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 (TODO)

- 9 pages including appendices but not acknowledgements
- 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)

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

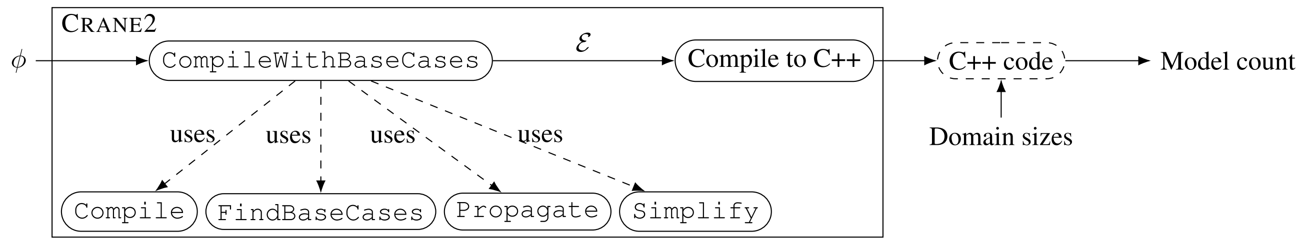

Figure 1: The outline of using CRANE2 to compute the model count of a formula ϕ. The formula is compiled into a set of equations $\mathcal{E}$ that are then compiled to a C++ program. This program can then be run with different command line arguments to compute the model count of ϕ for various domain 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`).

- By pointing to the outline in Figure 1, make forward references to the sections of the paper.
- Somewhere: mention that we're primarily concerned with computing unweighted model counts, but everything trivially extends to the weighted setting as well.
- Introduce greedy search and breadth-first search (BFS), together with CRANE2-GREEDY and CRANE2-BFS. 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.
- Introduce the WFOMC acronym.
- Introduce first-order knowledge compilation as a class of WFOMC algorithms, defining/mentioning compilation rules, FCG, etc.

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…

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. Then, 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 $S\sigma$ for a *substitution* σ applied to an expression S, where $\sigma = [x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n]$ denotes the replacement of all occurrences of x_i with y_i for all $i = 1, \ldots, n$.

2.1 Logic (TODO)

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.

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 $\mathsf{UFO}^2 + \mathsf{EQ}$:

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

- In FO:

$$\begin{gathered} \forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y) \\ \forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \land P(x, z) \Rightarrow y = z \end{gathered} \tag{1}$$

Note that Formula (1) is written with two domains instead of one. Clearly, the model count does not change whenever $|\Gamma| = |\Delta|$. As was previously observed by Dilkas and Belle (2023a), such a reformulation can help first-order knowledge compilation algorithms find efficient solutions.

2.2 Algebra

We write `expr` for an arbitrary algebraic expression. Note that 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 $\sum_{i=1}^n \texttt{expr}$. 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. An *equation* is $f(\mathbf{x}) = \texttt{expr}$, where $f(\mathbf{x})$ is a signature.

Definition 3. 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 a function call $f(\mathbf{y})$ is a *base case* of $f(\mathbf{x})$ if $f(\mathbf{y}) = f(\mathbf{x})\sigma$, where σ is a substitution that replaces one or more x_i with a constant.

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

[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

Logic	Sorts	Constants	Variables	Quantifiers	Additional atoms
FO	one or more	✓	unlimited	$\forall$, $\exists$	$x = y$
C^2	one	✗	two	$\forall$, $\exists$, $\exists^{=k}$, $\exists^{\leq k}$, $\exists^{\geq k}$	—
$\mathsf{UFO}^2 + \mathsf{EQ}$	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.

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 $\quad \psi \leftarrow \mathcal{F}(f)$;
5 $\quad$ **foreach** *i such that* $x_i \in \mathbb{N}_0$ **do**
6 $\quad\quad \psi \leftarrow$ `Propagate(`ψ, $\mathcal{D}(f, i)$, x_i`)`;
7 $\quad \mathcal{E} \leftarrow \mathcal{E} \cup$ `CompileWithBaseCases(`ψ`)`;

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 (described in Section 3.1) that returns a set of base cases that we need to find solutions for. For each base case $f(\mathbf{x})$, we identify the formula associated with f and 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. 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:

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

In particular, we examine the first solution that CRANE2-BFS returns for this formula.

After lines 1 and 2, we have

$$\mathcal{E} = \left\{ \begin{array}{c} f(m, n) = \sum_{l=0}^{n} \binom{n}{l} (-1)^{n-l} g(l, m), \\ g(l, m) = g(l-1, m) + m g(l-1, m-1) \end{array} \right\};$$
$$\mathcal{D} = \{ (f, 1) \mapsto \Gamma, (f, 2) \mapsto \Delta, (g, 1) \mapsto \Delta^\top, (g, 2) \mapsto \Gamma \},$$

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 $\forall x \in \Gamma.\ S(x) \vee \neg S(x)$, 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 $|\Delta| = n$.

Can I turn the above into a theorem?

Finally, we note that the `Simplify` procedure plays a crucial role in simplifying a common algebraic pattern $\sum_{m=0}^{n} [a \leq m \leq b] f(m)$. 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) $[a \leq m \leq b] = \begin{cases} 1 & \text{if } a \leq m \leq b \\ 0 & \text{otherwise} \end{cases}$ is 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 $g(l, m) = \sum_{k=0}^{m} [0 \leq k \leq 1] \binom{m}{k} g(l-1, m-k)$ into the simpler form above.

A concluding sentence that introduces the remaining subsections?

3.1 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of `FindBaseCases`. For each recursive call from a function f to itself, we consider two types of arguments: (i) constants and (ii) arguments of the form $x_i - c_i$, where $c_i \in \mathbb{N}_0$ is a constant, and x_i is the i-th argument of the signature of f. In the former case, we consider a base case specifically for that constant. In the latter case, we consider a base case for constants from zero up to (but not including) c_i. Theorem 1 below motivates this approach.

Example 3. Consider the recursive function g from Example 2. `FindBaseCases(`$\mathcal{E}$`)` 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 $g(l, 0)$.

Prove (lots of notes, handwritten and elsewhere)

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

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

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

Theorem 1 (Termination)**.** *Let us assume the following about the input set of equations $\mathcal{E}$:*

1. *Each function f has exactly one equation with f on the left-hand side. We call this equation the* definition *of f.*
2. *There exists a topological ordering of all functions $(f_i)_i$ such that the definition of f_i does not contain function calls to f_j with $j > i$.*[4]
3. *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 $c_i \in \mathbb{N}_0$;*
   - *there exists i such that $y_i = x_i - c_i$ for some $c_i > 0$.*

Then the set of base cases $\mathcal{B}$ returned by `FindBaseCases(`$\mathcal{E}$`)` *is* sufficient *for $\mathcal{E}$ in the following sense. The evaluation of any function call with non-negative arguments terminates as long as the evaluation of base cases is prioritised over the corresponding recursive definitions.*[5]

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 $n > 0$[6], 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.

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

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 Smoothing

Smoothness originates in propositional knowledge compilation where it is defined as the property that, for every disjunction node, all disjuncts (as subtrees of the circuit) contain the same atoms (Darwiche 2001). Van den Broeck et al. (2011) generalise smoothness to first-order logic, adding set-disjunction and inclusion-exclusion nodes alongside disjunction.

[4] This condition excludes the possibility of mutual recursion and similar cyclic scenarios and is akin to stratified logic programs (Lloyd 1987).

[5] Recall that, as previously discussed, the order in which base cases are considered is immaterial.

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

The motivation for smoothing is as follows. Whenever compilation rules such as unit propagation and inclusion-exclusion simplify the formula, some ground atoms might be eliminated from consideration (see Example 5 below). To account for them during counting, smoothing nodes (i.e., tautological clauses such as $P(c) \vee \neg P(c)$) are added to the FCG at an appropriate location.

In the rest of this section, we extend the smoothing algorithm of Van den Broeck et al. (2011) even further. Section 4.1 describes the role smoothing plays in the base-case-finding algorithm from Section 3. Then, Section 4.2 shows how to adapt smoothing to the compilation rules introduced by Dilkas and Belle (2023a).

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 $(\phi, \Delta, 0)$, i.e., we are simplifying formula ϕ by assuming that domain Δ is empty. Informally, if there is a predicate P in ϕ that has nothing to do with domain Δ, the role of smoothing is to preserve all occurrences of P even if all clauses with P become vacuously satisfied. We note that the approach presented in this section is far from unique and explain it via an example below.

Example 5. Let ϕ be

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

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

where $\Gamma' \subseteq \Gamma$ is a domain introduced by a compilation rule. Note that, as a relation, $P \subseteq \Gamma \times \Gamma$.

Let us reason by hand about the model count of ϕ when $\Delta = \emptyset$. Predicate Q can only take one value, i.e., $Q = \emptyset$. The value of P is fixed over $\Gamma' \times \Gamma'$ by Clause (3) but is allowed to vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since Clause (2) is vacuously satisfied by all structures. Hence, the right model count should be $2^{|\Gamma|^2 - |\Gamma'|^2}$.

However, without line 9, `Propagate` would simplify ϕ to $\forall y, z \in \Gamma'.\ P(y, z)$. Here, P is defined as a subset of $\Gamma' \times \Gamma'$. Clearly, this simplified formula has only one model: $\{\, P(y, z) \mid y, z \in \Gamma' \,\}$.

With line 9 included, ϕ is transformed to

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

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

which retains the correct model count.

Note 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. First, positive unit clauses (denoting sets of ground atoms that are accounted for in the FCG) are propagated 'upwards', i.e., in the opposite direction of FCG arcs. Then, at nodes of certain types, missing atoms are detected and additional nodes are added to account for them. In this section, we: (i) describe the relevant node types from previous work, (ii) show how smoothing ought to work for these node types, and (iii) illustrate how the new smoothing techniques work on two example FCGs.

Before describing the proposed changes to smoothing, we briefly review the compilation rules (and their corresponding node types) introduced by Dilkas and Belle (2023a) while referring to the aforementioned paper for precise definitions. *Domain recursion* selects a domain Δ, introduces a new constant $c \in \Delta$, and, for each variable x quantified over Δ, considers two possibilities: $x = c$ or $x \neq c$, modifying the formula accordingly. We denote the resulting node as $\mathrm{DR}(c \in \Delta)$. *Constraint removal* applies to formulas such that, for some domain Δ, each variable x quantified over Δ is followed by the inequality constraint $x \neq c$. In other words, the relevant clauses can be rewritten to begin with $\forall x \in \Delta.\ x \neq c \Rightarrow \ldots$ For such formulas, constraint removal replaces Δ with a new domain Δ' and removes the $x \neq c$ constraints. We denote the resulting node as $\mathrm{CR}(\Delta' \leftarrow \Delta \setminus \{\, c \,\})$. Finally, *caching* detects when the input formula ϕ is equal to a previously-encountered formula ψ except for having different domains. We denote the resulting node as $\mathrm{Ref}(\sigma)$, where σ is the substitution mapping the domains of ψ to their corresponding domains in ϕ.

Stage 1 for domain recursion. When visiting a $\mathrm{DR}(c \in \Delta)$ node, for each unit clause received from the child node, we replace each occurrence of $\phi(c)$ or $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$ with $\forall x \in \Delta.\ \phi(x)$. In other words, if the subgraph below the domain recursion node covers some of the ground atoms that are affected by domain recursion, then it should cover all of them. 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 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 $\mathrm{CR}(\Delta' \leftarrow \Delta \setminus \{\, c \,\})$ node, we reverse constraint removal. In other words, we replace 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 substitution σ to the unit clauses coming from the child node.

Stage 2 for domain recursion. We need not add smoothing nodes immediately below constraint removal or caching nodes. However, for domain recursion we do the following.

1. Suppose the set of unit clauses assigned to the child node during Stage 1 contains both $\phi(c)$ and $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$. In other words, the only difference between the two clauses is that one has the constant c whereas the other one has a variable $x \neq c$. In such a case, we merge the two clauses into $\forall x \in \Delta.\ \phi(x)$.
2. If necessary, we add smoothing nodes below the domain recursion node to cover 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.

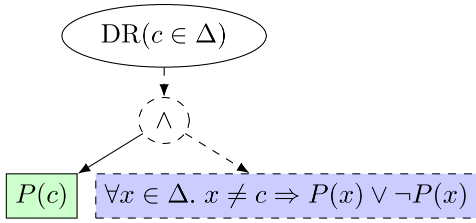

Figure 2: An artificial example of an FCG where a smoothing node must be added below a domain recursion node. The dashed nodes and arcs are added during Stage 2 of smoothing.

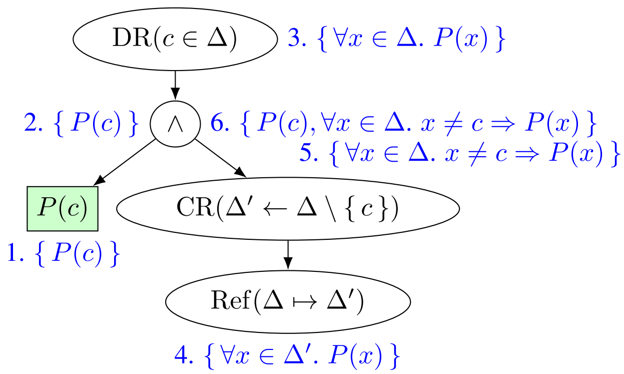

Figure 3: A smooth FCG loosely based on Example 2. The arc from the caching node to the domain recursion node is omitted for compactness. The labels next to nodes show the sets of unit clauses assigned to each node during Stage 1 and the order of these assignments, omitting empty assignments that replace a set S with S itself.

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

Example 7 (A smooth FCG)**.** Stage 1 of smoothing is more involved in the case of the FCG 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 then propagates to the caching node, changing 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 is then propagated to the conjunction node, joining $P(c)$.

In Stage 2, $P(c)$ and $\forall x \in \Delta.\ x \neq c \Rightarrow P(x)$ are merged into $\forall x \in \Delta.\ P(x)$. Since the resulting clause 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 covers $P(c)$, and the smoothing algorithm correctly recognises that the recursive call also covers $P(x)$ for all $x \in \Delta \setminus \{\, c \,\}$.

5 Generating C++ Code

In this section, we describe the last step of CRANE2 outlined in Figure 1: translating the set of equations $\mathcal{E}$ produced by `CompileWithBaseCases` into C++ code. The produced C++ program can then be compiled and executed with different command-line arguments to compute the model count of the formula for different (combinations of) domain sizes.

Each equation in $\mathcal{E}$ is compiled into a C++ function, together with its own cache for memoisation. Let $e = (f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$ be an arbitrary equation, 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 whether $\mathbf{c}$ is already in the cache of e (in which case the cached value is returned). Second, for each base case $f(\mathbf{y})$ of $f(\mathbf{x})$ (as in Definition 3), we check whether $\mathbf{c}$ *matches* $\mathbf{y}$, i.e., $c_i = y_i$ whenever $y_i \in \mathbb{N}_0$. In this case, $\mathbf{c}$ is redirected to the C++ function that corresponds to the definition of base case $f(\mathbf{y})$. Finally, if the above cases fail, we evaluate $\mathbf{c}$ according to `expr`, store the result in the cache, and return it.

6 Experimental Evaluation (TODO)

We compare CRANE2 (in both BFS and greedy modes) with FASTWFOMC and FORCLIFT on two problems previously considered by Dilkas and Belle (2023a): the function-counting problem from Example 1 and the bijection-counting problem described below. Note that 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). 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 executed using Python 3.8.19 with Python-FLINT 0.5.0.

Benchmarks (probably for supplementary material).

- Bijections
  - 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: see Example 2

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 bijection-counting problem in our experiments and other problems in my previous work).

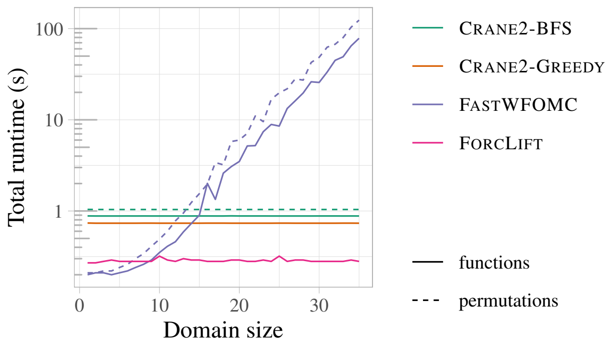

Figure 4: The runtime data of WFOMC algorithms on domains of sizes $1, 2, \dots, 35$. Note that the y axis is on a logarithmic scale.

- 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 bijection- 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 (TODO)

- 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?
  - consider turning small sections (e.g., Section 5) into subsections of even paragraphs.
  - Authors may submit a separate PDF with additional information supporting their claims (such as proof details, additional experimental results, further details on experimental design, etc).

References

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

Barvínek, J.; van Bremen, T.; Wang, Y.; Zelezný, F.; and Kuželka, O. 2021. Automatic conjecturing of P-recursions using lifted inference. In *ILP*, volume 13191 of *Lecture Notes in Computer Science*, 17–25. Springer.

Beame, P.; Van den Broeck, G.; Gribkoff, E.; and Suciu, D. 2015. Symmetric weighted first-order model counting. In *PODS*, 313–328. ACM.

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. 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.

Lloyd, J. W. 1987. *Foundations of Logic Programming, 2nd Edition*. Springer.

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.