c_2, \ldots, m - c_k$, then finding $f(0, x_1, x_2, \ldots)$, $f(1, x_1, x_2, \ldots)$, $\ldots f(m_0, x_1, x_2, \ldots)$ for every function f, where $m_0 = \max(c_1, c_2, \ldots c_k) - 1$ forms a sufficient set of base cases. Hence, in order to do the same efficiently, we can take that domain for which m_0 is the minimum, i.e. $\operatorname{argmin}(\max(c_1, c_2, \ldots c_k))$. Ideally, we should calculate the base cases by finding the base cases up to $\max(c_1, c_2, \ldots) - 1$. However, currently only empty and singleton domains are supported.

First, expand the summations in each equation. Here we expand the summations of the form: $\sum_{x=0}^{x_1} \texttt{expr} \cdot [a \le x < b]$ or similar inequalities where x is bounded by constants and a and b are constants, by substituting the value of x from a to $b - 1$. For example, we replace $\sum_{x=0}^{x_1} \binom{x_1}{x} f(x_1 - x) \cdot [0 \le x < 2]$ by $\binom{x_1}{0} f(x_1) + \binom{x_1}{1} f(x_1 - 1)$.

- line 10 for $f(\mathbf{x}) = f(y, z)$, $x_i = y$, and $n = 0$, add $f(y, z)[y \mapsto 0] = f(0, z)$ to $\mathcal{B}$.

Theorem 1. *Under the following assumptions, Algorithm 2 is guaranteed to return a sufficient set of base cases:*

- *there is no mutual recursion*
- *each function f has exactly one equation with f on the LHS;*
- *in the recursive definition of function $f(x_1, \ldots, x_n)$, the i-th argument of each call to f on the RHS is of the form $x_i - c_i$, where:*
  - *$c_i \ge 0$ for all i, and*
  - *$c_i > 0$ for at least one i.*

Algorithm 3: `Propagate`(ϕ, Δ, n)

Input: formula ϕ, domain Δ, $n \in \mathbb{N}_0$
Output: formula ϕ'

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 Δ;
12   **foreach** *clause* $C \in \phi$ **do**
13     $(x_i)_{i=1}^m \leftarrow$ the variables in C with domain Δ;
14   **if** $m = 0$ **then** $\phi' \leftarrow \phi' \cup \{ C \}$;
15   **else**
16       $\phi' \leftarrow \phi' \cup \{ C[x_1 \mapsto c_1, \ldots, x_m \mapsto c_m] \mid (c_i)_{i=1}^m \in D^m \}$;

Example 3. For an equation $f(m, n) = 2 \times f(m-1, n)$, Algorithm 2 returns $f(0, n)$.

3.2 Propagating Domain Size Assumptions

`Propagate` (Algorithm 3) modifies formula ϕ with the assumption that domain Δ has size $n \in \mathbb{N}_0$. In the case of $n = 0$, many clauses become vacuously satisfied and can be removed. In the case of $n > 0$[4], we perform partial grounding, using constants to replace all variables quantified over Δ. Algorithm 3 considers these two cases separately. For a literal or a clause C, we write $\mathrm{Doms}(C)$ to denote the set of corresponding domains.

In the case of $n = 0$, consider three types of clauses: (i) those that do not mention Δ, (ii) those in which every literal contains variables quantified over Δ, and (iii) those that have some literals with variables quantified over Δ and some without. Type (i) clauses are transferred to the new formula ϕ' unchanged. For Type (ii) clauses, $C' = \emptyset$, so these clauses are filtered out. One might think that the same should be done with Type (iii) clauses, however, lines 8 and 9 perform a new kind of smoothing, the explanation of which we defer to Section 4.1.

In the case of $n > 0$, we introduce n new constants. Consider an arbitrary clause $C \in \phi$ and let $m \in \mathbb{N}_0$ be the number of variables in C quantified over Δ. If $m = 0$, then, similarly to the previous case, we add C directly to ϕ'. Otherwise, we add a clause to ϕ' with every possible way of replacing the m variables in C with some combination of the n new constants.

Example 4. Let us consider the clause $C \equiv \forall x \in \Gamma.\ \forall y, z \in \Delta.\ \neg P(x, y) \lor \neg P(x, z) \lor y = z$. 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`($\{ C \}$, Δ, 3) would produce the following formula with nine clauses:

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

where c_1, c_2, and c_3 are the new constants.

[4] None of the formulas considered in this work had $n > 1$.

4 Smoothing

Goal of smoothing: whenever rules such as unit propagation or inclusion-exclusion (maybe just these two?) eliminate the consideration of some ground atoms, smoothing nodes should be inserted 'at the same level' so that these ground atoms are still considered when counting. Note that conjunction nodes are irrelevant here because multiplication is associative. 'At the same level' means that, e.g., if some ground atoms were eliminated from consideration before/after performing independent partial grounding, then the smoothing node should also appear before/after the independent partial grounding node.

- Cite something when introducing the idea of smoothing. Maybe both propositional and first-order sources.
- Can I prove that the additions to smoothing outlined in the rest of this section 'do the right thing'?

4.1 Smoothing for Base Cases

Must include an example of propagating the empty-domain assumption (ideally based on previous examples).

Key point. If there is a predicate P that has nothing to do with domain Δ, make sure that P remains as part of the formula even if all clauses with P can be removed. Which literal is picked on line 8 of Algorithm 3 is not important because any choice achieves the same goal: constructing a clause that mentions the same predicates as C' and is satisfied by any structure.

Fact 1. *Assuming that domain Δ is empty, any clause that contains '$\forall x \in \Delta$' (for any variable x) is vacuously satisfied by all structures.*

For example, consider the formula

$$\forall x \in \Delta.\ \forall y, z \in \Gamma.\ P(x) \lor Q(y, z) \tag{1}$$
$$\forall y, z \in \Gamma'.\ Q(y, z) \tag{2}$$

and assume that $\Gamma' \subseteq \Gamma$. If we set $|\Delta|$ to zero and remove clauses with variables quantified over Δ, we get

$$\forall y, z \in \Gamma'.\ Q(y, z), \tag{3}$$

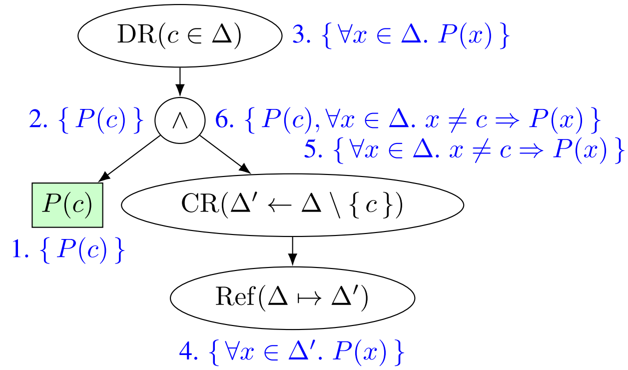

Figure 2: An example based on permutation counting (but simplified a lot). The arc from the Ref node to the DR node is omitted.

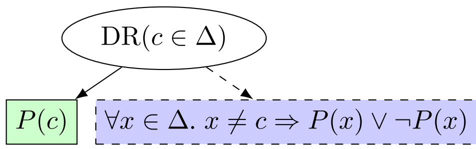

Figure 3: An artificial example of a situation when smoothing nodes need to be added below a DR node

but the model count of Clause (3) is one. However, the actual model count should be $2^{|\Gamma|^2-|\Gamma'|^2}$. That is, Q as a relation is a subset of $\Gamma \times \Gamma$. While Clause (1) becomes vacuously true, Clause (2) fixes the value of Q over $\Gamma' \times \Gamma' \subseteq \Gamma \times \Gamma$. Hence, the number of different values that Q can take is $|(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')| = |\Gamma|^2 - |\Gamma'|^2$.

We address this issue by converting clauses with universal quantifiers over the empty domain to tautologies, hence retaining all the predicates that have no argument assigned to the empty domain. For example, we would convert Clauses (1) and (2) to

$$\forall y, z \in \Gamma.\ Q(y, z) \vee \neg Q(y, z)$$
$$\forall y, z \in \Gamma'.\ Q(y, z).$$

The model count returned by this will also consider the truth value of Q over $y \in \Gamma \setminus \Gamma'$ or $z \in \Gamma \setminus \Gamma'$.

4.2 Smoothing the FCG

- It seems like I'll need to introduce:
  - FCGs
  - node types (be consistent w.r.t. node vs vertex)
  - their notation
- Describe Figures 2 and 3

Smoothing is a two-step process. First, atoms that are still accounted for in the circuit are propagated upwards. Then, at vertices of certain types, missing atoms are detected and additional sinks are created to account for them. If left unchanged, the first step of this process would result in an infinite loop whenever a cycle is encountered. Algorithm 4 outlines how the first step can be adapted to an arbitrary directed graph.

Algorithm 4: Propagate atoms for smoothing across the FCG

Input: FCG (V, s, N^+, τ)
Input: function ι that maps vertex types in $\mathcal{T}$ to sets of atoms
Input: functions $\{\, f_t \,\}_{t \in \mathcal{T}}$ that map a list of sets of atoms to a set of atoms
Output: function S that maps vertices in V to sets of atoms

```
S ← { v ↦ ι(τ(v)) | v ∈ V };
changed ← true;
while changed do
    changed ← false;
    foreach vertex v ∈ V do
        S′ ← f_τ(v)(⟨S(w) | w ∈ N⁺(v)⟩);
        if S′ ≠ S(v) then
            changed ← true;
            S(v) ← S′;
```

4.3 Stage 1: Propagating Unit Clauses 'Upwards'

`Ref`. During smoothing, when unit clauses are propagated in the opposite direction of the FCG arcs (i.e., 'upwards'), when visiting a `Ref` node, these clauses are translated using the domain map of the `Ref` node. For example, if the domain map include $\Delta \mapsto \Delta'$, and the unit clause mentions Δ, then replace it by Δ'.

Constraint removal. Do reverse constraint removal. Assuming that domain Δ with constraints $x \neq c$ (for some constant $c \in \Delta$) were replaced with domain Δ', replace each $\forall x \in \Delta'.\ \phi(x)$ with $\forall x \in \Delta.\ X \neq c \Rightarrow \phi(x)$.

Domain recursion. Suppose the domain recursion node introduces constant $c \in \Delta$. For each unit clause received from the child node, replace each occurrence of $\phi(x)$ or $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$ with $\forall x \in \Delta.\ \phi(x)$. This can be seen as a claim about what ground atoms the domain recursion node *should* cover (or a temporary assumption). If the relevant subgraph indeed covers those ground atoms, Stage 2 will do nothing. Otherwise, smoothing nodes will be added below the domain recursion node to cover the difference between what was propagated from the domain recursion node and what was received from the child node.

4.4 Stage 2: Adding Smoothing Nodes

We never need to add smoothing nodes after `Ref` or constraint removal nodes. However, for domain recursion we must do the following.

1. Whenever the set of unit clauses of the child node contains two formulas $\phi(c)$ and $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$ (i.e., the only difference between the two formulas is that one has the constant c whereas the other one has a variable $x \neq c$), merge them into $\forall x \in \Delta.\ \phi(x)$.

2. Add smoothing nodes below the domain recursion node 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. For example, if the child node 'covers' only $P(c)$, then Stage 1 assigns $\forall x \in \Delta.\ P(x)$ to the domain recursion node. The smoothing node below the domain recursion node then has the clause $\forall x \in \Delta.\ x \neq c \Rightarrow P(x)$.

Can I formally define what is meant by 'difference'?

5 Generating C++ Code

The target is to generate C++ code that can evaluate numerical values of the model counts based on the equations generated by `CompileWithBaseCases`. The translation of a set $\mathcal{E}$ of equations into a C++ program works as follows.

First, we create a cache for each function in $\mathcal{E}$. This is implemented as a multi-dimensional vector containing objects of `class cache_elem` defined as shown in the example code. The default initialization of this object is to -1 which is useful for recognizing unevaluated cases.

Next, we create a function definition for the LHS of each equation in $\mathcal{E}$, including all functions and base cases. The signatures of these functions is decided as follows. A function call containing only variable arguments is named as the function itself, and ones with constants in their arguments are suffixed with a string that contains `'x'` at the ith place if the ith argument is variable and the ith argument if that argument is a constant. For example, $f(x_1, x_2, x_3)$ is declared as `int f(int x1, int x2, int x3);` and $f(1, x_2, x_3)$ is declared as `int f_1xx(int x2, int x3);` (the constant arguments are removed from the signature).

The RHS of each equation in $\mathcal{E}$ is used to define the body of the equation corresponding to the LHS of that equation. The function body (for a function `func` corresponding to equation e) is formed as follows.

First, we check if the evaluation is already present in the cache. If so, then we return the cache element. The cache accesses are done using the `get_elem` function (definition given in the example), which resizes the cache if the accessed index is out of range.

Second, if the element is absent, then we decide if the arguments corresponding to e or one of the functions corresponding to the base cases, based on the value of the arguments. If it corresponds to the base cases, then we directly call the base case function and return its value. Else, we evaluate the value using the RHS, store the evaluated value in the cache and return the evaluated value. Note that in this step, we only call the base case function with one more constant argument than `func`. For example, `f0(x, y)` would call `f0_0x(y)` if $x = 0$ and `f0_x0(x)` if $y = 0$.

Third, to translate the RHS, we convert $\sum_{x=a}^{b}$ `expr` to

```
([y,z,...](){
    int sum = 0;
    for(int x = a; x <= b; x++)
        sum += expr;
    return sum;
})()
```

where $y, z, \ldots$ are the free variables present in `expr`.

6 Experimental Evaluation

Comparing CRANE2 and FASTWFOMC on a larger set of benchmarks is challenging because there is no automated way to translate a formula in FO or C^2 into $\mathsf{UFO}^2 + \mathsf{EQ}$ (or even check if such an encoding is possible).

Benchmarks (probably for supplementary material).

- Functions (Example 1)
- Permutations
  - In C^2:
$$\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$$
$$\forall y \in \Delta.\ \exists^{=1} x \in \Delta.\ P(x, y)$$
  - In $\mathsf{UFO}^2 + \mathsf{EQ}$:
$$\forall x, y \in \Delta.\ R(x) \vee \neg P(x, y)$$
$$\forall x, y \in \Delta.\ S(x) \vee \neg P(y, x)$$
$$|P| = |\Delta|$$
    with weights $w^-(R) = w^-(S) = -1$
  - In FO:
$$\forall x \in \Delta.\ \exists y \in \Delta.\ P(x, y)$$
$$\forall y \in \Delta.\ \exists x \in \Delta.\ P(x, y)$$
$$\forall x, y, z \in \Delta.\ P(x, y) \wedge P(x, z) \Rightarrow y = z$$
$$\forall x, y, z \in \Delta.\ P(x, y) \wedge P(z, y) \Rightarrow x = z$$
  - NOTE: when it's the same domain, counting bijections, injections, surjections, and partial surjections all boil down to the same integer sequence.

If it is possible to rewrite a formula to have more sorts, we do that. Explain in more detail and with an example. Mention that this has been observed previously by my previous work (i.e., linear vs cubic complexity).

Setup.

- The experiments were run on an AMD Ryzen 7 5800H processor with 16 GiB of memory and Arch Linux 6.8.2-arch2-1 operating system. FASTWFOMC was run using Python 3.8.19 with Python-FLINT 0.5.0.
- The knowledge compilation part of both CRANE and CRANE2 can be executed using either greedy (similar to FORCLIFT) or breadth-first search. We use both in our experiments, denoting them as CRANE2-GREEDY and CRANE2-BFS, respectively.

CRANE2-BFS must be introduced earlier (in the introduction?)

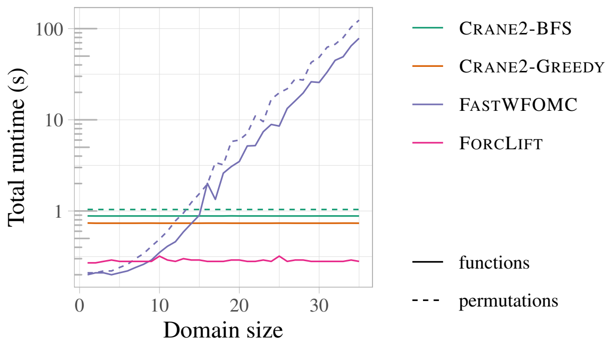

Figure 4: The runtime data of WFOMC algorithms on permutation- and function-counting problems on domains of sizes $1, 2, \ldots, 35$. Note that the y axis is on a logarithmic scale.

Results.

- As shown in Figure 4, the runtimes of all compilation-based algorithms remain practically constant in contrast to the rapidly increasing runtimes of FASTWFOMC.
- Note that CRANE2-BFS is able to handle more instances than FORCLIFT (e.g., the permutation-counting problem in our experiments and other problems in my previous work).
- Although the search/compilation part is slower in CRANE2 than in FORCLIFT, the difference is negligible.
- The runtimes of three out of four WFOMC algorithms appear constant because—for these counting problems and domain sizes—compilation time dominates inference time (recall that compilation time is independent of domain sizes). Indeed, the maximum inference time of both CRANE2-BFS and CRANE2-GREEDY across these experiments is only $4\,\mathrm{ms}$.
- The runtimes of CRANE2 have lower variation than those of FORCLIFT because with FORCLIFT we compile the formula anew for each domain size whereas with CRANE2 we compile it once and reuse the resulting C++ program for all domain sizes.
- As another point of comparison,—in at most $41\,\mathrm{s}$—CRANE2 scales up to domains of sizes 10^4 and 3×10^5 in permutation- and function-counting problems, respectively (whereas FASTWFOMC already takes longer with domains of sizes. . .)

maybe examine FORCLIFT's scalability as well

Some reproducibility requirements to keep in mind:

- A motivation is given for why the experiments are conducted on the selected datasets.
- All novel datasets introduced in this paper are included in a data appendix.
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations. (mention the counting quantifier paper and my KR paper)
- All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from.
- This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics.
- This paper states the number of algorithm runs used to compute each reported result.

7 Conclusion

- Maybe add some complete examples of C++ programs in the supplementary material.
- Later on:
  - Eliminate (e.g., Emacs) warnings.
  - must run it by all 3 other coauthors
  - re-check submission instructions and formatting guidelines
  - Full stop at the end of a single-sentence caption?
  - cite the CRANE paper (and other algorithms' papers) where necessary
  - Something to think about that could inspire some kind of theorems: how can I assume that a domain is empty if there are constants associated with it?

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.

Dilkas, P., and Belle, V. 2023a. Synthesising recursive functions for first-order model counting: Challenges, progress, and conjectures. In *KR*, 198–207.

Dilkas, P., and Belle, V. 2023b. Synthesising recursive functions for first-order model counting: Challenges, progress, and conjectures. *CoRR* abs/2306.04189.

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., and Poole, D. 2016. Knowledge compilation for lifted probabilistic inference: Compiling to a low-level language. In *KR*, 561–564. AAAI Press.

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.

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.

Kuželka, O. 2023. Counting and sampling models in first-order logic. In *IJCAI*, 7020–7025. ijcai.org.

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.

Richardson, M., and Domingos, P. M. 2006. Markov logic networks. *Mach. Learn.* 62(1-2):107–136.

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.; 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.

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. 2011. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *NIPS*, 1386–1394.

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.; 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.

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.