TBD

Abstract

1 Introduction

- mention/describe the arithmetic simplification code
- (in Fig. 1) Crane2 doesn't use Crane, it uses itself
- describe which compilation rules are currently greedy or non-greedy
- my agenda has a plan for proving completeness for $\mathbf{C}^2$
- Can I prove that Crane's domain recursion can always simulate ForcLift's domain recursion?

Changes to the compilation process.

- We make the FCGs satisfy the following property. For every domain Δ, for every directed path P without Ref nodes, the number of atom counting or (generalised) domain recursion or independent partial grounding operations on Δ on P is at most one.
- when checking whether recursion is possible between a source and a target formula, the target formula (maybe the source formula as well?) must not include a domain as well as its subdomain. This might be equivalent to the formula being 'unshattered'. Such formulas could be seen as transitory, and allowing such recursive calls leads to FCGs that I cannot make sense of.
- Implemented an if-then-else arithmetic operation as the interpretation of the new domain recursion node (i.e., there is one branch for when the domain is empty and another branch for any other domain size).
- Sections 5 and 6 might also be outdated: there is probably no need to search for base cases given the new version of domain recursion.

Proof of correctness (leave for later).

1. Define what it means for a formula to be *consistent*.
   - The formula (including the mapping of constants to their domains) does not mention a domain together with its subdomain (i.e., all mentioned domains are pairwise disjoint). Specifically, we can't have a subdomain together with its parent domain or a subdomain together with a constant that belongs to its parent domain.
   - The target of `Ref` must also be a consistent formula.
   - The initial formula is consistent by definition.
2. Aim to show that Crane works correctly on all consistent formulas.
3. Prove that, whenever some compilation rule makes the formula ϕ inconsistent, greedy compilation rules will make ϕ consistent again before ϕ encounters non-greedy compilation rules that depend on ϕ being consistent.
   - In particular, prove that shattering and unit propagation will always eliminate the parent domain of the domains introduced by atom counting.
   - Constraint removal only applies when it can eliminate the parent domain entirely.
   - Prove this for a particular assumption about which compilation rules are set to be greedy. For the greedy algorithm, although all rules are applied in a greedy manner, we still assume that greedy rules will be applied before non-greedy rules.
4. In particular, independent partial groundings and atom counting are the two classical rules that require the formula to be shattered.

The evaluation of base cases is done by simplifying the clauses and then using CRANE to find the base cases. First, while traversing the graph to find the equations, we store two maps: $\mathcal{F}$ (which stores the mapping from the function names to the formulae, whose model count they represent) and $\mathcal{D}$ (which stores the mapping from the variable names to the domains whose sizes they represent). Then, a particular domain is selected (using the algorithm described in previous reports), and the clauses are simplified. Then, CRANE is called on those clauses to evaluate the base cases. After that, we change the function names and variable to make it consistent with the previous domain to variable mapping, and append these base cases to the set of equations.

Contributions (outdated).

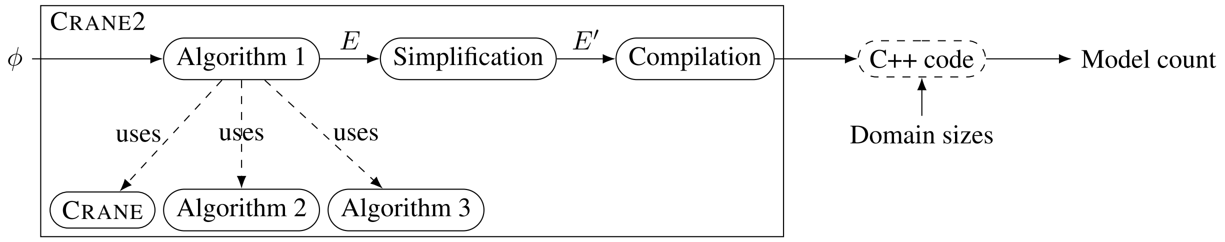


Figure 1: Using CRANE2 to compute the model count of formula ϕ. The formula is compiled into a set of equations E, which are then algebraically simplified and compiled to a C++ program. This program can then be run with different command line arguments to compute the model count of ϕ for various domain sizes.

- Section 6
- Section 7
- Converting the recursive equations into a C++ program, which can then be compiled and executed to obtain numerical values (see Section 8).
- Support for infinite precision integers using the GNU Multiple Precision Arithmetic Library.
- A new set of benchmarks.
- The experiments for Crane and ForcLift are (going to be) more comprehensive than any other experimental study performed on (W)FOMC algorithms.
- Some experiments vs FastWFOMC
- A proof of correctness
- A proof of completeness

references to my previous paper and some of the FORCLIFT and FASTWFOMC papers that contain experimental results

Sections 6 and 8 deal with algebraic constructs whereas Section 7 deals with logic.

2 Preliminaries

2.1 Algebra

Notation. We write `expr` for an arbitrary algebraic expression. For any signature or clause C, argument or variable x, and number or constant t, we shall write $C[t/x]$ for the result of substituting t for all occurrences of x in C.

- Variables in uppercase
- Domains in uppercase Greek
- Constant and predicate symbols in lowercase
- Clauses written on separate lines are implicitly conjoined
- For weights, for any predicate symbol p, we write $w^+(p), w^-(p) \in \mathbb{R}$ for its positive and negative weights, respectively. All weights not explicitly acknowledged are assumed to be equal to one.

- check if I'm still using this notation in all these cases
- explain how this works for function calls too
- introduce WFOMC as a problem

Definition 1. *A* function call *is a term of the form $f(x_1 - c_1, \dots, x_n - c_n)$ (written $f(\mathbf{x} - \mathbf{c})$ for short), where f is an n-ary function, each x_i is a variable, and each c_i is a non-negative constant.*

do I ever use this term for the case when some of the variables don't exist?

Definition 2. *A* signature *is a term of the form $f(x_1, \dots, x_n)$ (written $f(\mathbf{x})$ for short), where f is an n-ary function, and each x_i is a variable. The signature of a function call $f(\mathbf{x}-\mathbf{c})$ is $f(\mathbf{x})$. For example, the signature of $f(x-1, y-2)$ is $f(x, y)$.*

Definition 3. *An* equation *is always of the form $f(\mathbf{x}) =$ `expr`, where $f(\mathbf{x})$ is a signature, and `expr` is an algebraic expression. Henceforth, we call $f(\mathbf{x})$ and `expr` the left-hand side (LHS) and the right-hand side (RHS) of the equation, respectively.*

give some examples of formulas that:
- introduce new variables,
- call the function itself,
- call some other function, and
- is more nested than just a sum.

Definition 4. *A* base case *is a function call where all arguments are either variables or constants (i.e., there is no subtraction within any argument), and there is at least one constant.*

2.2 Logic

- What notation am I using for FOL? Introduce equality cardinality constraints, counting quantifiers, etc.
- Need (W)FOMC semantics. What is an interpretation? What is a model?
- Should table caption go above or below the table?
- Define terms.
- Maybe say "structure" instead of interpretation?
- What's the right table heading capitalization for KR?
- Should I say "sort" instead of domain everywhere?

Logic	Sorts	Constants	Variables	Quantifiers	Additional atomic constraints
FO	one or more	✓	unlimited	$\forall$, $\exists$	$s = t$
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) atomic constructions in addition to those of the form $p(t_1, \dots, t_n)$ for some predicate symbol p and terms $t_1, \dots, t_n$. Here: (i) k and m are non-negative integers, the latter of which can depend on the domain size, (ii) p is a predicate symbol, and (iii) s and t are terms.

Three types of logics

- All three logics are function-free.
- Domains are always assumed to be finite.
- See Table 1 for a detailed comparison. FO is used as the input format for CRANE and FORCLIFT. C^2 is used in the literature on FASTWFOMC and related methods (TODO: citations). $\mathsf{UFO}^2 + \mathsf{EQ}$ is the input format supported by a privately-obtained version of FASTWFOMC. Note that the publicly available version does not support any cardinality constraints.
- In the case of CRANE and FORCLIFT, 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 (TODO: citation).

A *formula* is a set of clauses. A *clause* is of the form $\forall x_1 \in \Delta_1 \forall x_2 \in \Delta_2 \dots \forall x_n \in \Delta_n \phi(x_1, x_2, \dots, x_n)$, where ϕ is a disjunction of literals that only contain variables $x_1, \dots, x_n$ (and any constants). A *literal* is either an atom or its negation. An *atom* is of the form $P(t_1, \dots, t_m)$, where P is a predicate, and $t_1, \dots, t_m$ are terms. A *term* is either a variable or a constant.

explain set-based notation for a clause (iterating over all literals)

For any clause C, let $\mathrm{Preds}(C)$ denote the set of predicates in C. For any clause or literal l, let $\mathrm{Vars}(l)$ denote the set of variables in l.

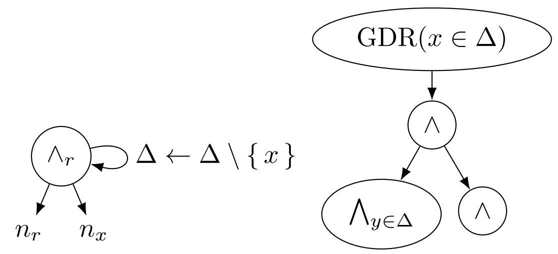
For any (bound) variable v, let $\mathrm{Dom}(v)$ denote the domain over which v is quantified. For a literal l, $\mathrm{Doms}(l) := \{\, \mathrm{Dom}(v) \mid v \in \mathrm{Vars}(l) \,\}$. Similarly, for any clause C, let $\mathrm{Doms}(C) = \{\, \mathrm{Dom}(v) \mid v \in \mathrm{Vars}(C) \,\}$.

3 Theoretical Results

Theorem 1. *The domain recursion by Dilkas and Belle (2023) can simulate the domain recursion by Van den Broeck (2011). This proves that Crane can (still) handle* FO^2.

Proof. 'A theory allows for domain recursion when the theory is shattered, the theory contains no independent subtheories, and there exists a root binding class.'

Stuck on some weird cases. Need to prove that the model count of n_x will never depend on $|\Delta|$. $\square$


$\wedge_r$
$\Delta \leftarrow \Delta \setminus \{\, x \,\}$
n_r
n_x
$\mathrm{GDR}(x \in \Delta)$
$\wedge$
$\bigwedge_{y \in \Delta}$
$\wedge$

4 Smoothing (only relevant for domain recursion v3)

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

- Can I prove that the additions to smoothing outlined in the rest of this section 'do the right thing'?
- I could include more info about the smoothing algorithm that didn't make it into the previous paper.

4.1 Stage 1: Propagating Unit Clauses 'Upwards'

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

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

Domain recursion. Suppose the domain recursion node introduces constant $x \in \Delta$. For each unit clause received from the child node, replace each occurrence of $\phi(x)$ or $\forall X \in \Delta.\ X \neq x \Rightarrow \phi(X)$ with $\forall X \in \Delta.\ \phi(X)$. This can be seen as a claim about what ground atoms the domain

Algorithm 1: Finding the definitions of the base cases

```
   Input: formula $\phi$
   Output: a set $E$ of equations that define $B$
1  $(E, \mathcal{F}, \mathcal{D}) \leftarrow$ Crane($\phi$);
2  foreach base case $f(\mathbf{x}) \in$ FindBaseCases($E$) do
3      $\psi \leftarrow \mathcal{F}(f)$;
4      foreach $i$ such that $x_i$ is a constant do
5          $\psi \leftarrow$ Propagate($\psi$, $\mathcal{D}(f, i)$, $x_i$);
6      $(E', \_, \_) \leftarrow$ Crane($\psi$);
7      $E \leftarrow E \cup E'$;
```

recursion node *should* cover (or a temporary assumption). If the relevant subgraph indeed covers those ground atoms, Stage 2 will do nothing. Otherwise, smoothing nodes will be added below the domain recursion node to cover the difference between what was propagated from the domain recursion node and what was received from the child node.

Add examples of both cases, with figures.

4.2 Stage 2: Adding Smoothing Nodes

`Ref`: nothing.

Constraint removal: nothing.

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

2. Add smoothing nodes below the domain recursion node for the difference between the unit clauses assigned to the domain recursion node during Stage 1 and the unit clauses of the child node post-processed by the step above. For example, if the child node 'covers' only $p(x)$, then Stage 1 assigns $\forall X \in \Delta.\ p(X)$ to the domain recursion node. The smoothing node below the domain recursion node then has the clause $\forall X \in \Delta.\ X \neq x \Rightarrow p(X)$.

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

5 The Main Algorithm: Finding the Definitions of the Base Cases

See Algorithm 1

- extend it to work with base cases that are themselves recursive
- what does it mean for a set of equations to define a set of base cases?
- for base cases, we will also use the $f(\mathbf{x})$ notation

Algorithm 2: `FindBaseCases(`E`)`

```
   Input: set $E$ of equations
   Output: set $B$ of base cases
1  $B \leftarrow \emptyset$;
2  foreach equation $(f(\mathbf{x}) = \texttt{expr}) \in E$ do
3      foreach function call $f(\mathbf{x} - \mathbf{c}) \in \texttt{expr}$ do
4          foreach $c_i \in \mathbf{c}$ do
5              for $n \leftarrow 0$ to $c_i - 1$ do
6                  $B \leftarrow B \cup \{ f(\mathbf{x})[x_i/n] \}$;
```

6 Identifying a Sufficient Set of Base Cases

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

- NOTE: line 3 iterates over all function calls in the algebraic expression
- remove the 'dependencies' data structure from the description

The following steps were followed while finding the base cases:

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

Now, we find a domain that has only terms of the form $x - 1$ appearing on the RHS of the dependencies. The base cases are then calculated by setting this domain size to zero. For the above example, n is the selected domain, and not m since there are $m - 2$ and $m - 3$ terms appearing in the arguments.

The algorithm is described as Algorithm 2.

add an example where Algorithm 2 runs on either

$$f_0(m, n) = f_1(m - 1, n) + f_2(m, n - 1)$$
$$f_1(m, n) = f_1(m - 1, n - 1) \times f_2(m - 2, n - 1)$$
$$f_2(m, n) = 2 \times f_1(m - 3, n - 1)$$

or

$$f(m, n) = g(m - 1, n) + f(m - 2, n - 1)$$
$$g(m, n) = f(m - 1, n - 2) + g(m - 1, n - 1).$$

- line 5: the limit is 2 for the arg $(x - 3)$.

- line 6 for $f(\mathbf{x}) = f(y,z)$, $x_i = y$, and $n = 0$, add $f(y,z)[y/0] = f(0,z)$ to B.

I changed the algorithm. Update the description.

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

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

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

NOTE: function calls such as $f(n-m)$ and $f(n-m-1)$ are ignored.

7 Propagating Domain Size Assumptions

7.1 Motivation: Why Tautologies Are Needed and Simply Removing The Clauses Does Not Work

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

For example, consider the formula

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

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

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

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

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

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

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

Algorithm 3: `Propagate`(ϕ, Δ, n)

```
Input: formula $\phi$, domain $\Delta$, domain size $n \in \{0, 1\}$
Output: formula $\phi'$
$\phi' \leftarrow \emptyset$;
if $n = 0$ then
    foreach clause $C \in \phi$ do
        if $\Delta \notin \mathrm{Doms}(C)$ then $\phi' \leftarrow \phi' \cup \{C\}$;
        $C' \leftarrow \{ l \in C \mid \Delta \notin \mathrm{Doms}(l) \}$;
        if $\Delta \in \mathrm{Doms}(C)$ and $C' \neq \emptyset$ then
            $l \leftarrow$ an arbitrary literal in $C'$;
            $\phi' \leftarrow \phi' \cup \{ C' \cup \{\neg l\} \}$;
else
    $c \leftarrow$ a new constant symbol;
    foreach clause $C \in \phi$ do
        $C' \leftarrow C$;
        foreach $v \in \mathrm{Vars}(C)$ with $\mathrm{Dom}(v) = \Delta$ do
            $C' \leftarrow C'[c/v]$;
        $\phi' \leftarrow \phi' \cup \{C'\}$;
```

7.2 The Solution

it would be easy to extend the second part to an arbitrary constant

We use Algorithm 3 to find the transformed formula corresponding to each base case obtained using Algorithm 2 and call CRANE on the formula to obtain the required base cases.

- Similar to generalised domain recursion (what's the difference?).
- Clauses where all literals have variables quantified over the empty domain are still removed.
- C' is guaranteed to be non-empty.

8 Generating C++ Code

the report has some (long) examples of formulas being transformed into programs perhaps suitable for supplementary material

The target is to generate C++ code that can evaluate numerical values of the model counts based on the equations generated by CRANE. We achieve this by parsing the equations generated by CRANE, simplifying them, and then generating C++ code. This approach can be done in linear time in the length of the formula using the Shunting Yard Algorithm.

The translation of a set E of equations into a C++ program works as follows.

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

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

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

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

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

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

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

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

9 Experiments

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

Benchmarks (probably for supplementary material).

- Functions
  - In C^2: $\forall X \in \Delta.\ \exists^{=1} Y \in \Delta.\ p(X, Y)$
  - In $\mathsf{UFO}^2 + \mathsf{EQ}$:

$$\forall X, Y \in \Delta.\ s(X) \vee \neg p(X, Y)$$
$$|p| = |\Delta|$$

  - In FO:

$$\forall X \in \Delta.\ \exists Y \in \Delta.\ p(X, Y)$$
$$\forall X, Y, Z \in \Delta.\ p(X, Y) \wedge p(X, Z) \Rightarrow Y = Z$$

- Permutations

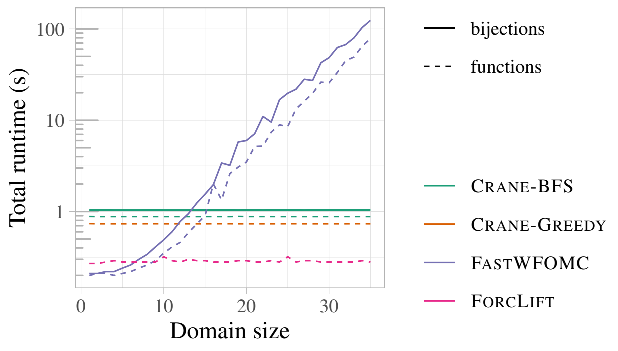


Figure 2: The runtime data of WFOMC algorithms on bijection- and function-counting problems on domains of sizes $1, 2, \ldots, 35$

  - In C^2:

$$\forall X \in \Delta.\ \exists^{=1} Y \in \Delta.\ p(X, Y)$$
$$\forall Y \in \Delta.\ \exists^{=1} X \in \Delta.\ p(X, Y)$$

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

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

  with weights $w^{-}(r) = w^{-}(s) = -1$
  - In FO:

$$\forall X \in \Delta.\ \exists Y \in \Delta.\ p(X, Y)$$
$$\forall Y \in \Delta.\ \exists X \in \Delta.\ p(X, Y)$$
$$\forall X, Y, Z \in \Delta.\ p(X, Y) \wedge p(X, Z) \Rightarrow Y = Z$$
$$\forall X, Y, Z \in \Delta.\ p(X, Y) \wedge p(Z, Y) \Rightarrow X = Z$$

Setup.

- The experiments were run on an AMD Ryzen 7 5800H processor with 16 GiB of memory and Arch Linux 6.8.2-arch2-1 operating system. FASTWFOMC was run using Python 3.8.19 with Python-FLINT 0.5.0.

Results.

- the runtime appears constant because compilation time dominates inference time for small domain sizes. Indeed, the maximum inference time of both CRANE-BFS and CRANE-GREEDY is only 4 ms.
- describe Figure 2
- On bijection counting, FASTWFOMC takes about 124 s on a domain of size 35, whereas CRANE-BFS scales up to 10^4 in 34 s
- On function counting, FASTWFOMC takes about 78 s on a domain of size 35, whereas both CRANE-BFS and CRANE-GREEDY scale up to 3×10^5 in 41 s.
- Note that, both model counts are huge (TODO: add details)

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.

10 Conclusion

References

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

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