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 $\mathbf{C}^2$

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.

- 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, we might have a function g that depends on m (i.e., domain $a^\top$), which is called as $g(n)$ (i.e., on domain a). When smoothing that function call, we want to translate all mentions of $a^\top$ to a.

- 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).

- Smoothing for domain recursion now works the same way as smoothing for disjunction: both sides must have the same atoms. Almost all of Section 5 is now outdated because smoothing makes the "tautology insertion" unnecessary.

- Sections 3 and 4 might also be outdated: there is probably no need to search for base cases given the new version of domain recursion.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
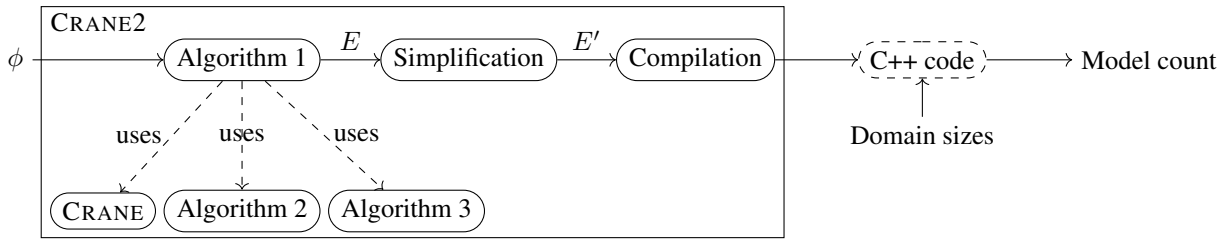
Figure 1: Using CRANE2 to compute the model count of formula ϕ. The formula is compiled into a set of equations E, which are then algebraically simplified and compiled to a C++ program. This program can then be run with different command line arguments to compute the model count of ϕ for various domain sizes.

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.

> references to my previous paper and some of the FOR-CLIFT 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.

> - 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* $f(x_1 - c_1, \ldots, x_n - c_n)$ *(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* $f(x_1, \ldots, x_n)$ *(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 $\forall x_1 \in \Delta_1 \forall x_2 \in \Delta_2 \ldots \forall x_n \in \Delta_n \phi(x_1, x_2, \ldots, x_n)$, where ϕ is a disjunction of literals that only contain variables $x_1, \ldots, x_n$ (and any constants). A *literal* is either an atom or its negation. An *atom* is of the form $P(t_1, \ldots, t_m)$, where P is a predicate, and $t_1, \ldots, t_m$ 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.

Algorithm 1: Finding the definitions of the base cases

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

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

> **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** $\quad$ **foreach** *function call* $f(\mathbf{x} - \mathbf{c}) \in \texttt{expr}$ **do**
> **4** $\quad\quad$ **foreach** $c_i \in \mathbf{c}$ **do**
> **5** $\quad\quad\quad$ **for** $n \leftarrow 0$ **to** $c_i - 1$ **do**
> **6** $\quad\quad\quad\quad$ $B \leftarrow B \cup \{\, f(\mathbf{x})[x_i/n] \,\}$;

134 For any (bound) variable v, let $\mathrm{Dom}(v)$ denote the do-
135 main over which v is quantified. For a literal l, $\mathrm{Doms}(l) :=$
136 $\{\, \mathrm{Dom}(v) \mid v \in \mathrm{Vars}(l) \,\}$. Similarly, for any clause C, let
137 $\mathrm{Doms}(C) = \{\, \mathrm{Dom}(v) \mid v \in \mathrm{Vars}(C) \,\}$.

138 ## 3 The Main Algorithm: Finding the
139 ## Definitions of the Base Cases

140 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

141

142 ## 4 Identifying a Sufficient Set of Base Cases

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

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

153

154 The following steps were followed while finding the base
155 cases:

156 First, expand the summations in each equation. Here we
157 expand the summations of the form: $\sum_{x=0}^{x_1} \texttt{expr} \cdot [a \le x < b]$
158 or similar inequalities where x is bounded by constants and a
159 and b are constants, by substituting the value of x from a to
160 $b - 1$. For example, we replace $\sum_{x=0}^{x_1} \binom{x_1}{x} f(x_1 - x) \cdot [0 \le$
161 $x < 2]$ by $\binom{x_1}{0} f(x_1) + \binom{x_1}{1} f(x_1 - 1)$.

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

167 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
>
> $$f(m, n) = g(m - 1, n) + f(m - 2, n - 1)$$
> $$g(m, n) = f(m - 1, n - 2) + g(m - 1, n - 1).$$

168

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

170 - line 6 for $f(\mathbf{x}) = f(y, z)$, $x_i = y$, and $n = 0$, add
171 $f(y, z)[y/0] = f(0, z)$ to B.

> I changed the algorithm. Update the description.

172

173 **Theorem 1.** *Under the following assumptions, Algorithm 2 is*
174 *guaranteed to return a sufficient set of base cases:*

175 - *there is no mutual recursion (i.e., cyclic dependencies*
176   *between functions);*

177 - *each function f has exactly one equation with f on the*
178   *LHS;*

179 - *in the recursive definition of function $f(x_1, \ldots, x_n)$, the*
180   *i-th argument of each call to f on the RHS is of the form*
181   *$x_i - c_i$, where:*

182   – *$c_i \ge 0$ for all i, and*
183   – *$c_i > 0$ for at least one i.*

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

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

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** $\Delta \notin \mathrm{Doms}(C)$ **then** $\phi' \leftarrow \phi' \cup \{C\}$;
5         $C' \leftarrow \{l \in C \mid \Delta \notin \mathrm{Doms}(l)\}$;
6   **if** $\Delta \in \mathrm{Doms}(C)$ **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** $v \in \mathrm{Vars}(C)$ *with* $\mathrm{Dom}(v) = \Delta$ **do**
14            $C' \leftarrow C'[c/v]$;
15         $\phi' \leftarrow \phi' \cup \{C'\}$;

---

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

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

$$\forall y, z \in \Gamma' : Q(y, z) \tag{2}$$

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 $z \in \Gamma \setminus \Gamma'$.

5.2   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.

6   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

```
266  ([y,z,...]() {
267      int sum = 0;
268      for(int x = a; x <= b; x++)
269          sum += expr;
270      return sum;
271  }) ()
```

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

7 Experiments

Remarks on comparing the model counts of the formulas with their corresponding OEIS sequences. Disregarding differences where the OEIS sequence starts with one or two numbers that are not included in model counts (which is somewhat justified since we start with domains of size one rather than zero), two differences remain:

- on A62971, the OEIS sequence starts as $1, 2, 16, \ldots$, but the model counts start as $1, 16, \ldots$ (with the rest exactly the same);

- similarly, on A62971, the OEIS sequence starts as $32, 290, \ldots$, but the model counts start as $2, 32, 290, \ldots$ (with the rest exactly the same).

8 Conclusion