Towards Practical First-Order Model Counting

Abstract

1 Introduction

2 Methodology

- Added support for infinite precision integers using the `gmp` library to the numerical evaluation part. Now, all functions take arguments (domain sizes) as unsigned integers and return `mpz_class` objects (defined in the `gmpxx.h` header).
- Added a CLI option `--sizes` to specify the domain sizes on the command line.
- Converting the recursive equations into a `C++` program, which can then be compiled and executed to obtain numerical values.
- The translation of a set of equations(E) into `C++` works as follows.
  1. First, we create a cache for each function in E. For a function with name `func`, this cache is called `func_cache`. This is implemented as a multi-dimensional vector containing objects of `class cahce_elem` defined as shown in the example code. The default initialization of this object is to -1 which is useful for recognizing unevaluated cases.
  2. Next, we create a function definition for each lhs 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, $f(x1, x2, x3)$ is declared as `int f(int x1, int x2, int x3);` and $f(1, x2, x3)$ is declared as `int f_1xx(int x2, int x3);` (the constant arguments are removed from the signature).
  3. 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-
     (a) 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.
     (b) 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`.
     (c) In order to translate the rhs, we do the following changes-
     - `a^b` is converted to `power(a,b)`.
     - `Sum[exp, x, a, b]` = ($\sum_{x=a}^{b} exp$) is converted to -

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

     where y, z, ... are the free variables present in `exp`.
- Implemented the evaluation of basecases. The evaluation is done by simplifying the clauses and then using CRANE to find the basecases. First, in the `SimplifyUsingWolfram` class, while traversing the graph to find the equations, we store 2 `Map` 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 using `WeightedCNF.SimplifyInWolfram`. After that, we change the function names and variable to make it consistent with the previous domain to variable mapping, and append these basecases to the set of equations.

- Ideas for finding which base cases are needed. The goal is to find the 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. $argmin(\max(c_1, c_2, \ldots c_k))$.
- If a domain is set to 0, the model count can be found by simplifying the universal and existential quantifiers over that domain, and then using crane. However, if a domain is set to contain some non-empty number of elements, we can apply GDR over that domain and proceed with crane, substituting the known values of preveiously calculated model counts in the process.
- Implemented a function to simplify functions (recursive and non-recursive) using the Wolfram Engine. Modified the wolfram simplification functions to incorporate binomial and indicator functions.

2.1 Foo

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 for CRANE do not necessarily have the same meanings for the function arguments and the functions. For example, if `f1[x0, x1, x2]` represents the model count of a constrained formula F, where f1 is an auxilliary formula and $x0 = |A|, x1 = |B|, x2 = |C|$, ($A, B, C$ are domains) and we want to evaluate `f1[0, x1, x2]`($|A| = 0$), then CRANE may return the required model count as `f0[x0, x1]`, where $x0 = |B|, x1 = |C|$. We'll need to translate this to `f1[0,x1,x2]`, which has already been implemented in `Basecases.scala`. Also, the second approach can be done in linear time in the length of the formula using the Shunting Yard Algorithm. Hence, we are implementing the second approach (have already implemented the parser).

2.2 Bar

The previous method of base case evaluation on setting a domain to size 0 or 1 had the following error - in case a domain size was set to 0, 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 constrained CNF formula -

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

$$\forall y \in \Gamma^{\top} : Q(y, z) \quad (2)$$

In this case, if we set $|\Delta|$ to 0, then based on the previous idea, the new set of clauses would be

$$\forall y, z \in \Gamma^{\top} : Q(y, z) \quad (3)$$

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

There are 2 ideas which we could think of to solve this -

1. *(Currently, we have implemented this)* Convert the clauses having universal quantification over the null domain to tautology clauses, hence retaining all the predicates which don't have an argument belonging to the null domain. For example, we would convert the above mentioned CNF formula to

   $$\forall y, z \in \Gamma : Q(y, z) \vee \neg Q(y, z) \quad (4)$$

   $$\forall y, z \in \Gamma^{\top} : Q(y, z) \quad (5)$$

   The model count returned by this will also consider the truth value of Q over $y \notin \Gamma^{\top}$ or $z \notin \Gamma^{\top}$.
2. We find the domain size over which the predicate has not been defined by the clauses and multiply the required factor. For example, in the above example, the multiplier would be $2^{|\Gamma|^2 - |\Gamma^{\top}|^2}$ and the simplified set of clauses would be

   $$\forall y, z \in \Gamma^{\top} : Q(y, z) \quad (6)$$

   which has a model count of 1. Hence, the obtained value of model count would be $2^{|\Gamma|^2 - |\Gamma^{\top}|^2} \times 1$.

3 Algorithms

3.1 Algorithm to Find a Sufficient Set of Base Cases

The pseudo-code of the algorithm is given as Algorithm 1. Here, `dependencies` maps the function call on the lhs of each equation to the set of function calls 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

$$f(m, n) \mapsto \{g(m-1, n), f(m-2, n-1)\}$$

$$g(m, n) \mapsto \{f(m-1, n-2), g(m-1, n-1)\}$$

3.2 Algorithm to Transform CNF Based on Domain Sizes

The pseudo code is given as Algorithm 2. We use Algorithm 2 to find the transformed CNF corresponding to each base case obtained using Algorithm 1 and call CRANE on the CNF to obtain the required base cases.

3.3 Baz

The following steps were followed while finding the basecases-

1. Expand the summations in each equation - Here we expand the summations of the form: $\sum_{x=0}^{x1} < \text{something} > \cdot[a \leq x < b]$, 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 $\binom{x_1}{0} f(x_1) + \binom{x_1}{1} f(x_1 - 1)$.
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-

$$f0(m,n) = f1(m-1,n) + f2(m,n-1) \tag{7}$$
$$f1(m,n) = f1(m-1,n-1) \times f2(m-2,n-1) \tag{8}$$
$$f2(m,n) = 2 \times f1(m-3,n-1) \tag{9}$$

   In this case, the dependencies computed are

$$f1(m,n) \to f1(m-1,n-1), f2(m,n-1)$$
$$f2(m,n) \to f1(m-3,n-1)$$

3. Now, based on idea II of the previous report, 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 0. 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 based on idea II of the previous report by finding the basecases upto $\max(c1, c2, ...) - 1$. However, we are yet to implement code for finding the basecases for the non-null domain case.

References

Algorithm 1: Algorithm For Sufficient Base Cases

```
Input: dependencies
Output: base_cases
base_cases ← Set()
foreach dependency ∈ dependencies do
    foreach elem ∈ dependency.value do
        if dependency.key.func_name = elem.func_name
         then
            foreach arg ∈ elem.args do
                // eg : lim is 2 for the
                   arg (x-3)
                lim ← arg.const_subtracted - 1
                for l ∈ 0 to lim do
                    // eg : for
                       dependency.key =
                       f(x,y), arg = x and
                       l = 0, add f(0, y)
                       to base_cases
                    base_cases += depen-
                     dency.key.substitute_arg(arg_var, l)
        else
            foreach arg ∈ elem.args do
                // eg : lim is 2 for the
                   arg (x-3)
                lim ← arg.const_subtracted - 1
                for l ∈ 0 to lim do
                    // eg : for
                       dependency.key =
                       f(x,y), arg = x and
                       l = 0, add f(0, y)
                       to base_cases
                    base_cases += depen-
                     dency.key.substitute_arg(arg, l)
                    // signature for f(x-1,
                       y-2) is f(x,y)
                    base_cases +=
                     elem.signature.substitute_arg(arg,
                     l)
```

Algorithm 2: Algorithm For Transforming CNF Based on Domain Sizes

Input: cnf, dom, dom_size
Output: transformed_cnf, multiplier

```
removed_predicates ← Set() retained_predicates ← Set() if dom_size = 0 then
    foreach clause ∈ cnf do
        if dom ∈ clause.domains then
            removed_predicates += clause.predicates
        else
            retained_predicates += clause.predicates
            transformed_cnf += clause
    removed_predicates = removed_predicates \ retained_predicates foreach pred ∈ removed_predicates do
        multiplier *= 2^(pred.domain_sizes.join("*"))
else if dom_size = 1 then
    const ← newConst() foreach clause ∈ cnf do
        transformed_clause ← clause foreach v ∈ clause.vars — v.domain = dom do
            transformed_cnf ← transformed_clause.substituteVar(v, const)
        transformed_cnf += transformed_clause
```