

Towards Practical First-Order Model Counting

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 C^2
- Can I prove that Crane's domain recursion can always simulate ForLift'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- Sections 4 and 5 might also be outdated: there is probably no need to search for base cases given the new version of domain recursion.

Proof of correctness.

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.

- Section 5
- Section 6

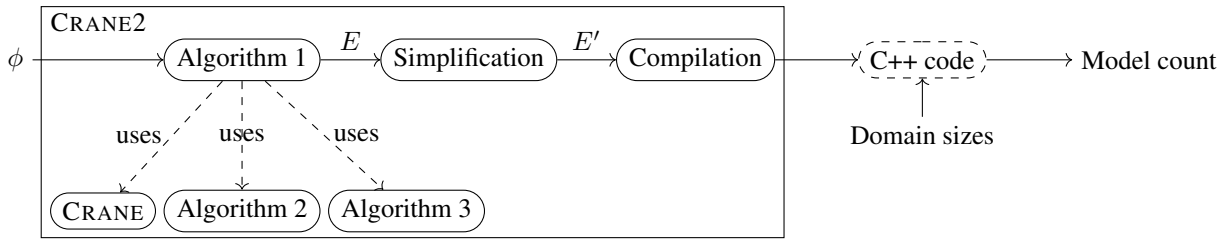


Figure 1: Using CRANE2 to compute the model count of formula ϕ . The formula is compiled into a set of equations E , which are then algebraically simplified and compiled to a C++ program. This program can then be run with different command line arguments to compute the model count of ϕ for various domain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

- Converting the recursive equations into a C++ program, which can then be compiled and executed to obtain numerical values (see Section 7).
- 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 5 and 7 deal with algebraic constructs whereas Section 6 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 .

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

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

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? Colon after every quantifier?
- (Similar to CNF, maybe QBF). Incomplete. Need (W)FOMC semantics. What is an interpretation? What is a model?

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 $\text{Preds}(C)$ 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 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.

3.1 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: we merge $P(c)$ and $\forall X \in \Delta. X \neq c \Rightarrow P(X)$ into $\forall X \in \Delta. P(X)$. NOTE: the clauses may have other constraints, variables, etc. I could describe this more formally.

3.2 Stage 2: Adding Smoothing Nodes

Ref: nothing.

Constraint removal: nothing.

Domain recursion: Smoothing for domain recursion now works the same way as smoothing for disjunction: both sides must have the same atoms EXCEPT the ‘base case’ side need not include unit clauses that mention the constant introduced by this domain recursion node or its domain (since this branch assumes that this domain is empty). NOTE: As a consequence, almost all of Section 6 is now outdated because smoothing makes the “tautology insertion” unnecessary.

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

See Algorithm 1

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: 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  $f(\mathbf{x} - \mathbf{c}) \in \text{expr}$  do
4     foreach  $c_i \in \mathbf{c}$  do
5       for  $n \leftarrow 0$  to  $c_i - 1$  do
6          $B \leftarrow B \cup \{f(\mathbf{x})[x_i/n]\}$ ;
```

- 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

5 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 $f(0, x_1, x_2, \dots), f(1, x_1, x_2, \dots), \dots, f(m_0, x_1, x_2, \dots)$ 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.

- 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: $\sum_{x=0}^{x_1} \text{expr} \cdot [a \leq x < b]$ 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 $\sum_{x=0}^{x_1} \binom{x_1}{x} f(x_1 - x) \cdot [0 \leq$

192 $x < 2]$ by $\binom{x_1}{0}f(x_1) + \binom{x_1}{1}f(x_1 - 1)$.
 193 Now, we find a domain that has only terms of the form $x - 1$
 194 appearing on the RHS of the dependencies. The base cases
 195 are then calculated by setting this domain size to zero. For
 196 the above example, n is the selected domain, and not m since
 197 there are $m - 2$ and $m - 3$ terms appearing in the arguments.
 198 The algorithm is described as Algorithm 2.

add an example where Algorithm 2 runs on either

$$\begin{aligned} f_0(m, n) &= f_1(m - 1, n) + f_2(m, n - 1) \\ f_1(m, n) &= f_1(m - 1, n - 1) \times f_2(m - 2, n - 1) \\ f_2(m, n) &= 2 \times f_1(m - 3, n - 1) \end{aligned}$$

or

$$\begin{aligned} f(m, n) &= g(m - 1, n) + f(m - 2, n - 1) \\ g(m, n) &= f(m - 1, n - 2) + g(m - 1, n - 1). \end{aligned}$$

- line 5: the limit is 2 for the arg $(x - 3)$.
- line 6 for $f(x) = f(y, z)$, $x_i = y$, and $n = 0$, add $f(y, z)[y/0] = f(0, z)$ to B .

I changed the algorithm. Update the description.

Theorem 1. 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.

6 Propagating Domain Size Assumptions

6.1 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

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

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     $\phi' \leftarrow \phi' \cup \{C'\}$ ;
```

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

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

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

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

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

6.2 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.

7 Generating C++ Code

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

253 The target is to generate C++ code that can evaluate numer-
 254 ical values of the model counts based on the equations gen-
 255 erated by CRANE. We achieve this by parsing the equations
 256 generated by CRANE, simplifying them, and then generating
 257 C++ code. This approach can be done in linear time in the
 258 length of the formula using the Shunting Yard Algorithm.

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

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

266 Next, we create a function definition for the LHS of each
 267 equation in E , including all functions and base cases. The
 268 signatures of these functions is decided as follows. A func-
 269 tion call containing only variable arguments is named as the
 270 function itself, and ones with constants in their arguments are
 271 suffixed with a string that contains ' x ' at the i th place if the
 272 i th argument is variable and the i th argument if that argument
 273 is a constant. For example, $f(x_1, x_2, x_3)$ is declared as `int`
 274 `f(int x1, int x2, int x3);` and $f(1, x_2, x_3)$ is de-
 275 clared as `int f_1xx(int x2, int x3);` (the constant
 276 arguments are removed from the signature).

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

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

286 Second, if the element is absent, then we decide if the argu-
 287 ments corresponding to e or one of the functions correspond-
 288 ing to the base cases, based on the value of the arguments.
 289 If it corresponds to the base cases, then we directly call the
 290 base case function and return its value. Else, we evaluate the
 291 value using the RHS, store the evaluated value in the cache
 292 and return the evaluated value. Note that in this step, we only
 293 call the base case function with one more constant argument
 294 than `func`. For example, $f_0(x, y)$ would call $f_{0_0x}(y)$
 295 if $x = 0$ and $f_{0_x0}(x)$ if $y = 0$.

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

```
297 ([y, z, ...]) {
298     int sum = 0;
299     for(int x = a; x <= b; x++)
300         sum += expr;
301     return sum;
302 }
```

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

304 8 Experiments

305 **Remarks on comparing the model counts of the formulas**
 306 **with their corresponding OEIS sequences.** Disregarding
 307 differences where the OEIS sequence starts with one or two

308 numbers that are not included in model counts (which is some-
 309 what justified since we start with domains of size one rather
 310 than zero), two differences remain:

- on A62971, the OEIS sequence starts as 1, 2, 16, \dots , but
 311 the model counts start as 1, 16, \dots (with the rest exactly
 312 the same); 313
- similarly, on A62971, the OEIS sequence starts as
 314 32, 290, \dots , but the model counts start as 2, 32, 290, \dots
 315 (with the rest exactly the same). 316

317 9 Conclusion