Synthesising Recursive Functions for First-Order Model Counting: Challenges, Progress, and Conjectures

Paulius Dilkas[1] , **Vaishak Belle**[2]

[1]National University of Singapore, Singapore, Singapore
[2]University of Edinburgh, Edinburgh, UK
paulius.dilkas@nus.edu.sg, vbelle@ed.ac.uk

Abstract

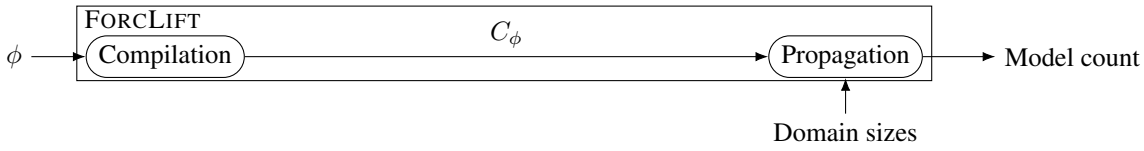
First-order model counting (FOMC) is a computational problem that asks to count the models of a sentence in finite-domain first-order logic. In this paper, we argue that the capabilities of FOMC algorithms to date are limited by their inability to express many types of recursive computations. To enable such computations, we relax the restrictions that typically accompany domain recursion and generalise the circuits used to express a solution to an FOMC problem to directed graphs that may contain cycles. To this end, we adapt the most well-established (weighted) FOMC algorithm FORCLIFT to work with such graphs and introduce new compilation rules that can create cycle-inducing edges that encode recursive function calls. These improvements allow the algorithm to find efficient solutions to counting problems that were previously beyond its reach, including those that cannot be solved efficiently by any other exact FOMC algorithm. We end with a few conjectures on what classes of instances could be domain-liftable as a result.

1 Introduction

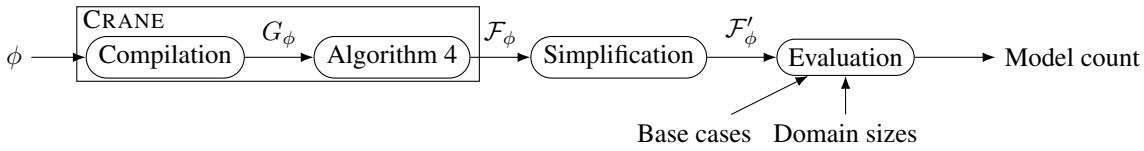
First-order model counting (FOMC) is the problem of computing the number of models of a sentence in first-order logic given the size(s) of its domain(s) (Beame et al. 2015). *Symmetric weighted FOMC* (WFOMC) extends FOMC with (pairs of) weights on predicates and asks for a weighted sum across all models instead. By fixing the sizes of the domains, a WFOMC instance can be rewritten as an instance of (propositional) weighted model counting (Chavira and Darwiche 2008). WFOMC emerged as the dominant approach to *lifted (probabilistic) inference*. Lifted inference techniques exploit symmetries in probabilistic models by reasoning about sets rather than individuals (Kersting 2012). By doing so, many instances become solvable in polynomial time (Van den Broeck 2011). The development of lifted inference algorithms coincided with work on probabilistic relational models that combine the syntactic power of first-order logic with probabilistic models such as Bayesian and Markov networks, allowing for a more relational view of uncertainty modelling (De Raedt et al. 2016; Kimmig, Mihalkova, and Getoor 2015; Richardson and Domingos 2006). Lifted inference techniques for probabilistic databases have also been inspired by WFOMC (Gatterbauer and Suciu 2015; Gribkoff, Suciu, and Van den Broeck 2014). While WFOMC has received more attention in the literature, FOMC is an interesting problem in and of itself because of its connections to finite model theory (van Bremen and Kuželka 2021b) and applications in enumerative combinatorics (Barvínek et al. 2021).

Traditionally in computational complexity theory, a problem is *tractable* if it can be solved in time polynomial in the instance size. The equivalent notion in (W)FOMC is liftability. A (W)FOMC instance is *(domain-)liftable* if it can be solved in time polynomial in the size(s) of the domain(s) (Jaeger and Van den Broeck 2012). Many classes of instances are known to be liftable. First, Van den Broeck (2011) showed that the class of all sentences of first-order logic with up to two variables (denoted FO^2) is liftable. Then Beame et al. (2015) proved that there exists a sentence with three variables for which FOMC is $\#\mathsf{P}_1$-complete (i.e., FO^3 is not liftable). Since these two seminal results, most research on (W)FOMC focused on developing faster solutions for the FO^2 fragment (van Bremen and Kuželka 2021a; Malhotra and Serafini 2022) and defining new liftable fragments. These fragments include $\mathsf{S}^2\mathsf{FO}^2$ and $\mathsf{S}^2\mathsf{RU}$ (Kazemi et al. 2016), U_1 (Kuusisto and Lutz 2018), C^2 (i.e., the two-variable fragment with counting quantifiers) (Kuželka 2021; Malhotra and Serafini 2022), and C^2 extended with axioms for trees (van Bremen and Kuželka 2021b). On the empirical front, there are several implementations of exact WFOMC algorithms: ALCHEMY (Gogate and Domingos 2016), FASTWFOMC (van Bremen and Kuželka 2021a), FORCLIFT (Van den Broeck et al. 2011), and L2C (Kazemi and Poole 2016). Approximate counting is supported by ALCHEMY, APPROXWFOMC (van Bremen and Kuželka 2020), FORCLIFT (Van den Broeck, Choi, and Darwiche 2012), MAGICIAN (Venugopal, Sarkhel, and Gogate 2015), and TUFFY (Niu et al. 2011).

We claim that the capabilities of (W)FOMC algorithms can be significantly extended by empowering them with the ability to construct recursive solutions. The topic of recursion in the context of WFOMC has been studied before but in limited ways. Barvínek et al. (2021) use WFOMC to generate numerical data that is then used to conjecture recurrence relations that explain that data. Van den Broeck (2011) introduced the idea of *domain recursion*. Intuitively, domain recursion partitions a domain of size n into a single explicitly named constant and the remaining domain of size $n-1$. However, many stringent conditions are enforced to ensure



(a) FORCLIFT compiles ϕ into a circuit C_ϕ. The domain sizes are then used to propagate values through C_ϕ, computing the model count.



(b) CRANE compiles ϕ into a graph G_ϕ and then converts G_ϕ into a collection of functions $\mathcal{F}_\phi$. In some cases, these functions can benefit from algebraic simplification. If some functions are defined recursively, base cases need to be established. One can then compute the model count of ϕ by running the main function in $\mathcal{F}'_\phi$ with domain sizes as arguments.

Figure 1: A comparison of how FORCLIFT and CRANE can be used to compute the model count of a formula ϕ

that the search for a tractable solution always terminates.

In this work, we show how to relax these restrictions in a way that results in an algorithm capable of handling more instances in a lifted manner. The ideas presented in this paper are implemented in CRANE—an extension of the (W)FOMC algorithm FORCLIFT. The differences in how these algorithms operate are depicted in Figure 1. Compilation is performed by applying various *(compilation) rules* to the input (or some derivative) formula, gradually constructing a circuit (in the case of FORCLIFT) or a graph (in the case of CRANE). FORCLIFT applies compilation rules via greedy search, whereas CRANE also supports a hybrid search algorithm that applies some rules greedily and some using breadth-first search.[1] This alternative was introduced because there is no reason to expect greedy search to be complete. Another difference is that—in CRANE—the product of compilation is not directly evaluated but transformed into a collection of functions on domain sizes. Hence, our approach is reminiscent of previous work on lifted inference via compilation to C++ programs (Kazemi and Poole 2016) and the broader area of functional synthesis (Golia, Roy, and Meel 2020; Kuncak et al. 2010; Sanathanan and Koerner 1963).

Using labelled directed graphs instead of circuits enables CRANE to construct recursive solutions by representing recursive function calls via cycle-inducing edges. A hypothetical instance of compilation could proceed as follows. Suppose the input formula ϕ depends on a domain of size $n \in \mathbb{N}_0$. *Generalised domain recursion* (GDR)—one of the new compilation rules—transforms ϕ into a different formula ψ with an additional constant and some *constraints*. After some more transformations, the constraints in ψ can be removed, replacing the domain of size n with a new domain of size $n-1$—this is the responsibility of the *constraint removal* (CR) compilation rule. Afterwards, another compilation rule recognises that the resulting formula matches the input formula ϕ except for referring to a different domain. This observation allows us to add a cycle-forming edge to the graph, which can be interpreted as a function f relying on $f(n-1)$ to compute $f(n)$.

We begin by introducing some notation, terminology, and the problem of FOMC in Section 2. Then, in Section 3, we define the graphs that replace circuits in representing a solution to such a problem. Section 4 introduces the new compilation rules. Section 5 describes an algorithm that converts such a graph into a collection of (potentially recursive) functions. Section 6 compares FORCLIFT and CRANE on various counting problems. We show that: (i) CRANE performs as well as FORCLIFT on the instances that were already solvable by FORCLIFT, (ii) CRANE is also able to handle most of the instances that FORCLIFT fails on, *including those outside of currently-known domain-liftable fragments* such as C^2. Finally, Section 7 outlines some conjectures and directions for future work.

2 Preliminaries

Our representation of FOMC instances builds on the format used internally by FORCLIFT, some aspects of which are described by Van den Broeck et al. (2011). FORCLIFT can translate sentences in a variant of function-free many-sorted first-order logic with equality to this internal format. We use lowercase Latin letters for predicates (e.g., $\mathtt{p}$) and constants (e.g., x), uppercase Latin letters for variables (e.g., X), and uppercase Greek letters for domains (e.g., Δ). Sometimes we write predicate $\mathtt{p}$ as $\mathtt{p}/n$, where $n \in \mathbb{N}^+$ is the *arity* of $\mathtt{p}$. An *atom* is $\mathtt{p}(x_1, \ldots, x_n)$ for some predicate $\mathtt{p}/n$ and terms $x_1, \ldots, x_n$. A *term* is either a constant or a variable. A *literal* is either an atom or the negation thereof (denoted by $\neg\mathtt{p}(x_1, \ldots, x_n)$). Let $\mathcal{D}$ be the set of all relevant (finite) domains. Initially, $\mathcal{D}$ contains all domains mentioned by the input formula. During compilation, new domains are added to $\mathcal{D}$. Each new domain is interpreted as a subset of another domain in $\mathcal{D}$.

We write $\langle\cdot\rangle$ for lists, $|\cdot|$ for the length of a list, and $+\!\!+$ for list concatenation. We write $\nrightarrow$ to denote partial functions

[1] In the current implementation, the rules applied non-greedily are: atom counting, inclusion-exclusion, independent partial groundings, Shannon decomposition, shattering, and two new rules described in Sections 4.1 and 4.3—see previous work (Van den Broeck et al. 2011) for more information about the rules.

and $\mathrm{dom}(\cdot)$ for the domain of a function. Let Doms (respectively, Vars) be the function that maps any expression to the set of domains (respectively, variables) used in it. Let S be a set of constraints or literals, V a set of variables, and x a term. We write $S[x/V]$ to denote S with all occurrences of all variables in V replaced with x.

Definition 1 (Constraint)**.** *An* (inequality) constraint *is a pair (a, b), where a is a variable, and b is a term. It constrains a and b to be different.*

Definition 2 (Clause)**.** *A* clause *is $c = (L, E, \delta_c)$, where L is a set of literals, E is a set of constraints, and δ_c is the domain map.* Domain map $\delta_c \colon \mathrm{Vars}(c) \to \mathcal{D}$ *is a function that maps all variables in c to their domains such that (s.t.) if $(X, Y) \in E$ for some variables X and Y, then $\delta_c(X) = \delta_c(Y)$. For convenience, we sometimes write δ_c for the domain map of c without unpacking c into its three constituents.*

Definition 3 (Formula)**.** *A* formula *is a set of clauses s.t. all constraints and atoms 'type check' with respect to domains.*

Example 1. *Let $\phi \coloneqq \{ c_1, c_2 \}$ be a formula with clauses*

$$c_1 \coloneqq (\{ \neg \mathtt{p}(X, Y), \neg \mathtt{p}(X, Z) \}, \{ (Y, Z) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \}),$$
$$c_2 \coloneqq (\{ \neg \mathtt{p}(X, Y), \neg \mathtt{p}(Z, Y) \}, \{ (X, Z) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Gamma \})$$

for some predicate $\mathtt{p}/2$, variables X, Y, Z, and domains Γ and Δ. All variables that occur as the first argument to $\mathtt{p}$ are in Γ, and, likewise, all variables that occur as the second argument to $\mathtt{p}$ are in Δ. Therefore, ϕ is a valid formula.

One can read such a formula as a sentence in first-order logic. All variables in a clause are implicitly universally quantified, and all clauses in a formula are implicitly linked by a conjunction. Thus, formula ϕ from Example 1 reads as

$$\begin{aligned} &(\forall X \in \Gamma.\ \forall Y, Z \in \Delta. \\ &\quad Y \neq Z \Rightarrow \neg \mathtt{p}(X, Y) \vee \neg \mathtt{p}(X, Z)) \wedge \\ &(\forall X, Z \in \Gamma.\ \forall Y \in \Delta. \\ &\quad X \neq Z \Rightarrow \neg \mathtt{p}(X, Y) \vee \neg \mathtt{p}(Z, Y)). \end{aligned} \tag{1}$$

There are two differences between Definitions 1–3 and the corresponding concepts by Van den Broeck et al. (2011).[2] First, we decouple variable-to-domain assignments from constraints and move them to a separate function δ_c in Definition 2. Second, while they allow for equality constraints and constraints of the form $X \notin \Delta$ for some variable X and domain Δ, we exclude these constraints simply because they are inessential. Note that if we replace $Y \neq Z$ in Formula (1) with $Y = Z$, then Formula (1) can be simplified to have one fewer variables. Similarly, if the same inequality is replaced by $Y = x$ for some constant x, then Y can be eliminated as well. Since constraints are always interpreted as preconditions for the disjunction of literals in the clause (as in Formula (1)), equality constraints can be eliminated without any loss in expressivity.

[2] Van den Broeck et al. (2011) refer to clauses and formulas as c-clauses and c-theories, respectively.

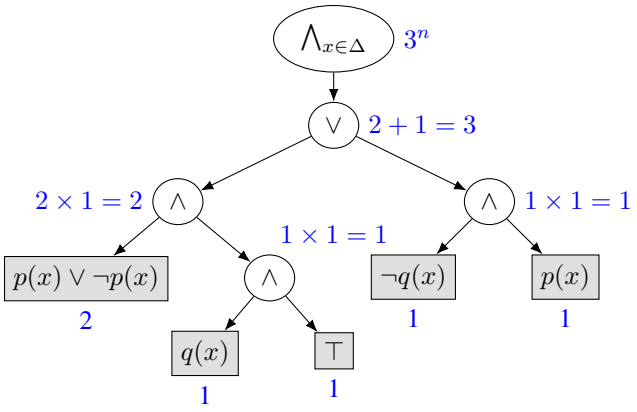


Figure 2: A circuit produced by FORCLIFT for Example 2. Values and computations in blue (on the outside of each node) show how a bottom-up evaluation of the circuit computes the model count.

Example 2. *Let Δ be a domain of size $n \in \mathbb{N}_0$. The model count of $\forall X \in \Delta.\ \mathtt{p}(X) \vee \mathtt{q}(X)$ is then 3^n. Intuitively, since both predicates are of arity one, they can be interpreted as subsets of Δ. Thus, the formula says that each element of Δ has to be in $\mathtt{p}$ or $\mathtt{q}$ or both.*

Example 3. *Consider a variant of the well-known 'friends and smokers' example $\forall X, Y \in \Delta.\ \mathtt{smokes}(X) \wedge \mathtt{friends}(X, Y) \Rightarrow \mathtt{smokes}(Y)$. Letting $n \coloneqq |\Delta|$ as before, the model count is $\sum_{k=0}^{n} \binom{n}{k} 2^{n^2 - k(n-k)}$ (Van den Broeck, Meert, and Darwiche 2014).*

The model count of a formula ϕ also depends on the sizes of the domains in ϕ. Let $\sigma \colon \mathcal{D} \to \mathbb{N}_0$ be the *domain size function* that maps each domain to a non-negative integer.

Example 4. *Let ϕ be as in Example 1 and $\Gamma = \Delta = \{ 1, 2 \}$, i.e., $\sigma(\Gamma) = \sigma(\Delta) = 2$. There are $2^{2 \times 2} = 16$ possible relations between Γ and Δ. Let us count how many of them satisfy the conditions imposed on predicate $\mathtt{p}$. The empty relation does. All four relations of cardinality one (e.g., $\{ (1, 1) \}$) do too. Finally, there are two relations of cardinality two—$\{ (1, 1), (2, 2) \}$ and $\{ (1, 2), (2, 1) \}$—that satisfy the conditions as well. Therefore, the FOMC of (ϕ, σ) is 7. Incidentally, the FOMC of ϕ counts partial injections from Γ to Δ. We will continue to use the problem of counting partial injections as the main running example.*

3 First-Order Computational Graphs

Darwiche (2001) introduced *deterministic decomposable negation normal form* (d-DNNF) circuits for propositional knowledge compilation and showed that the model count of a propositional formula can be computed in time linear in the size of the circuit. Van den Broeck et al. (2011) generalised them to first-order logic via *first-order d-DNNF (FO d-DNNF) circuits*. FO d-DNNF circuits (hereafter called *circuits*) are directed acyclic graphs with nodes corresponding to formulas in first-order logic—see Figure 2 for an example. The following types of nodes are supported by FORCLIFT: caching (REF), contradiction ($\bot$), tautology ($\top$), decomposable conjunction ($\wedge$), decompos-

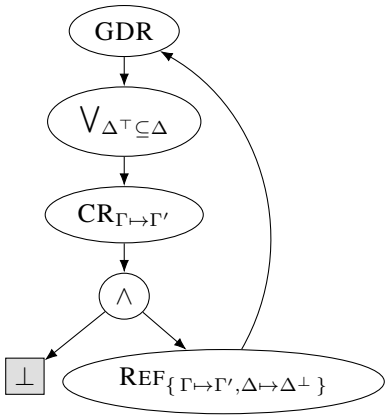


Figure 3: A simplified version of the FCG constructed by CRANE for the problem of counting partial injections from Example 1. Here we omit some parameters as well as nodes whose only arithmetic effect is multiplication by one.

able set-conjunction ($\bigwedge$), deterministic disjunction ($\vee$), deterministic set-disjunction ($\bigvee$), domain recursion, grounding, inclusion-exclusion, smoothing, and unit clause. We refer the reader to previous work (Van den Broeck 2011; Van den Broeck et al. 2011) for more information about node types and their interpretations for computing the (W)FOMC.

We introduce *first-order computational graphs* (FCGs) that generalise circuits by dispensing with acyclicity. An FCG is a (weakly connected) directed graph with a single source, node labels, and ordered outgoing edges. Node labels consist of two parts: the *type* and the *parameters*. The type of a node determines its out-degree. We make the following changes to the node types already supported by FORCLIFT. First, we introduce a new type for constraint removal (CR). Second, we replace domain recursion with generalised domain recursion (GDR). And third, for reasoning about partially-constructed FCGs, we write $\star$ for a placeholder type that is yet to be replaced. We write T_p for an FCG that has a node with label T_p (i.e., type T and parameter(s) p) and $\star$'s as all of its direct successors. We also write $T_p(v)$ for an FCG with one edge from a node labelled T_p to another node v. See Figure 3 for an example FCG.

Finally, we introduce a structure that represents a solution while it is still being built. A *chip* is a pair (G, L), where G is an FCG, and L is a list of formulas, s.t. $|L|$ is equal to the number of $\star$'s in G. L contains formulas that still need to be compiled. Once a formula is compiled, it replaces one of the $\star$'s in G according to a set order. We call an FCG *complete* (i.e., it represents a *complete solution*) if it has no $\star$'s. Similarly, a chip is complete if its FCG is complete.

4 New Compilation Rules

A *(compilation) rule* takes a formula and returns a set of chips. The cardinality of this set is the number of ways the rule can be applied to the input formula. While FORCLIFT (Van den Broeck et al. 2011) heuristically chooses one of them, in an attempt to not miss a solution, CRANE returns them all. In particular, if a rule returns an empty set, then that rule does not apply to the formula.

Algorithm 1: The compilation rule for GDR nodes

Input: formula ϕ, set of all relevant domains $\mathcal{D}$
Output: set of chips S

$S \leftarrow \emptyset$;
foreach *domain* $\Omega \in \mathcal{D}$ *s.t. there is* $c \in \phi$ *and* $X \in \mathrm{Vars}(L_c)$ *s.t.* $\delta_c(X) = \Omega$ **do**
  $\phi' \leftarrow \emptyset$;
  $x \leftarrow$ a new constant in domain Ω;
  foreach *clause* $c = (L, E, \delta) \in \phi$ **do**
    $V \leftarrow \{\, X \in \mathrm{Vars}(L) \mid \delta(X) = \Omega \,\}$;
    foreach $W \subseteq V$ *s.t.* $W^2 \cap E = \emptyset$ **and** $W \cap \{\, X \in \mathrm{Vars}(E) \mid (X, y) \in E$ *for some constant* $y \,\} = \emptyset$ **do**
      /* δ' restricts δ to the new set of variables */
      $\phi' \leftarrow \phi' \cup \{\, (L[x/W], E[x/W] \cup \{\, (X, x) \mid (X \in V \setminus W) \,\}, \delta') \,\}$;
  $S \leftarrow S \cup \{\, (\mathrm{GDR}, \langle \phi' \rangle) \,\}$;

4.1 Generalised Domain Recursion

The main idea behind domain recursion (the original version by Van den Broeck (2011) and the one presented here) is as follows. Let $\Omega \in \mathcal{D}$ be a domain. Assuming that $\Omega \neq \emptyset$, pick some $x \in \Omega$. Then, for every variable $X \in \Omega$ that occurs in a literal, consider two possibilities: $X = x$ and $X \neq x$.

Example 5. *Let ϕ be a formula with a single clause*

$$(\{\, \neg \mathrm{p}(X, Y), \neg \mathrm{p}(X, Z) \,\}, \{\, (Y, Z) \,\}, \{\, X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \,\}).$$

Then we can introduce constant $x \in \Gamma$ and rewrite ϕ as $\phi' = \{\, c_1, c_2 \,\}$, where

$$c_1 \coloneqq (\{\, \neg \mathrm{p}(x, Y), \neg \mathrm{p}(x, Z) \,\}, \{\, (Y, Z) \,\}, \{\, Y \mapsto \Delta, Z \mapsto \Delta \,\}),$$

$$c_2 \coloneqq (\{\, \neg \mathrm{p}(X, Y), \neg \mathrm{p}(X, Z) \,\}, \{\, (X, x), (Y, Z) \,\}, \{\, X \mapsto \Gamma', Y \mapsto \Delta, Z \mapsto \Delta \,\}),$$

and $\Gamma' \coloneqq \Gamma \setminus \{\, x \,\}$.

Van den Broeck (2011) imposes stringent conditions on the input formula to ensure that the expanded version of the formula (as in Example 5) can be handled efficiently. For instance, Example 1 cannot be handled by FORCLIFT because there is no root binding class, i.e., the two root variables belong to different equivalence classes with respect to the binding relationship. The clauses in this expanded formula are then partitioned into three parts based on whether the transformation introduced constants, constraints, or both. The conditions ensure that these parts can be treated independently.

In contrast, GDR has only one precondition: for GDR to be applicable to domain $\Omega \in \mathcal{D}$, there must be at least one

variable with domain Ω featured in a literal (and not just in constraints). Without such variables, GDR would have no effect on the formula. The expanded formula is then left as-is to be handled by other compilation rules. Typically, after a few more rules are applied, a combination of CR and REF nodes introduces a cycle-inducing edge back to the GDR node, thus completing the definition of a recursive function. The GDR compilation rule is summarised as Algorithm 1 and explained in more detail using the example below.

Example 6. *Let $\phi \coloneqq \{ c_1, c_2 \}$ be the formula from Example 1. While GDR is possible on both domains, we illustrate how it works on Γ. The algorithm iterates over the clauses of ϕ. Suppose line 5 picks $c = c_1$ as the first clause. Then, set V is constructed to contain all variables with domain $\Omega = \Gamma$ that occur in the literals of clause c. In this case, $V = \{ X \}$.*

Line 7 iterates over all subsets $W \subseteq V$ of variables that can be replaced by a constant without resulting in evidently unsatisfiable formulas. We impose two restrictions on W. First, $W^2 \cap E = \emptyset$ ensures that no pairs of variables in W are constrained to be distinct since that would result in an $x \neq x$ constraint after substitution. Similarly, we want to avoid variables in W that have inequality constraints with constants: after the substitution, such constraints transform into inequality constraints between two constants. In this case, both subsets of V satisfy the conditions, and line 8 generates two clauses:

$$(\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(X,Z) \}, \{ (Y,Z), (X,x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \}),$$

from $W = \emptyset$ and

$$(\{ \neg\mathrm{p}(x,Y), \neg\mathrm{p}(x,Z) \}, \{ (Y,Z) \}, \{ Y \mapsto \Delta, Z \mapsto \Delta \})$$

from $W = V$.

When line 5 picks $c = c_2$, we have $V = \{ X, Z \}$. The subset $W = V$ fails to satisfy the conditions on line 7 because of the $X \neq Z$ constraint. The other three subsets of V all generate clauses for ϕ'. Indeed, $W = \emptyset$ generates

$$(\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(Z,Y) \}, \{ (X,Z), (X,x), (Z,x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Gamma \}),$$

$W = \{ X \}$ generates

$$(\{ \neg\mathrm{p}(x,Y), \neg\mathrm{p}(Z,Y) \}, \{ (Z,x) \}, \{ Y \mapsto \Delta, Z \mapsto \Gamma \}),$$

and $W = \{ Z \}$ generates

$$(\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(x,Y) \}, \{ (X,x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta \}).$$

Theorem 1 (Correctness of GDR)**.** *Let ϕ be the formula used as input to Algorithm 1, $\Omega \in \mathcal{D}$ the domain selected on line 2, and ϕ' the formula constructed by the algorithm for Ω. Suppose that $\Omega \neq \emptyset$. Then $\phi \equiv \phi'$.*

Theorem 1 is equivalent to Proposition 3 by Van den Broeck (2011). For the proofs of this and other theorems, see supplementary material.

Algorithm 2: The compilation rule for CR nodes

Input: formula ϕ, set of all relevant domains $\mathcal{D}$
Output: set of chips S

1 $S \leftarrow \emptyset$;
2 **foreach** *replaceable pair* $(\Omega \in \mathcal{D}, x \in \Omega)$ **do**
3 add a new domain Ω' to $\mathcal{D}$;
4 $\phi' \leftarrow \emptyset$;
5 **foreach** *clause* $(L, E, \delta) \in \phi$ **do**
6 $E' \leftarrow \{ (a,b) \in E \mid b \neq x \}$;
7 $\delta' \leftarrow X \mapsto \begin{cases} \Omega' & \text{if } \delta(X) = \Omega \\ \delta(X) & \text{otherwise;} \end{cases}$
8 $\phi' \leftarrow \phi' \cup \{ (L, E', \delta') \}$
9 $S \leftarrow S \cup \{ (\mathrm{CR}_{\Omega \mapsto \Omega'}, \langle \phi' \rangle) \}$;

4.2 Constraint Removal

Recall that GDR on a domain Ω creates constraints of the form $X_i \neq x$ for some constant $x \in \Omega$ and family of variables $X_i \in \Omega$. Once some conditions are satisfied, we can eliminate these constraints and replace Ω with a new domain $\Omega' \coloneqq \Omega \setminus \{ x \}$. These conditions are that a constraint of the form $X \neq x$ exists for all variables $X \in \Omega$ across all clauses, and such constraints are the only place where x occurs. We formalise the conditions as Definition 4.

Definition 4. *For a formula ϕ, a pair (Ω, x) of a domain $\Omega \in \mathcal{D}$ and its element $x \in \Omega$ is called* replaceable *if (i) x does not occur in any literal of any clause of ϕ, and (ii) for each clause $c = (L, E, \delta_c) \in \phi$ and variable $X \in \mathrm{Vars}(c)$, either $\delta_c(X) \neq \Omega$ or $(X, x) \in E$.*

Once a replaceable pair is found, Algorithm 2 constructs the new formula by removing constraints and defining a new domain map δ' that replaces Ω with Ω'.

Example 7. *Let $\phi = \{ c_1, c_2 \}$ be a formula with clauses*

$$c_1 = (\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(X,Z) \}, \{ (X,x), (Y,Z) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \}),$$

$$c_2 = (\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(Z,Y) \}, \{ (X,x), (Z,X), (Z,x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Gamma \}).$$

Domain Γ and its element $x \in \Gamma$ satisfy the preconditions for CR. The rule introduces a new domain Γ' and transforms ϕ to $\phi' = (c'_1, c'_2)$, where

$$c'_1 = (\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(X,Z) \}, \{ (Y,Z) \}, \{ X \mapsto \Gamma', Y \mapsto \Delta, Z \mapsto \Delta \}),$$

$$c'_2 = (\{ \neg\mathrm{p}(X,Y), \neg\mathrm{p}(Z,Y) \}, \{ (Z,X) \}, \{ X \mapsto \Gamma', Y \mapsto \Delta, Z \mapsto \Gamma' \}).$$

Theorem 2 (Correctness of CR)**.** *Let ϕ be the input formula of Algorithm 2, (Ω, x) a replaceable pair, and ϕ' the output formula for when (Ω, x) is selected on line 2. Then $\phi \equiv \phi'$, where the domain Ω' introduced on line 3 is interpreted as $\Omega \setminus \{ x \}$.*

Algorithm 3: The compilation rule for REF nodes

Input: formula ϕ, cache C
Output: a set of chips

1 **foreach** *formula and node* $(\psi, v) \in C(\#\phi)$ **do**
2 $\rho \leftarrow \mathtt{r}\,(\phi,\, \psi)$;
3 **if** $\rho \neq \mathtt{null}$ **then return** $\{\, (\text{REF}_\rho(v), \langle\rangle)\,\}$;
4 **return** $\emptyset$;
5 **Function** r (*formula* ϕ, *formula* ψ, *map* $\rho = \emptyset$) **:**
6 **if** $|\phi| \neq |\psi|$ **or** $\#\phi \neq \#\psi$ **then return** null;
7 **if** $\phi = \emptyset$ **then return** ρ;
8 **foreach** *clause* $c \in \psi$ **do**
9 **foreach** *clause* $d \in \phi$ *s.t.* $\#d = \#c$ **do**
10 **foreach** $\gamma \in \mathtt{genMaps}\,(c,\, d,\, \rho)$ **do**
11 $\rho' \leftarrow \mathtt{r}\,(\phi \setminus \{\, d\,\},\, \psi \setminus \{\, c\,\},\, \rho \cup \gamma)$;
12 **if** $\rho' \neq \mathtt{null}$ **then return** ρ';
13 **return** null;

There is no analogue to CR in previous work on first-order knowledge compilation. CR plays a key role by recognising when the constraints of a formula essentially reduce the size of a domain by one and extracting this observation into the definition of a new domain. This then allows us to relate maps between sets of domains to the arguments of a function call. Sections 4.3 and 5 describe this process.

4.3 Identifying Opportunities for Recursion

Hashing We use (integer-valued) hash functions to discard pairs of formulas too different for recursion. The hash code of a clause $c = (L, E, \delta_c)$ (denoted by $\#c$) combines the hash codes of the sets of constants and predicates in c, the numbers of positive and negative literals, the number of inequality constraints $|E|$, and the number of variables $|\operatorname{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted by $\#\phi$.

Caching FORCLIFT (Van den Broeck et al. 2011) uses a cache to check if a formula is identical to one of the formulas that have already been fully compiled. To facilitate recursion, we extend the caching scheme to include formulas encountered before but not fully compiled yet. Formally, we define a *cache* as a map from integers (e.g., hash codes) to sets of pairs of the form (ϕ, v), where ϕ is a formula, and v is an FCG node.

Algorithm 3 describes the compilation rule for creating REF nodes. For every formula ψ in the cache s.t. $\#\psi = \#\phi$, function r checks whether a recursive call is feasible. If it is, r returns a (total) map $\rho\colon \operatorname{Doms}(\psi) \to \operatorname{Doms}(\phi)$ that shows how ψ can be transformed into ϕ by replacing each domain $\Omega \in \operatorname{Doms}(\psi)$ with $\rho(\Omega) \in \operatorname{Doms}(\phi)$. Otherwise, r returns null to signify that ϕ and ψ are too different for recursion to work. This happens if ϕ and ψ (or their subformulas explored in recursive calls) are structurally different (i.e., the numbers of clauses or the hash codes fail to match) or if a clause of ψ cannot be paired with a sufficiently similar clause of ϕ. Function r works by iterating over pairs of clauses of ϕ and ψ with the same hash codes. For every pair of similar clauses, r calls itself on the remaining clauses until the map $\rho\colon \operatorname{Doms}(\psi) \nrightarrow \operatorname{Doms}(\phi)$ becomes total.

Function genMaps checks the compatibility of a pair of clauses. It considers every possible bijection $\beta\colon \operatorname{Vars}(c) \to \operatorname{Vars}(d)$ and map $\gamma\colon \operatorname{Doms}(c) \to \operatorname{Doms}(d)$ s.t.

$$\begin{array}{ccc} \operatorname{Vars}(c) & \xrightarrow{\beta} & \operatorname{Vars}(d) \\ {\scriptstyle\delta_c}\downarrow & & \downarrow{\scriptstyle\delta_d} \\ \operatorname{Doms}(c) & \xrightarrow{\gamma} & \operatorname{Doms}(d) \\ \downarrow & & \downarrow \\ \operatorname{Doms}(\psi) & \xrightarrow[\rho]{} & \operatorname{Doms}(\phi). \end{array}$$

commutes, and c becomes equal to d when its variables are replaced according to β and its domains replaced according to γ. The function then returns each such γ as soon as possible.

Theorem 3 (Correctness of REF)**.** *Let ϕ be the formula used as input to Algorithm 3. Let ψ be any formula selected on line 1 of the algorithm s.t. $\rho \neq$ null on line 3. Let σ be a domain size function. Then the set of models of $(\psi, \sigma \circ \rho)$ is equal to the set of models of (ϕ, σ).*

As FORCLIFT only supports REF nodes when the two formulas are equal, our approach is much more general and capable of creating function calls as complex as $f(n-k-2)$.

5 Converting FCGs to Function Definitions

Algorithm 4 constructs a list of function definitions from an FCG. The algorithm consists of two main functions: v and visit. The former handles new function definitions, while the latter produces an algebraic interpretation of each node depending on its type. As there are many node types, we only include the ones pertinent to the contributions of this paper; see previous work (Van den Broeck et al. 2011) for information about other types. Given a node v as input, both v and visit return a pair $(\mathsf{e}, \mathsf{funs})$. Here, e is the algebraic expression representing v, and funs is a list of auxiliary functions created while formulating e.

The algorithm gradually constructs two partial maps F and D providing names to functions and domains, respectively. F is a global variable that maps FCG nodes to function names (e.g., f, g). D maps domains to their names, and it is passed as an argument to v and visit. Here, a *domain name* is either a parameter of a function or an algebraic expression consisting of function parameters, subtraction, and parentheses. We call a domain name *atomic* if it is free of subtraction (e.g., m, n); otherwise it is *non-atomic* (e.g., $m-1$, $n-l$). Functions newDomainName and newFunctionName both generate previously-unused names. The latter also takes an FCG node as input and links it with the new name in F.

We assume a fixed total ordering of all domains. In particular, in Example 8 below, let Γ go before Δ. For each node v, let $\mathfrak{D}(v)$ denote the (pre-computed) list of domains, the sizes of which we would use as parameters if we were to

Algorithm 4: Construct functions from an FCG

Input: $\mathcal{D}$—the set of domains of the input formula
Input: s—the source node of the FCG
Data: $\mathsf{F} = \emptyset$ (function names)
Output: a list of function definitions

```
1  D ← { Ω ↦ newDomainName() | Ω ∈ 𝒟 };
   /* Equivalent condition:  s has
      in-degree greater than one.     */
2  if s is a direct successor of a REF node then
3  |  (e, funs) ← v(s, D);
4  |  return funs;
5  f ← newFunctionName(s);
6  (e, funs) ← v(s, D);
7  return ⟨f(⟨D(Ω) | Ω ∈ 𝒟⟩) = e⟩ + funs;
8  Function v(node v, domain names D):
9  |  if v is not a direct successor of a REF node then
10 |  |  return visit(v, D);
11 |  f ← newFunctionName(v);
12 |  D' ← D;
13 |  foreach Ω ∈ 𝔇(v) s.t. D'(Ω) is non-atomic do
14 |  |  D'(Ω) ← newDomainName();
15 |  (e, funs) ← visit(v, D');
16 |  call ← f(⟨D(Ω) | Ω ∈ 𝔇(v)⟩);
17 |  signature ← f(⟨D'(Ω) | Ω ∈ 𝔇(v)⟩);
18 |  return (call, ⟨signature = e⟩ + funs);
19 Function visit(node v, domain names D):
20 |  switch label of v do
21 |  |  case GDR(v') do return v(v', D);
22 |  |  case CR_{Ω↦Ω'}(v') do
23 |  |  |  return v(v', D ∪ { Ω' ↦ (D(Ω) − 1) });
24 |  |  case REF_ρ(v') do
25 |  |  |  args ← ⟨D ∘ ρ(Ω) | Ω ∈ 𝔇(v')⟩;
26 |  |  |  return (F(v')(args), ⟨⟩);
27 |  |  ...
```

define a function that begins at v. As a set, it is computed by iteratively setting $\mathfrak{D}(v) \leftarrow (\bigcup_u \mathfrak{D}(u) \setminus I_v) \cup U_v$ until convergence, where: (i) the union is over the direct successors of v, (ii) I_v is the set of domains *introduced* at node v, and (iii) U_v is the set of domains *used* by v. The set is then sorted according to the ordering.

Example 8. *Here we examine how Algorithm 4 works on the FCG from Figure 3. In this case,* $\mathcal{D} = \{\Gamma, \Delta\}$. *Let* $\mathsf{D}_1 \coloneqq \{\Gamma \mapsto m, \Delta \mapsto n\}$, $\mathsf{D}_2 \coloneqq \mathsf{D}_1 \cup \{\Delta^\top \mapsto l, \Delta^\perp \mapsto (n-l)\}$, *and* $\mathsf{D}_3 \coloneqq \mathsf{D}_2 \cup \{\Gamma' \mapsto (m-1)\}$ *denote versions of* D *at various points throughout the algorithm's execution. Here,* l, m, *and* n *are all arbitrary names generated by* `newDomainName`.

Figure 4 shows the outline of function calls and their return values. For simplicity, we refer to FCG nodes by their types. When v *calls* `visit` *and returns its output unchanged, we shorten the two function calls to* v/visit.

Line 1 initialises D *to* D_1. *Once* v(GDR, D_1) *is called on line 3,* `newFunctionName` *updates* F *to* $\{\text{GDR} \mapsto f\}$, *where* f *is a new function name. There are no non-atomic names in* D_1, *so lines 13 and 14 are skipped, and calls to* visit(GDR, D_1) *and* v/visit($\bigvee$, D_1) *follow. Here,* D_1 *becomes* D_2, *i.e., the* $\bigvee$ *node introduces two domains* $\Delta^\top$ *and* $\Delta^\perp$ *that 'partition'* Δ, *i.e.,* $\mathsf{D}_2(\Delta^\top) + \mathsf{D}_2(\Delta^\perp) = l + (n-l) = n = \mathsf{D}_2(\Delta)$. *Similarly,* visit(CR, D_2) *on line 23 adds* Γ', *replacing* D_2 *with* D_3.

Eventually, visit($\perp$, D_3) *is called. A contradiction node with clause* c *as a parameter is interpreted as one if the clause has groundings and zero otherwise. In this case, the parameter is* $c = (\emptyset, \{(X, Y)\}, \{X \mapsto \Delta^\top, Y \mapsto \Delta^\top\})$ *(not shown in Figure 3). Equivalently,* $\forall X, Y \in \Delta^\top.\ X \neq Y \Rightarrow \perp$, *i.e.,* $\forall X, Y \in \Delta^\top.\ X = Y$, *i.e.,* $\mathsf{D}_3(\Delta^\top) < 2$. *We use the Iverson bracket notation to write*

$$[l < 2] \coloneqq \begin{cases} 1 & \text{if } l < 2 \\ 0 & \text{otherwise} \end{cases}$$

for the output expression of visit($\perp$, D_3).

Next, visit($\wedge$, D_3) *calls* v/visit(REF, D_3). *Since* $\mathfrak{D}(\text{GDR}) = \langle\Gamma, \Delta\rangle$, *and the pair* $\langle\Gamma, \Delta\rangle$ *is transformed by* ρ *to* $\langle\Gamma', \Delta^\perp\rangle$ *and then by* D_3 *to* $\langle m-1, n-l\rangle$, args *is set to* $\langle m-1, n-l\rangle$ *on line 25, and the output expression becomes* $f(m-1, n-l)$. *The call to* visit($\wedge$, D_3) *then returns the product of the two expressions* $[l < 2] f(m-1, n-l)$. *Next,* visit($\bigvee$, D_1) *returns* $\sum_{l=0}^{n} [l < 2] f(m-1, n-l)$. *The same expression (and an empty list of functions) is then returned to line 15 of the call to* v(GDR, D_1). *Here, both* call *and* signature *are set to* $f(m, n)$. *As the definition of* f *is the final answer and not part of some other algebraic expression, lines 3 and 4 discard the function call expression*

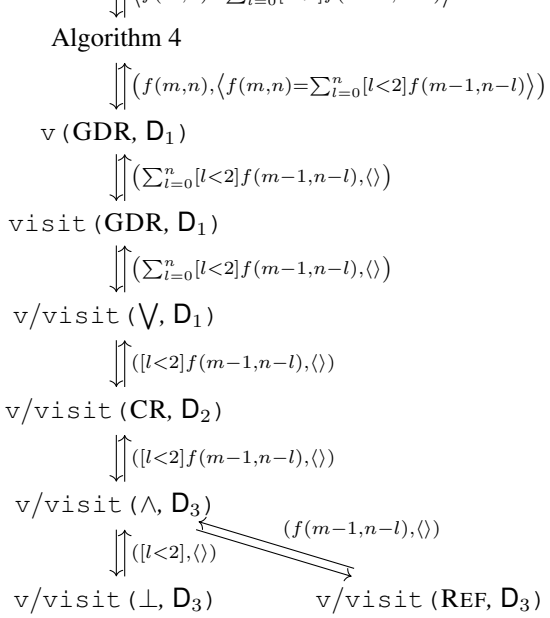


Figure 4: Function calls and their return values when Algorithm 4 is run on the FCG from Figure 3

and return the definition of f. Thus,

$$f(m,n) = \sum_{l=0}^{n} \binom{n}{l} [l < 2] f(m-1, n-l)$$
$$= f(m-1,n) + nf(m-1,n-1) \qquad (2)$$

is the function that CRANE *constructed for computing partial injections. To use f in practice, one has to identify the base cases $f(0,n)$ and $f(m,0)$ for all $m,n \in \mathbb{N}_0$.*

6 Complexity Results

We compare CRANE and FORCLIFT[3] (Van den Broeck et al. 2011) on their ability to count various classes of functions. We chose this class of instances because of its simplicity and the inability of publicly available WFOMC algorithms to solve many such counting problems. Note that, except for a particular version of FASTWFOMC, other exact WFOMC algorithms cannot solve any instances FORCLIFT fails on. First, we describe how to express these function-counting problems in first-order logic. FORCLIFT then translates these sentences in first-order logic to formulas as in Definition 3.

Let $\mathtt{p}$ be a predicate that models relations between sets Γ and Δ. To restrict all such relations to just $\Gamma \to \Delta$ functions, one might write $\forall X \in \Gamma.\, \forall Y, Z \in \Delta.\, \mathtt{p}(X,Y) \wedge \mathtt{p}(X,Z) \Rightarrow Y = Z$ and

$$\forall X \in \Gamma.\, \exists Y \in \Delta.\, \mathtt{p}(X,Y). \qquad (3)$$

The former sentence says that one element of Γ can map to at *most* one element of Δ, and the latter sentence says that each element of Γ must map to at *least* one element of Δ. One can then add $\forall X, Z \in \Gamma.\, \forall Y \in \Delta.\, \mathtt{p}(X,Y) \wedge \mathtt{p}(Z,Y) \Rightarrow X = Z$ to restrict $\mathtt{p}$ to injections or $\forall Y \in \Delta.\, \exists X \in \Gamma.\, \mathtt{p}(X,Y)$ to ensure surjectivity or remove Equation (3) to consider partial functions. Lastly, one can replace all occurrences of Δ with Γ to model endofunctions (i.e., functions with the same domain and codomain) instead.

We consider all sixteen combinations of these properties (injectivity, surjectivity, partiality, and endo-), omitting duplicate descriptions of the same function/sequence. We run each algorithm once on each instance. CRANE is run in hybrid search mode until either it finds five solutions or the search tree reaches height six. FORCLIFT is always run until it terminates. If successful, CRANE returns one or more sets of function definitions, and FORCLIFT returns a circuit, which can similarly be interpreted as a function definition. We then assess the complexity of each solution by hand and pick the best in case CRANE returns several solutions of varying complexities.

Recall that, in the case of FORCLIFT, solving a (W)FOMC problem consists of two parts: compilation (via search) and propagation. (A similar dichotomy applies to CRANE as well.) The complexity of the former depends only on the formula and so far has received very little attention except for the work by Beame et al. (2015). The complexity of the latter, on the other hand, is a function of domain sizes and can be determined without measuring runtime. We establish the asymptotic complexity of a solution by counting the number of arithmetic operations needed to follow the definitions of constituent functions without recomputing the same quantity multiple times. In particular, we assume that each function call and binomial coefficient is computed at most once, and computing $\binom{n}{k}$ takes $\Theta(nk)$ time. For example, the complexity of Equation (2) is $\Theta(mn)$ since $f(m,n)$ can be computed by a dynamic programming algorithm that computes $f(i,j)$ for all $i = 0,\ldots,m$ and $j = 0,\ldots,n$, taking a constant amount of time on each $f(i,j)$.

Let $m = |\Gamma|$ and $n = |\Delta|$ be domain sizes. We summarise the results in Table 1, where we compare the solutions found by both algorithms to the manually-constructed ways of computing the same quantities that have been proposed before.[4] Examining the last two columns of the table, we see that the two algorithms perform equally well on instances that could already be solved by FORCLIFT. However, CRANE can also solve all but one of the instances that FORCLIFT fails on in at most cubic time. See supplementary material for the exact solutions produced by CRANE.

6.1 A Comparison with FastWFOMC

The function-counting problems described above can be expressed in FO^2 with cardinality constraints, and so are known to be liftable (Kuželka 2021). However, two algorithms that both run in polynomial time are not necessarily equally good: the degree of the polynomial can make a substantial difference. Hence, we compare CRANE with FASTWFOMC (van Bremen and Kuželka 2021a), extended to support cardinality constraints, which was provided to us by one of the authors. We compare them on the task of counting permutations of a set, i.e., bijective/injective endofunctions. We can describe this problem in FO^2 with cardinality constraints as

$$(|\mathtt{p}| = |\Delta|) \wedge (\forall X, Y \in \Delta.\, \mathtt{r}(X) \vee \neg\mathtt{p}(X,Y)) \wedge$$
$$(\forall X, Y \in \Delta.\, \mathtt{s}(X) \vee \neg\mathtt{p}(Y,X)),$$

where $\mathtt{r}$ and $\mathtt{s}$ are Skolem predicates as described by Van den Broeck et al. (2014).

We run FASTWFOMC on an AMD Ryzen 7 5800H processor with 16 GB of memory, running Arch Linux 6.2.2-arch1-1 and Python 3.10.9, with $|\Delta|$ ranging from 1 to 50. The results are in Figure 5. We use ten-fold cross-validation to check the degree of the polynomial that best fits the data. The best-fitting degree is 6 with an average mean squared error (aMSE) of 16.9, although all degrees above 4 fit the data almost equally well. As Figure 5 demonstrates, a quartic (i.e., degree 4) polynomial (with an aMSE of 113.5) fits well too. However, the aMSE quickly grows to 949.5 and higher values for degrees less than 4. According to Table 1, CRANE finds a cubic solution to this problem. However, we can also use the linear solution for counting bijections between different domains for this task by setting $n \coloneqq m$.

6.2 Miscellaneous Other Instances

We also note that standard (W)FOMC benchmarks such as Example 3—that are already supported by FORCLIFT—

[3] https://dtaid.cs.kuleuven.be/wfomc

[4] https://oeis.org, https://scicomp.stackexchange.com/q/30049

Function class			Complexity of $\mathcal{F}_\phi$ (as in Figure 1)		
Partial	Endo-	Class	By Hand	With FORCLIFT	With CRANE
✓/✗	✓/✗	Functions	$\log m$	m	m
✗	✗	Surjections	$n \log m$	$m^3 + n^3$	$m^3 + n^3$
✗	✓		$m \log m$	m^3	m^3
✗	✗	Injections	m	—	mn
✗	✓		m	—	m^3
✓	✗		$\min\{ m, n \}^2$	—	mn
✓	✓		m^2	—	—
✗	✗	Bijections	m	—	m

Table 1: The worst-case complexity of counting various types of functions. All asymptotic complexities are in $\Theta(\cdot)$. A dash means that the algorithm was not able to find a solution. In the case of FORCLIFT, this means that the greedy search algorithm ended with a formula unsuitable for any compilation rule. In the case of CRANE, this means that a complete solution could not be found after having explored the maximum allowed depth of the search tree.

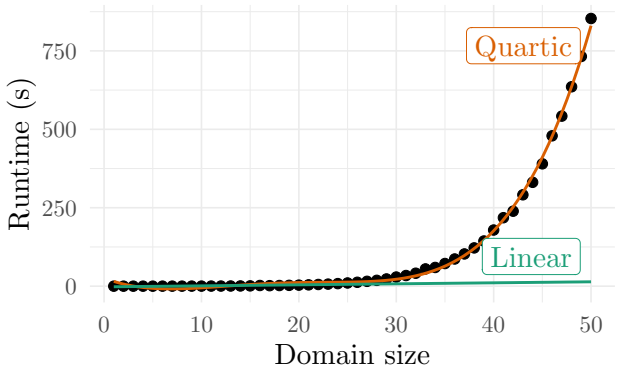


Figure 5: The runtime of FASTWFOMC when counting permutations, together with a quartic regression model fitted on all of the data and a linear model fitted on the first half of the data points

remain feasible for CRANE as well. To demonstrate that CRANE is capable of finding efficient solutions to problems beyond known domain-liftable fragments such as C^2, we run CRANE in greedy search mode on the following formula:

$$\begin{aligned}&(\forall X, W \in \Gamma.\ \forall Y \in \Delta.\ \forall Z \in \Lambda.\\ &\quad \mathtt{p}(X, Y, Z) \wedge \mathtt{p}(W, Y, Z) \Rightarrow X = W) \wedge\\ &(\forall X \in \Gamma.\ \forall Y, W \in \Delta.\ \forall Z \in \Lambda.\\ &\quad \mathtt{p}(X, Y, Z) \wedge \mathtt{p}(X, W, Z) \Rightarrow Y = W).\end{aligned} \tag{4}$$

For every element z of Λ, Formula (4) restricts $\mathtt{p}(X, Y, z)$ to be a $\Gamma \to \Delta$ partial injection, independent of $\mathtt{p}(X, Y, z')$ for any $z' \neq z$. CRANE instantly returns a solution of complexity $\Theta(l + mn)$, where $l = |\Lambda|$.

7 Conclusion and Future Work

This paper presents CRANE—an extension of FORCLIFT (Van den Broeck et al. 2011) that benefits from a more general version of domain recursion and support for graphs with cycles. Our results in Section 6 reveal a range of counting problems that became newly feasible as a result of these enhancements, including instances outside of currently-known domain-liftable fragments such as C^2. (Although we focused on unweighted counting, FORCLIFT's support for weights trivially transfers to CRANE too.) The common thread across these newly liftable problems is a version of partial injectivity. Thus, we formulate the following conjecture.

Conjecture 1. *Let* IFO^2 *be the class of formulas in first-order logic that contain clauses with at most two variables as well as any number of copies of*

$$\begin{aligned}&(\forall X_1 \in \Delta_1.\ \ldots \forall X_n \in \Delta_n.\ \forall Y \in \Delta_i.\ \mathtt{p}(X_1, \ldots, X_n) \wedge\\ &\qquad \mathtt{p}(X_1, \ldots, X_{i-1}, Y, X_{i+1}, \ldots, X_n) \Rightarrow X_i = Y) \wedge\\ &(\forall X_1 \in \Delta_1.\ \ldots \forall X_n \in \Delta_n.\ \forall Y \in \Delta_j.\ \mathtt{p}(X_1, \ldots, X_n) \wedge\\ &\qquad \mathtt{p}(X_1, \ldots, X_{j-1}, Y, X_{j+1}, \ldots, X_n) \Rightarrow X_j = Y)\end{aligned}$$

for some predicate $\mathtt{p}/n$*, domains* $\Delta_1, \ldots, \Delta_n$*, and* $i, j \in \{ 1, \ldots, n \}$*. Then* IFO^2 *is liftable by* CRANE.

Since most of the instances in Section 6 are in C^2, we can also conjecture that C^2 is liftable by CRANE.

The most important direction for future work is to automate the process in Figure 1b. First, we need a way to find the base cases for the recursive definitions produced by CRANE. Second, since the first solution found by CRANE is not always optimal in terms of its complexity, an automated way to determine the asymptotic complexity of a solution would be helpful as well. Third, as we saw at the end of Example 8, the functions constructed by CRANE can often benefit from elementary algebraic simplifications. This process can be automated using a computer algebra system. Furthermore, note that having the solution to a (W)FOMC problem expressed in terms of functions enables many new possibilities such as (i) the use of more sophisticated simplification techniques, (ii) asymptotic analysis of, e.g., how the model count grows as the domain size goes to infinity, and (iii) answering questions parameterised by domain sizes, e.g., 'how big does domain Δ have to be for the probability of event E to be above 95 %?' In addition to the potential impact on areas of artificial intelligence (AI) like statistical relational AI (De Raedt et al. 2016), CRANE could be beneficial to research in combinatorics as well (Barvínek et al. 2021).

References

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

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

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

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

De Raedt, L.; Kersting, K.; Natarajan, S.; and Poole, D. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gatterbauer, W., and Suciu, D. 2015. Approximate lifted inference with probabilistic databases. *Proc. VLDB Endow.* 8(5):629–640.

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

Golia, P.; Roy, S.; and Meel, K. S. 2020. Manthan: A data-driven approach for Boolean function synthesis. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, 611–633. Springer.

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.

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.

Kimmig, A.; Mihalkova, L.; and Getoor, L. 2015. Lifted graphical models: A survey. *Mach. Learn.* 99(1):1–45.

Kuncak, V.; Mayer, M.; Piskac, R.; and Suter, P. 2010. Complete functional synthesis. In *PLDI*, 316–329. ACM.

Kuusisto, A., and Lutz, C. 2018. Weighted model counting beyond two-variable logic. In *LICS*, 619–628. ACM.

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.

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

Sanathanan, C., and Koerner, J. 1963. Transfer function synthesis as a ratio of two complex polynomials. *IEEE transactions on automatic control* 8(1):56–58.

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. 2021a. 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. 2021b. Lifted inference with tree axioms. In *KR*, 599–608.

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.; Choi, A.; and Darwiche, A. 2012. Lifted relax, compensate and then recover: From approximate to exact lifted probabilistic inference. In *UAI*, 131–141. AUAI Press.

Van den Broeck, G.; Meert, W.; and Darwiche, A. 2014. Skolemization for weighted first-order model counting. In *KR*. AAAI Press.

Van den Broeck, G. 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.