Towards Practical First-Order Model Counting

Abstract

1 Introduction

Contributions.

- 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.

2 Snippets

- 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 objects: `clause_func_map` (which stores the mapping from the function names to the formulae, whose model count they represent) and `var_domain_map` (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.
- Finding 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. $\operatorname{argmin}(\max(c_1, c_2, \dots c_k))$.

2.1 Bar

The previous method of base case evaluation on setting a domain to size zero or one had the following error. In case a domain size was set to zero, it assumed that those predicates which were deleted from the clauses could take any truth value over the entire domain and the rest were fully covered by the remaining clauses.

For example, consider the formula

$$\begin{aligned} \forall x \in \Delta \forall y, z \in \Gamma &: P(x) \lor Q(y, z) \\ \forall y \in \Gamma^\top &: Q(y, z). \end{aligned} \tag{1}$$

In this case, if we set $|\Delta|$ to zero, the transformed formula would be

$$\forall y, z \in \Gamma^\top : Q(y, z),$$

and the set of removed predicates is empty, and hence the model count returned would be 1. However, the actual model count should be $2^{|\Gamma|^2 - |\Gamma^\top|^2}$.

We solve this problem 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 the above mentioned formula to

$$\begin{aligned} \forall y, z \in \Gamma &: Q(y, z) \lor \neg Q(y, z) \\ \forall y, z \in \Gamma^\top &: Q(y, z) \end{aligned}$$

The model count returned by this will also consider the truth value of Q over $y \notin \Gamma^\top$ or $z \notin \Gamma^\top$.

3 Preliminaries

TODO: Introduce terms: left-hand side (LHS), right-hand side (RHS).

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 $f(x, y)$.*

Remark. *The LHS of an equation is always a signature.*

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.

Algorithm 1: Identifying a set of sufficient base cases

```
Input: set $D$ of dependencies
Output: set $B$ of base cases
1 $B \leftarrow \emptyset$;
2 foreach $(f(\mathbf{x}), g(\mathbf{y} - \mathbf{c})) \in D$ do
3     foreach $c_i \in \mathbf{c}$ do
4         for $n \leftarrow 0$ to $c_i - 1$ do
5             $B \leftarrow B \cup \{ f(\mathbf{x})[x_i/n] \}$;
6             if $f \neq g$ then $B \leftarrow B \cup \{ g(\mathbf{y})[y_i/n] \}$;
```

4 Identifying a Sufficient Set of Base Cases

The following steps were followed while finding the base cases:

1. Expand the summations in each equation. Here we expand the summations of the form: $\sum_{x=0}^{x_1} <$ something $>$ $\cdot[a \le x < b]$, 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 $\sum_{x=0}^{x_1} \binom{x_1}{x} f(x_1 - x) \cdot [0 \le x < 2]$ by $\binom{x_1}{0} f(x_1) + \binom{x_1}{1} f(x_1 - 1)$.
2. Next, we find the dependencies of those functions that appear on the RHS of any equation. Consider, for example the following set of equations:
$$f_0(m,n) = f_1(m-1,n) + f_2(m,n-1)$$
$$f_1(m,n) = f_1(m-1,n-1) \times f_2(m-2,n-1)$$
$$f_2(m,n) = 2 \times f_1(m-3,n-1)$$
In this case, the dependencies computed are
$$f_1(m,n) \mapsto \{ f_1(m-1,n-1), f_2(m,n-1) \}$$
$$f_2(m,n) \mapsto \{ f_1(m-3,n-1) \}$$
3. 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.

Limitations of the current implementation. 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 4: the limit is 2 for the arg $(x-3)$.
- line 5 for $f(\mathbf{x}) = f(x_1, x_2)$, arg $= x$, and $n = 0$, add $f(0, x_2)$ to B.
- note that on line 6 it is the signature and not the original function call.

The algorithm is described as Algorithm 1. Here, a dependency is a pair (a, b), where a is the signature on the LHS of each equation and b is each function call on the RHS of the equation. For example, for the equations
$$f(m,n) = g(m-1,n) + f(m-2,n-1)$$
$$g(m,n) = f(m-1,n-2) + g(m-1,n-1)$$
the dependencies are
$$(f(m,n), g(m-1,n)), (f(m,n), f(m-2,n-1)),$$
$$(g(m,n), f(m-1,n-2), g(m-1,n-1)).$$

Algorithm 2: Transforming Formulas Based on Domain Sizes

```
Input: formula $\phi$, domain $\Delta$, target domain size
       $n \in \{ 0, 1 \}$, domain size function $| \cdot |$
Output: formula $\phi'$
1  $P^- \leftarrow \emptyset$; $P^+ \leftarrow \emptyset$; $\phi' \leftarrow \emptyset$;
2  if $|\Delta| = 0$ then
3      foreach clause $C \in \phi$ do
4          if $\Delta \in \mathrm{Doms}(C)$ then
5              $P^- \leftarrow P^- \cup \mathrm{Preds}(C)$;
6          else
7              $P^+ \leftarrow P^+ \cup \mathrm{Preds}(C)$;
8              $\phi' \leftarrow \phi' \cup \{ C \}$;
9      $P^- \leftarrow P^- \setminus P^+$;
10 else if $|\Delta| = 1$ then
11     $c \leftarrow$ a new constant symbol;
12     foreach clause $C \in \phi$ do
13         $C' \leftarrow C$;
14         foreach $v \in \mathrm{Vars}(C)$ with $\mathrm{Dom}(v) = \Delta$ do
15             $C' \leftarrow C'[c/v]$;
16         $\phi' \leftarrow \phi' \cup \{ C' \}$;
```

5 Algorithm to Transform CNF Based on Domain Sizes

- Formulas are in CNF. Formulas are (multi) sets of clauses.
- P^- — removed predicates
- P^+ — retained predicates
- ϕ' — transformed version of ϕ
- C' — transformed version of clause C
- For any clause C, let $\mathrm{Doms}(C)$, $\mathrm{Preds}(C)$, and $\mathrm{Vars}(C)$ denote respectively the set of domains, predicates, and variables in C. For any variable v, let $\mathrm{Dom}(v)$ denote its domain. Note that, for any clause C, we have that $\mathrm{Doms}(C) = \{ \mathrm{Dom}(v) \mid v \in \mathrm{Vars}(C) \}$. Similarly, for any predicate P, let $\mathrm{Doms}(P)$ denote the set of domains associated with P.

We use Algorithm 2 to find the transformed formula corresponding to each base case obtained using Algorithm 1 and call CRANE on the formula to obtain the required base cases.

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. There are two ways to do the same.

1. Generate C++ code by traversing the FCG, similar to what is done in `OutputVisitor.scala`.

2. Parse the equations generated by CRANE after simplifying in wolfram and then generate C++ code.

The problem with the first approach is that, while generating base cases, the subsequent calls to CRANE do not necessarily have the same meanings for the function arguments and the functions. For example, if $f_1(x_0, x_1, x_2)$ represents the model count of a constrained formula ϕ, where f_1 is an auxiliary formula and $x_0 = |A|$, $x_1 = |B|$, $x_2 = |C|$, and A, B, C are domains and we want to evaluate $f_1(0, x_1, x_2)$ (i.e., set $|A| = 0$), then CRANE may return the required model count as $f_0(x_0, x_1)$, where $x_0 = |B|$ and $x_1 = |C|$. We will then need to translate this to $f_1(0, x_1, x_2)$. Also, the second approach can be done in linear time in the length of the formula using the Shunting Yard Algorithm. Hence, we stick to the second approach.

6.1 Our Approach

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.

1. 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.

2. 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 that `func`. For example, `f0(x, y)` would call `f0_0x(y)` if `x == 0` and `f0_x0(x)` if `y == 0`.

3. In order to translate the RHS, we convert $\sum_{x=a}^{b}$ `exp` to

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

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

j

References