

Towards Practical First-Order Model Counting

Abstract

1 Introduction

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.

- Section 4
- Section 5
- Converting the recursive equations into a C++ program, which can then be compiled and executed to obtain numerical values (see Section 6).
- Support for infinite precision integers using the GNU Multiple Precision Arithmetic Library.
- The experiments are (going to be) more comprehensive than any other experimental study performed on (W)FOMC algorithms (TODO: references to my previous paper and some of the FORCLIFT and FASTWFOMC papers that contain experimental results).

Sections 4 and 6 deal with algebraic constructs whereas Section 5 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 .

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.

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 $f(\mathbf{x}) = \text{expr}$, 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.

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

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.

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

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

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

See Algorithm 1

4 Identifying a Sufficient Set of Base Cases

We know that if, say, on the RHS of all equations, the domain size appears as $m - c_1, m - c_2, \dots, m - c_k$, then finding $f(0, x_1, x_2, \dots), f(1, x_1, x_2, \dots), \dots, f(m_0, x_1, x_2, \dots)$

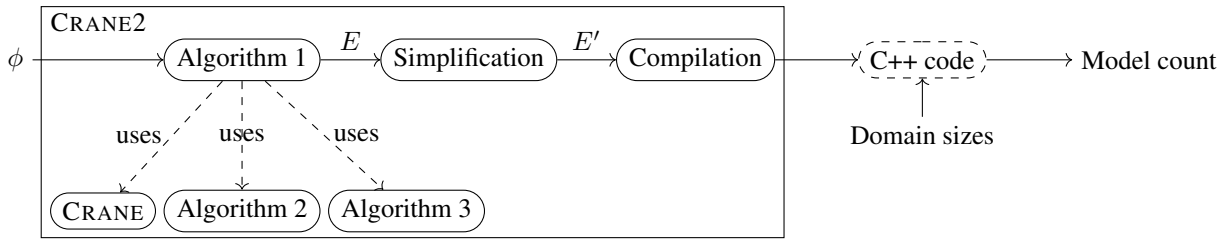


Figure 1: Using CRANE2 to compute the model count of formula ϕ . The formula is compiled into a set of equations E , which are then algebraically simplified and compiled to a C++ program. This program can then be run with different command line arguments to compute the model count of ϕ for various domain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

Algorithm 1: Finding the definitions of the base cases

Input: formula ϕ

Output: a set E of equations that define B

```

1  $(E, \mathcal{F}, \mathcal{D}) \leftarrow \text{Crane}(\phi);$ 
2 foreach base case  $f(\mathbf{x}) \in \text{FindBaseCases}(E)$  do
3    $\psi \leftarrow \mathcal{F}(f);$ 
4   foreach  $i$  such that  $x_i$  is a constant do
5      $\psi \leftarrow \text{Propagate}(\psi, \mathcal{D}(f, i), x_i);$ 
6    $(E', -, -) \leftarrow \text{Crane}(\psi);$ 
7    $E \leftarrow E \cup E';$ 

```

Algorithm 2: $\text{FindBaseCases}(E)$

Input: set E of equations

Output: set B of base cases

```

1  $B \leftarrow \emptyset;$ 
2 foreach equation  $(f(\mathbf{x}) = \text{expr}) \in E$  do
3   foreach function call  $g(\mathbf{y} - \mathbf{c}) \in \text{expr}$  do
4     foreach  $c_i \in \mathbf{c}$  do
5       for  $n \leftarrow 0$  to  $c_i - 1$  do
6          $B \leftarrow B \cup \{f(\mathbf{x})[x_i/n]\};$ 
7         if  $f \neq g$  then  $B \leftarrow B \cup \{g(\mathbf{y})[y_i/n]\};$ 

```

for every function f , where $m_0 = \max(c_1, c_2, \dots, c_k) - 1$ 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. $\text{argmin}(\max(c_1, c_2, \dots, c_k))$. Ideally, we should calculate the base cases by finding the base cases up to $\max(c_1, c_2, \dots) - 1$. However, currently only empty and singleton domains are supported.

- line 3 iterates over all function calls in the algebraic expression.
- note that on line 7 it is the signature and not the original function call.

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: $\sum_{x=a}^b \text{expr} \cdot [a \leq x < b]$ 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.

Example 1. *TODO: 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 .

5 Propagating Domain Size Assumptions

5.1 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

$$\forall x \in \Delta \forall y, z \in \Gamma : P(x) \vee Q(y, z) \quad (1)$$

$$\forall y, z \in \Gamma' : Q(y, z) \quad (2)$$

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

$$\forall y, z \in \Gamma' : Q(y, z), \quad (3)$$

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 $|\Gamma \times \Gamma| \setminus |\Gamma' \times \Gamma'| = |\Gamma|^2 - |\Gamma'|^2$.

Algorithm 3: Propagate (ϕ, Δ, n)

Input: formula ϕ , domain Δ , domain size $n \in \{0, 1\}$ **Output:** formula ϕ'

```
1  $\phi' \leftarrow \emptyset$ ;  
2 if  $n = 0$  then  
3   foreach clause  $C \in \phi$  do  
4     if  $\Delta \notin \text{Doms}(C)$  then  $\phi' \leftarrow \phi' \cup \{C\}$ ;  
5      $C' \leftarrow \{l \in C \mid \Delta \notin \text{Doms}(l)\}$ ;  
6     if  $\Delta \in \text{Doms}(C)$  and  $C' \neq \emptyset$  then  
7        $l \leftarrow$  an arbitrary literal in  $C'$ ;  
8        $\phi' \leftarrow \phi' \cup \{C' \cup \{\neg l\}\}$ ;  
9 else  
10   $c \leftarrow$  a new constant symbol;  
11  foreach clause  $C \in \phi$  do  
12     $C' \leftarrow C$ ;  
13    foreach  $v \in \text{Vars}(C)$  with  $\text{Dom}(v) = \Delta$  do  
14       $C' \leftarrow C'[c/v]$ ;  
15     $\phi' \leftarrow \phi' \cup \{C'\}$ ;
```

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

$$\begin{aligned} \forall y, z \in \Gamma : Q(y, z) \vee \neg Q(y, z) \\ \forall y, z \in \Gamma' : Q(y, z). \end{aligned}$$

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

5.2 The Solution

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.

6 Generating C++ Code

The target is to generate C++ code that can evaluate numerical values of the model counts based on the equations generated by 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 i th place if the i th argument is variable and the i th 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 $\sum_{x=a}^b \text{expr}$ to

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

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

7 Experiments

8 Conclusion

References