Recursive Solutions to First-Order Model Counting

21st February 2022

1 Definitions

Things I might need to explain.

- atom, (positive/negative) literal, constant, predicate, variable, literal variable, clause, unit clause
- Vars, $\mathrm{Vars}(c) = \mathrm{Vars}(L) \cup \mathrm{Vars}(C)$
- Doms on both formulas and clauses. $\mathrm{Doms}(c) = \mathrm{Im}\,\delta_c$, and $\mathrm{Doms}(\phi) = \bigcup_{c \in \phi} \mathrm{Doms}(c)$.
- maybe: notation for partial function, notation for powerset, notation for disjoint union
- WMC, w, $\overline{\mathrm{w}}$, Im
- two parts: compilation and inference.
- introduce and use arrows for bijections, injections, set inclusions, etc.
- $\emptyset$ for an empty partial map.
- notation for variable substitution for literal and constraint sets: $S[x/V]$, where S is the set, V is a set of variable, and x is the new constant or variable. Note that for constraint sets this could mean that a variable and a constant need to switch places.
- During inference, there is a domain size map $\sigma \colon \mathcal{D} \to \mathbb{N}_0$.
- mention which rules are in Γ and which ones are in Δ (and why tryCache has to be in Δ).
- ForcLift
- In a way, we're dividing the idea of domain recursion between the IDR and the Ref nodes, thus also generalising it. [This is good context to refer to Fig. 1.]

TODO

- should I always say 'compilation rule' instead of 'operator'?
- maybe π is global enough to have a more unique name
- capitalise variable and domain names.
- maybe introduce/use notation for vertices, edges of a graph
- maybe put each function into its own algorithm environment, using the caption for the function name
- replace the Compile operator with my new definition of a rule. Instead of preconditions, have the rule return a set for all applicable domains.

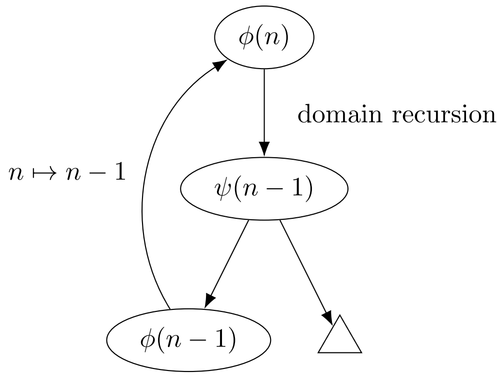

Figure 1: An illustration of the main idea. TODO: refer to this in the introduction.

Most of the definitions here are adaptations/formalisations of [2] and the corresponding code.

Definition 1. A *domain* is a symbol for a finite set.[1] Let $\mathcal{D}$ be the set of all domains and $\mathcal{C} \subset \mathcal{D}$ be the subset of domains introduced as a consequence of constraint removal. Note that both sets (can) expand during the compilation phase.

Let $\pi\colon \mathcal{D} \nrightarrow \mathcal{D}$ be a partial endomorphism on $\mathcal{D}$ that denotes the *parent* relation, i.e., if $\pi(d) = e$ for some $d, e \in \mathcal{D}$, then we call e the parent (domain) of d, and e a child of d. Intuitively, π arranges all domains into a forest—thus, we often use graph theoretical terminology to describe properties of and relationships between domains.

Definition 2. An *(inequality) constraint* is a pair (a, b), where a is a variable, and b is either a variable or a constant.

Definition 3. A *clause* is a triple $c = (L, C, \delta_c)$, where L is the set of literals, C is a set of inequality constraints, and $\delta_c\colon \mathrm{Vars}(c) \to \mathcal{D}$ is a function that maps all variables in c to their domains such that if $(x, y) \in C$ for some variables x and y, then $\delta_c(x) = \delta_c(y)$. Equality of clauses is defined in the usual way (i.e., all variables, domains, etc. must match). TODO: we will always use this subscript notation for the δ's.

A *formula* is a set of clauses.

We use hash codes to efficiently check whether a recursive relationship between two formulas is plausible. (It is plausible if the formulas are equal up to variables and domains.) The hash code of a clause $c = (L, C, \delta_c)$ combines the hash codes of the sets of constants and predicates in c, the numbers of positive and negative literals, the number of inequality constraints $|C|$, and the number of variables $|\mathrm{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted $\#\phi$.

Definition 4. Let $\mathrm{gr}(\cdot; \sigma)$ be the function (parameterised by the domain size function σ) that takes a clause $c = (L, C, \delta)$ and returns the number of ways the variables in c can be replaced by elements of their respective domains in a way that satisfies the inequality constraints.[2] Formally, for each variable $v \in \mathrm{Vars}(c)$, let $C_v = \{w \mid (u, w) \in C \setminus \mathrm{Vars}(c)^2,\ u \neq v\}$ be the set of (explicitly named) constants that v is permitted to be equal to. Then

$$\mathrm{gr}(c; \sigma) \coloneqq \left| \left\{ (e_v)_{v \in \mathrm{Vars}(c)} \in \prod_{v \in \mathrm{Vars}(c)} C_v \sqcup [\sigma(\delta(v)) - |C_v|] \;\middle|\; e_u \neq e_w \text{ for all } (u, w) \in C \cap \mathrm{Vars}(c)^2 \right\} \right|$$

for any clause c. (Here, $[n] \coloneqq \{ 1, 2, \dots, n \}$ for any non-negative integer n.)

TODO: I'm already using the square brackets to denote lists.

TODO: how does the algorithm prevent the number in $[\cdot]$ from being negative?

[1]In the context of functions, the domain of a function f retains its usual meaning and is denoted $\mathrm{dom}(f)$.

[2]Note that the literals of the clause have no effect on gr.

Notation for lists. We let [] and $[x]$ denote an empty list and a list with one element x, respectively. We write $\in$ for (in-order) enumeration, $+\!\!+$ for concatenation, and $|\cdot|$ for the length of a list. Let $h : t$ denote a list with first element (a.k.a. head) h and remaining list (a.k.a. tail) t. We also use list comprehensions written equivalently to set comprehensions. For example, let $L \coloneqq [1]$ and $M \coloneqq [2]$ be two lists. Then $M = [2x \mid x \in L]$, $L +\!\!+ M = 1 : [2]$, and $|M| = 1$.

2 Search/Compilation

2.1 From Circuits to Labelled Graphs

A *first-order deterministic decomposable negation normal form computational graph* (FCG) is a (weakly connected) directed graph with a a single source, vertex labels, and ordered outgoing edges.[3] We denote an FCG as $G = (V, s, N^+, \tau)$, where V is the set of vertices, and $s \in V$ is the unique source. Also, N^+ is the direct successor function that maps each vertex in V to a *list* that contains either other vertices in V or a special symbol $\star$. This symbol means that the target of the edge is yet to be determined.

Vertex labels consist of two parts: the *type* and the *parameters*. For the main definition, we leave the parameters implicit and let $\tau \colon V \to \mathcal{T}$ denote the vertex-labelling function that maps each vertex in V to its type in $\mathcal{T}$. Most of our list of types $\mathcal{T} \coloneqq \{ \bigcirc, \textcircled{1}, \textcircled{\top}, \textcircled{\bot}, \textcircled{\land}, \textcircled{\lor}, \mathrm{CR}, \mathrm{DR}, \mathrm{IE}, \textsc{Ref} \}$ is as described in previous work [1, 2] as well as the source code of ForcLift[4] but with one new type CR and two revamped types DR and Ref. For each vertex $v \in V$, the length of the list $N^+(v)$ (i.e., the out-degree of v) is determined by its type $\tau(v)$.

As in previous work [2], to describe individual vertices and small (sub)-FCGs, we also use notation of the form, e.g., $\textsc{Ref}_\rho(v)$. Here, the type of the vertex (e.g., Ref) is 'applied' to its direct successors (e.g., v) using either function or infix notation and provided with its parameter(s) (e.g., ρ) in the subscript. We say that 'G is an FCG for formula ϕ' if two conditions are satisfied. First, the image of N^+ contains no $\star$'s. Second, G encodes a function that maps the sizes of the domains in ϕ to the WMC of ϕ (more on this in Section 7).

TODO.

- provide a short explanation of the types (emphasising which ones are new/updated).
- have an example of a simple FCG

2.2 Everything Else

Definition 5. A *state* (of the search for an FCG for a given formula) is a tuple (G, C, L), where:

- G is an FCG (or `null`),
- C is a compilation cache that maps integers to sets of pairs (ϕ, v), where ϕ is a formula, and v is a vertex of G (which is used to identify opportunities for recursion),
- and L is a list of formulas (that are yet to be compiled). (Note that the order is crucial!)

Definition 6. A (compilation) *rule* is a function that takes a formula and returns a set of (G, L) pairs, where G is a (potentially `null`) FCG, and L is a list of formulas. TODO: add an example showing that it's usually an FCG with one vertex and a bunch of $\star$'s marking a fixed number of outgoing edges.

We assume that if there is a pair $(\texttt{null}, L)$ in the set returned by a rule, then $|L| = 1$, i.e., the rule transformed the formula without creating any vertices.

[3] Note that imposing an ordering on outgoing edges is just a limited version of edge labelling.
[4] `https://dtai.cs.kuleuven.be/drupal/wfomc`

Algorithm 1: The (main part of the) search algorithm

Input: a formula ϕ_0
Result: all found FCGs for ϕ_0 are in the set $\mathsf{solutions}$

$\mathsf{solutions} \leftarrow \emptyset$;
$C_0 \leftarrow \emptyset$;
$(G_0, C_0, L_0) \leftarrow$ `applyGreedyRules(`ϕ_0, C_0`)`;
if $L_0 = []$ **then** $\mathsf{solutions} \leftarrow \{\, G_0 \,\}$;
else
  $q \leftarrow$ an empty queue of states;
  q`.put(`(G_0, C_0, L_0)`)`;
  while not q`.empty()` **do**
    foreach *state* $(G, C, L) \in$ `applyAllRules(`q`.get())` **do**
      if $L = []$ **then** $\mathsf{solutions} \leftarrow \mathsf{solutions} \cup \{\, G \,\}$;
      else q`.put(`(G, C, L)`)`;

TODO: explain the 'tail' part of the algorithm, i.e., that the first formula is replaced by some vertices and some formulas. And explain why we don't want to have REF vertices in the cache.

Note: At the end, `mergeFcgs` will never return `null` because there is going to be at least one $\star$ in G and the function will find it.

3 Smoothing

[Insert motivation for smoothing from Section 3.4. of the ForcLift paper.] Originally, smoothing was (and still is) a two-step process. First, atoms that are still accounted for in the circuit are propagated upwards. Then, at vertices of certain types, missing atoms are detected and additional sinks are created to account for them. If left unchanged, the first step of this process would result in an infinite loop whenever a cycle is encountered. Algorithm 4 outlines how the first step can be adapted to an arbitrary directed graph.

4 Identifying Possibilities for Recursion

Definition 7 (Notation)**.** For any clause $c = (L, C, \delta_c)$, bijection $\beta\colon \mathrm{Vars}(c) \to V$ (for some set of variables V) and function $\gamma\colon \mathrm{Doms}(c) \to \mathcal{D}$, let $c[\beta, \gamma] = d$ be the clause with all occurrences of any variable $v \in \mathrm{Vars}(c)$ in L and C replaced with $\beta(v)$ (so $\mathrm{Vars}(d) = V$) and $\delta_d\colon V \to \mathcal{D}$ defined as $\delta_d \coloneqq \gamma \circ \delta_c \circ \beta^{-1}$. In other words, δ_d is the unique function that makes the following diagram commute:

$$\begin{tikzcd} \mathrm{Vars}(c) \arrow[r, tail, two heads, "\beta"] \arrow[d, "\delta_c"'] & V = \mathrm{Vars}(d) \arrow[d, dashed, "\exists! \delta_d"] \\ \mathrm{Doms}(c) \arrow[r, "\gamma"'] & \mathcal{D}. \end{tikzcd}$$

The function `traceAncestors` returns `null` if domain $d \in \mathcal{D}$ is not an ancestor of domain $e \in \mathcal{D}$. Otherwise, it returns `true` if the size of e is guaranteed to be strictly smaller than the size of d (i.e., there is domain created by the constraint removal rule on the path from d to e) and `false` if their sizes will be equal at some point during inference.

Notation: For partial functions $\alpha, \beta\colon A \nrightarrow B$ such that $\alpha|_{\mathrm{dom}(\alpha)\cap\mathrm{dom}(\beta)} = \beta|_{\mathrm{dom}(\alpha)\cap\mathrm{dom}(\beta)}$, we write $\alpha \cup \beta$ for the unique partial function such that $\alpha \cup \beta|_{\mathrm{dom}(\alpha)} = \alpha$, and $\alpha \cup \beta|_{\mathrm{dom}(\beta)} = \beta$. TODO: explain $\sqcup$ for both sets and functions.

Algorithm 2: Functions used in Algorithm 1 for applying compilation rules

```
Data: a set of greedy rules Γ
Data: a set of non-greedy rules Δ
Function applyGreedyRules(φ, C):
    foreach rule r ∈ Γ do
        if r(φ) ≠ ∅ then
            (G, L) ← any element of r(φ);
            if G = null then return applyGreedyRules(the only element of L, C);
            else
                (V, s, N⁺, τ) ← G;
                C ← updateCache(C, φ, G);
                return applyGreedyRulesToAllFormulas(G, C, L);
    return (null, C, [φ]);
Function applyGreedyRulesToAllFormulas((V, s, N⁺, τ), C, L):
    if L = [] then return ((V, s, N⁺, τ), C, L);
    N⁺(s) ← [];
    L′ ← [];
    foreach formula φ ∈ L do
        (G′, C, L″) ← applyGreedyRules(φ, C);
        L′ ← L′ ++ L″;
        if G′ = null then N⁺(s) ← N⁺(s) ++ [⋆];
        else
            (V′, s′, N′, τ′) ← G′;
            V ← V ⊔ V′;
            N⁺ ← N⁺ ⊔ N′;
            N⁺(s) ← N⁺(s) ++ [s′];
            τ ← τ ⊔ τ′;
    return ((V, s, N⁺, τ), C, L′);
Function applyAllRules(s):
    (G, C, L) ← s;
    φ : T ← L;
    (G′, C′, L′) ← a copy of s;
    newStates ← [];
    foreach rule r ∈ Δ do
        foreach (G″, L″) ∈ r(φ) do
            if G″ = null then newStates ← newStates ++ applyAllRules((G′, C′, L″));
            else
                (V, s, N⁺, τ) ← G″;
                C′ ← updateCache(C′, φ, G″);
                (G″, C′, L″) ← applyGreedyRulesToAllFormulas(G″, C′, L″);
                if G′ = null then newStates ← newStates ++ [(G″, C′, L″ ++ T)];
                else newStates ← newStates ++ [(mergeFcgs(G′, G″), C′, L″ ++ T)];
        (G′, C′, L′) ← a copy of s;
    return newStates;
```

Algorithm 3: Helper functions used by Algorithm 2

Function `updateCache`(C, ϕ, (V, s, N^+, τ))**:**
  if $\tau(s) = \textsc{Ref}$ **then return** C;
  if $\#\phi \notin \mathrm{dom}(C)$ **then return** $C \cup \{\, \#\phi \mapsto (\phi, s) \,\}$;
  if *there is no* $(\phi', v) \in C(\#\phi)$ *such that* $v = s$ **then** $C(\#\phi) \leftarrow (\phi, s) +\!\!+\, C(\#\phi)$;
  return C;

Function `mergeFcgs`($G = (V, s, N^+, \tau)$, $G' = (V', s', N', \tau')$, $r = s$)**:**
  if $\tau(r) = \textsc{Ref}$ **then return** `null`;
  foreach $t \in N^+(r)$ **do**
    if $t = \star$ **then**
      replace t with s' in $N^+(r)$;
      return $(V \sqcup V', s, N^+ \sqcup N', \tau \sqcup \tau')$;
    $G'' \leftarrow$ `mergeFcgs`(G, G', t);
    if $G'' \neq$ `null` **then return** G'';
  return `null`;

Algorithm 4: Propagating atoms for smoothing across the FCG in a way that avoids infinite loops

Input: FCG (V, s, N^+, τ)
Input: function ι that maps vertex types in $\mathcal{T}$ to sets of atoms
Input: functions $\{\, f_t \,\}_{t \in \mathcal{T}}$ that take a list of sets of atoms and return a set of atoms
Output: function S that maps vertices in V to sets of atoms

$S \leftarrow \{\, v \mapsto \iota(\tau(v)) \mid v \in V \,\}$;
changed $\leftarrow$ `true`;
while changed **do**
  changed $\leftarrow$ `false`;
  foreach *vertex* $v \in V$ **do**
    $S' \leftarrow f_{\tau(v)}([S(w) \mid w \in N^+(v)])$;
    if $S' \neq S(v)$ **then**
      changed $\leftarrow$ `true`;
      $S(v) \leftarrow S'$;

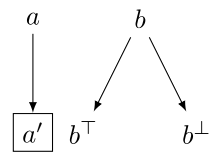

Figure 2: The forest of domains from Example 1. The directed edges correspond to the pairs of domains in π but in the opposite direction, e.g., $\pi(a') = a$. The domain enclosed in a rectangle is the only domain created by constraint removal, i.e., $\mathcal{C} = \{ a' \}$.

TODO

- introduce/describe Algorithm 5 and Algorithm 6 and describe the cache that's being used.
- explain why $\rho \cup \gamma$ is possible
- explain what the second return statement is about and why a third one is not necessary
- mention which one is the main function, what each function takes and returns
- explain the yield keyword
- in the example below: write down both formula using the ForcLift format
- the parent relation and ancestor tracing is no longer necessary

The algorithm could be improved in two ways:

- by constructing a domain map first and then using it to reduce the number of viable variable bijections.
- by similarly using the domain map ρ.

However, it is not clear that this would result in an overall performance improvement, since the number of variables in instances of interest never exceeds three and the identity bijection is typically the right one.

Diagramatically, `constructDomainMap` attempts to find $\gamma\colon \mathrm{Doms}(c) \to \mathrm{Doms}(d)$ such that the following diagram commutes (and returns `null` if such a function does not exist):

$$\begin{tikzcd}
\mathrm{Vars}(c) \arrow[r, tail, two heads, "\beta"] \arrow[d, "\delta_c"'] & \mathrm{Vars}(d) \arrow[d, "\delta_d"] \\
\mathrm{Doms}(c) \arrow[r, dashed, "\gamma"] \arrow[d, hook] & \mathrm{Doms}(d) \arrow[d, hook] \\
\mathcal{D} \arrow[r, "\rho"'] & \mathcal{D}.
\end{tikzcd}$$

Example 1. Let ϕ be the formula

$$\forall X \in a. \forall Y \in b. \forall Z \in b. Y \neq Z \implies \neg p(X, Y) \lor \neg p(X, Z) \tag{1}$$
$$\forall X \in a. \forall Y \in b. \forall Z \in a. X \neq Z \implies \neg p(X, Y) \lor \neg p(Z, Y). \tag{2}$$

and ψ be the formula

$$\forall X \in a'. \forall Y \in b^\perp. \forall Z \in b^\perp. Z \neq Y \implies \neg p(X, Y) \lor \neg p(X, Z) \tag{3}$$
$$\forall X \in a'. \forall Y \in b^\perp. \forall Z \in a'. X \neq Z \implies \neg p(X, Y) \lor \neg p(Z, Y) \tag{4}$$

The relevant domains and the definition of π is in Fig. 2. Since $\#\phi = \#\psi$, and the formulas are non-empty, the algorithm proceeds with the for-loops on Lines 6 to 8. Suppose c in the algorithm refers to Eq. (1), and d to Eq. (3). Since both clauses have three variables, in the worst case, function `generateMaps` would have $3! = 6$ bijections to check. Suppose the identity bijection is picked first. Then `constructDomainMap` is called with the following parameters:

- $V = \{ X, Y, Z \}$,
- $\delta_c = \{ X \mapsto a, Y \mapsto b, Z \mapsto b \}$,
- $\delta_d = \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto b^\perp \}$,
- $\beta = \{ X \mapsto X, Y \mapsto Y, Z \mapsto Z \}$,
- $\rho = \emptyset$.

Since $\delta_i(Y) = \delta_i(Z)$ for $i \in \{ c, d \}$, `constructDomainMap` returns $\gamma = \{ a \mapsto a', b \mapsto b^\perp \}$. Thus, `generateMaps` yields its first pair of maps (β, γ) to Line 8. Furthermore, the pair satisfies $c[\beta, \gamma] = d$.

Since $\pi(a') = a$, and $a' \in \mathcal{C}$, `traceAncestors(`a`, `a'`)` returns `true`, which sets $\mathsf{foundConstraintRemoval}'$ to `true` as well. When $e = b$, however, `traceAncestors(`b`, `$b^\perp$`)` returns `false` since $b^\perp$ is a descendant of b but not created by the constraint removal compilation rule. On Line 18, a recursive call to `identifyRecursion(`ϕ'`, `ψ'`, `γ`, true)` is made, where ϕ' and ψ' are new formulas with one clause each: Eq. (2) and Eq. (4), respectively.

Again we have two non-empty formulas with equal hash codes, so `generateMaps` is called with c set to Eq. (2), d set to Eq. (4), and $\rho = \{ a \mapsto a', b \mapsto b^\perp \}$. Suppose Line 22 picks the identity bijection first again. Then `constructDomainMap` is called with the following parameters:

- $V = \{ X, Y, Z \}$,
- $\delta_c = \{ X \mapsto a, Y \mapsto b, Z \mapsto a \}$,
- $\delta_d = \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto a' \}$,
- $\beta = \{ X \mapsto X, Y \mapsto Y, Z \mapsto Z \}$,
- $\rho = \{ a \mapsto a', b \mapsto b^\perp \}$.

Since β and ρ 'commute' (TODO: as in the diagram above), and there are no new domains in $\mathrm{Doms}(c)$ and $\mathrm{Doms}(d)$, γ exists and is equal to ρ. Again, the returned pair (β, γ) satisfies $c[\beta, \gamma] = d$. This γ passes the `traceAncestors` checks exactly the same way as the one before, and Line 18 calls `identifyRecursion(`$\emptyset$`, `$\emptyset$`, `ρ`, true)`, which immediately returns $\rho = \{ a \mapsto a', b \mapsto b^\perp \}$ as the final answer. This means that one can indeed use an FCG for $\mathrm{WMC}(\phi)$ to compute $\mathrm{WMC}(\psi)$ by replacing every mention of a with a' and every mention of b with $b^\perp$.

4.1 Evaluation

$\mathrm{WMC}(\textsc{Ref}_\rho(v); \sigma) = \mathrm{WMC}(v; \sigma')$ (n is the target vertex), where σ' is defined as

$$\sigma'(x) = \begin{cases} \sigma(\rho(x)) & \text{if } x \in \mathrm{dom}(\rho) \\ \sigma(x) & \text{otherwise} \end{cases}$$

for all $x \in \mathcal{D}$.

5 New Compilation Rules

Throughout this section, let ϕ be an arbitrary formula.

5.1 Constraint Removal

Preconditions. There is a domain $d \in \mathcal{D}$ and an element $e \in d$ such that:

- for each clause $c = (L, C, \delta_c) \in \phi$ and variable $v \in \mathrm{Vars}(c)$, either $\delta_c(v) \neq d$ or $(v, e) \in C$;
- e does not occur in any literal of any clause of ϕ.

Operator. First, we introduce a new domain (i.e., $\mathcal{D} \coloneqq \mathcal{D} \sqcup \{ d' \}$), add it to $\mathcal{C}$ (i.e., $\mathcal{C} \coloneqq C \sqcup \{ d' \}$), and set $\pi(d') \coloneqq d$. Then $\mathrm{CR}(\phi) = \mathrm{CR}_{d \mapsto d'}(\textsc{Compile}(\phi'))$. The new formula ϕ' is defined by replacing every clause $(L, C, \delta) \in \phi$ by a clause $c' = (L, C', \delta') \in \phi'$, where

$$C' = \{ (x, y) \in C \mid y \neq e \},$$

and

$$\delta'(x) = \begin{cases} d' & \text{if } \delta(x) = d \\ \delta(x) & \text{otherwise} \end{cases}$$

for all $x \in \mathrm{Vars}(c') \subseteq \mathrm{Vars}(c)$.

Example 2. Let $\phi = \{ c_1, c_2, c_e \}$ be a formula with clauses (constants lowercase, variables uppercase)

$$\begin{aligned} c_1 &= (\emptyset, \{ (Y, X) \}, \{ X \mapsto b^\top, Y \mapsto b^\top \}), \\ c_2 &= (\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (X, x), (Y, Z) \}, \{ X \mapsto a, Y \mapsto b^\perp, Z \mapsto b^\perp \}), \\ c_3 &= (\{ \neg p(X, Y), \neg p(Z, Y) \}, \{ (X, x), (Z, X), (Z, x) \}, \{ X \mapsto a, Y \mapsto b^\perp, Z \mapsto a \}). \end{aligned}$$

Domain a and with its element $x \in a$ satisfy the preconditions for constraint removal. The operator introduces a new domain a' and transforms ϕ to $\phi' = (c'_1, c'_2, c'_3)$, where

$$\begin{aligned} c'_1 &= c_1 \\ c'_2 &= (\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (Y, Z) \}, \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto b^\perp \}) \\ c'_3 &= (\{ \neg p(X, Y), \neg p(Z, Y) \}, \{ (Z, X) \}, \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto a' \}). \end{aligned}$$

Evaluation.

$$\mathrm{WMC}(\mathrm{CR}_{d \mapsto d'}(n); \sigma) = \begin{cases} \mathrm{WMC}(n; \sigma \cup \{ d' \mapsto \sigma(d) - 1 \}) & \text{if } \sigma(d) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

5.2 A Generalisation of Domain Recursion

TODO: Compare with the original [1].

Precondition. Domain recursion can be applied to domain $d \in \mathcal{D}$ if there is a clause c with a literal variable $v \in \mathrm{Vars}(L_c)$ such that $\delta_c(v) = d$.

The reason for this precondtion is the same as in the initial version of domain recursion: there must be a variable with that domain featured among the literals because it needs to be replaced by a constant. TODO: expand this.

Operator. First, introduce a new constant x. Then $\mathrm{DR}(\phi) = \mathrm{DR}_d(\textsc{Compile}(\phi'))$, where ϕ' is generated by Algorithm 7.

Example 3. Let $\phi = \{ c_1, c_2 \}$ be a formula, where

$$\begin{aligned} c_1 &= (\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (Z,Y) \}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\}), \\ c_2 &= (\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (Z,X) \}, \{X \mapsto a, Y \mapsto b, Z \mapsto a\}). \end{aligned}$$

While domain recursion is possible on both domains, here we illustrate how it works on a.

Suppose Line 2 picks $c = c_1$ first. Then $V = \{ X \}$. Both subsets of V satisfy the conditions on Line 4 and generate new clauses

$$(\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (Z,Y), (X,x) \}, \{X \mapsto a, Y \mapsto b, Z \mapsto b\}),$$

(from $W = \emptyset$) and

$$(\{ \neg p(x,Y), \neg p(x,Z) \}, \{ (Z,Y) \}, \{Y \mapsto b, Z \mapsto b\})$$

(from $W = V$).

When $c = c_2$, then $V = \{ X, Z \}$. The subset $W = V$ fails to satisfy the first condition because of the $Z \neq X$ constraint; without this condition, the resulting clause would have an unsatisfiable constraint $x \neq x$. The other three subsets of V all generate clauses for ϕ':

$$(\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (Z,X), (X,x), (Z,x) \}, \{X \mapsto a, Y \mapsto b, Z \mapsto a\})$$

(from $W = \emptyset$),

$$(\{ \neg p(x,Y), \neg p(Z,Y) \}, \{ (Z,x) \}, \{Y \mapsto b, Z \mapsto a\})$$

(from $W = \{ X \}$), and

$$(\{ \neg p(X,Y), \neg p(x,Y) \}, \{ (X,x) \}, \{X \mapsto a, Y \mapsto b, \})$$

(from $W = \{ Z \}$).

Evaluation.

$$\mathrm{WMC}(\mathrm{DR}_d(n); \sigma) = \begin{cases} \mathrm{WMC}(n; \sigma) & \text{if } \sigma(d) > 0 \\ 1 & \text{otherwise.} \end{cases}$$

One is picked as the multiplicative identity.

6 Other Topics

- new rules that don't create vertices (e.g., duplicate removal, unconditional contradiction detection, etc.)
- some notes on halting
  - Search is infinite. Some rules increase the size of the formula(s), but most reduce it.
  - Inference is guaranteed to terminate if at least one domain shrinks by at least one. But note that allowing recursive calls with the same domain sizes (e.g., $f(n) = f(n) + \dots$) could be useful because these problematic terms might cancel out.
  - It's impossible for $n \leftarrow n-1$ and `for` $n \in \dots$ to combine in a way that results in an infinite loop.
- care should be taken when cloning to preserve the validity of the cache and avoid infinite cycles (we use a separate (node $\rightarrow$ node) cache for this)
- No (more) caching during inference.

7 How to Evaluate an FCG

Along with the three vertex types described above, here are all the other ones. This section is mostly just taken from [2] but with some changes in notation.

TODO: explain that x, y, z refer to vertices, c refers to a clause, and describe each vertex type in a bit more detail.

tautology $\mathrm{WMC}(\textcircled{\top}; \sigma) = 1$

contradiction $\mathrm{WMC}(\textcircled{\bot} c; \sigma) = 0^{\mathrm{gr}(c;\sigma)}$

unit clause

$$\mathrm{WMC}(\textcircled{l} c; \sigma) = \begin{cases} \mathrm{w}(p)^{\mathrm{gr}(c;\sigma)} & \text{if the literal is positive} \\ \overline{\mathrm{w}}(p)^{\mathrm{gr}(c;\sigma)} & \text{otherwise,} \end{cases}$$

where p is the predicate of the literal.

smoothing $\mathrm{WMC}(\bigcirc c; \sigma) = (\mathrm{w}(p) + \overline{\mathrm{w}}(p))^{\mathrm{gr}(c;\sigma)}$, where p is the predicate of the literal.

decomposable conjunction $\mathrm{WMC}(x \owedge y; \sigma) = \mathrm{WMC}(x; \sigma) \times \mathrm{WMC}(y; \sigma)$

deterministic disjunction $\mathrm{WMC}(x \ovee y; \sigma) = \mathrm{WMC}(x; \sigma) + \mathrm{WMC}(y; \sigma)$

decomposable set-conjunction $\mathrm{WMC}(\owedge_D x; \sigma) = \mathrm{WMC}(x; \sigma)^{\sigma(D)}$

deterministic set-disjunction $\mathrm{WMC}(\ovee_{D \subseteq S} x; \sigma) = \sum_{d=0}^{\sigma(S)} \binom{\sigma(S)}{d} \mathrm{WMC}(x; \sigma \cup \{\, D \mapsto d \,\})$

inclusion-exclusion $\mathrm{WMC}(\mathrm{IE}(x, y, z); \sigma) = \mathrm{WMC}(x; \sigma) + \mathrm{WMC}(y; \sigma) - \mathrm{WMC}(z; \sigma)$

7.1 Observations and Theoretical Results

TODO: explain how FCGs and domain sizes connect to functions on integers

Observation 1. Let n be a positive integer and $d \in \mathcal{D}$ a domain. Then one can construct an FCG x such that

$$\mathrm{WMC}(x; \sigma) = \begin{cases} 0 & \text{if } \sigma(d) \geq n \\ \mathrm{WMC}(y; \sigma) & \text{otherwise,} \end{cases}$$

where y is a vertex of arbitrary type. Indeed, $(\textcircled{\bot} c) \owedge y$ is such an FCG, where $c = (L, C, \delta)$ is a clause with n variables $(v_i)_{i=1}^n$ such that:

- L is unimportant,
- $C = \{\, (v_i, v_j) \mid i = 1, \ldots, n-1,\ j = i+1, \ldots, n \,\}$,
- and $\delta(v_i) = d$ for all $i = 1, \ldots, n$.

TODO: explain why this works.

Remark. Such FCGs can be constructed automatically using compilation rules, although n is upper bounded by the maximum number of variables in any clause of the input formula since there is no rule that would introduce new variables.

8 Examples

ForcLift fails on all of these.

TODO.

- Explain the algebraic notation that I'm using here (e.g., that f is always the main function)
- Explain the Knuth's bracket notation (or use something else, since I'm already using brackets for lists)
- Explain the importance of comparing domain sizes to 2.
- Maybe a big table: name, formula, optimal solution and its complexity, best found solution, its depth, its complexity. Also mention that it only takes a few seconds to find these solutions.
- Have at least one example of how an FCG maps to recursive functions
- Going beyond depth 6 (or sometimes even completing depth 6) is computationally infeasible with the current implementation.
- max time to search through depth 5?
- 1d bijections
  - depth 3 exponential solution:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} g(n, m) \\ g(n, m) &= \sum_{l=0}^{n} \binom{n}{l} [l < 2] g(n-l, m-1) \\ &= g(n, m-1) + ng(n-1, m-1), \end{aligned}$$

    which works with base case $g(n, 0) = 1$.
  - depth 4 gives 3 more solutions:
    1. a $\Theta(n^3)$ number of recursive calls (the first definition is exactly as before):

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} g(n, m) \\ g(n, m) &= \sum_{l=0}^{m} \binom{m}{l} [l < 2](1 + (-1)[l < 1]) g(n-1, m-l) \\ &= mg(n-1, m-1), \end{aligned}$$

       which works with base case $g(0, m) = 1$.
    2. exponential:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} \sum_{l=0}^{n} \binom{n}{l} (-1)^{n-l} g(m, l) \\ g(m, l) &= \sum_{k=0}^{m} \binom{m}{k} [k < 2] g(m-k, l-1) \\ &= g(m, l-1) + mg(m-1, l-1), \end{aligned}$$

       which works with base case $g(m, 0) = 1$.
    3. same as the one before but with the parameters of g swapped.
  - depth 5 gives 2 more solutions:

1. a $\Theta(n^2)$ number of recursive calls:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} g(n,m) \\ g(n,m) &= -g(n-1,m) + \sum_{l=0}^{m} \binom{m}{l} [l<2] g(n-1,m-l) \\ &= m g(n-1,m-1), \end{aligned}$$

   which works with base case $g(0,m) = 1$.
2. essentially the same as the first solution of depth 4 but the same g is expressed differently:

$$g(n,m) = -g(n-1,m) + \sum_{l=0}^{m} \binom{m}{l} [l<2] g(n-1,m-l).$$

- 1d injections (same as 1d bijections, but note that the 2d case is different) (no liftable solutions)
  - depth 3 gives 2 solutions:
    1. exponential:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} g(n,m) \\ g(n,m) &= \sum_{l=0}^{m} \binom{m}{l} [l<2] g(n-1,m-l) \\ &= g(n-1,m) + m g(n-1,m-1), \end{aligned}$$

       which works with base case $g(0,m) = 1$.
    2. same as the one before but with the parameters of g swapped.
  - depth 5 gives 4 more solutions:
    1. a very complex exponential solution:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} \sum_{l=0}^{m} \binom{m}{l} [l<2] g(n-1,m-l) \\ &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} (g(n-1,m) + m g(n-1,m-1)) \\ g(j,k) &= \sum_{i=0}^{k} \binom{k}{i} [i<2] g(j-1,k-i) \\ &= g(j-1,k) + k g(j-1,k-1), \end{aligned}$$

       which works with base case $g(0,k) = 1$.
    2. same as the one before but with the parameters of g swapped.

3. exponential:

$$\begin{aligned} f(n) &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} \sum_{l=0}^{n} \binom{n}{l} [l < 2] g(m-1, n-l) \\ &= \sum_{m=0}^{n} \binom{n}{m} (-1)^{n-m} (g(m-1, n) + ng(m-1, n-1)) \\ g(j,k) &= \sum_{i=0}^{j} \binom{j}{i} [i < 2] g(j-i, k-1) \\ &= g(j, k-1) + jg(j-1, k-1), \end{aligned}$$

   which works with base case $g(j,0) = 1$.

4. same as the one before but with the parameters of g swapped.

  – 2 more solutions at depth 6, but who cares?

- 1d partial injections. 2 solutions at depth 6, but they're too complicated to check by hand. A contradiction with $X \neq x$ constraints makes things complicated.

- 2d bijections

  – depth 3 gives 3 solutions:

1. linear:

$$\begin{aligned} f(m,n) &= \sum_{l=0}^{m} \binom{m}{l} [l < 2](1 - [l < 1]) f(m-l, n-1) \\ &= mf(m-1, n-1), \end{aligned}$$

   which works with base cases $f(0,0) = 1$, $f(0,n) = 0$, $f(m,0) = 0$.

2. same but with m and n switched.

3.

$$\begin{aligned} f(m,n) &= \sum_{l=0}^{m} \binom{m}{l} (-1)^{m-l} g(l,n) \\ g(l,n) &= \sum_{k=0}^{n} \binom{n}{k} [k < 2] g(l-1, n-k) \\ &= g(l-1, n) + ng(l-1, n-1), \end{aligned}$$

   which works with base cases $g(0,0) = g(l,0) = 1$ and $g(0,n) = 0$.

- 2d injections

- 2d partial injections

9 Conclusions and Future Work

Conclusions and observations.

- CR must be separate from DR because initially the requirement to not have the newly introduced constant in the literals is not satisfied.

Future work.

- Transform FCGs to definitions of (possibly recursive) functions on integers. Use a computer algebra system to simplify them.
- Design an algorithm to infer the necessary base cases. (Note that there can be an infinite amount of them when functions have more than one parameter.)
- Observation: -1 (and powers thereof) appear in every solution to a formula if and only if the formula has existential quantification. That's not very smart! By putting unit propagation into Γ, these powers are pushed to the outer layers of the solution (i.e., 'early' in the FCG). It's likely that removing this restriction would enable the algorithm to find asymptotically optimal solutions.

References

[1] Van den Broeck, G. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain* (2011), J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, Eds., pp. 1386–1394.

[2] Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., and De Raedt, L. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011* (2011), T. Walsh, Ed., IJCAI/AAAI, pp. 2178–2185.

Algorithm 5: A recursive function for checking whether one can reuse the FCG for computing WMC(ϕ) to compute WMC(ψ). Both ϕ and ψ are formulas, and $\rho\colon \text{Doms}(\phi) \nrightarrow \text{Doms}(\psi)$ is a partial map.

Function `identifyRecursion`(ϕ, ψ, $\rho = \emptyset$, $\mathsf{foundConstraintRemoval} = \mathtt{false}$)**:**
  if $|\phi| \neq |\psi|$ **or** $\#\phi \neq \#\psi$ **then return** `null`;
  if $\phi = \emptyset$ **then**
    if $\mathsf{foundConstraintRemoval}$ **then return** ρ;
    return `null`;
  foreach *clause* $c \in \phi$ **do**
    foreach *clause* $d \in \psi$ *such that* $\#d = \#c$ **do**
      foreach $(\beta, \gamma) \in$ `generateMaps`(c, d, ρ) *such that* $c[\beta, \gamma] = d$ **do**
        $\mathsf{foundConstraintRemoval}' \leftarrow \mathsf{foundConstraintRemoval}$;
        $\mathsf{suitableBijection} \leftarrow \mathtt{true}$;
        foreach $e \in \text{Doms}(c)$ **do**
          $\mathsf{foundConstraintRemoval}'' \leftarrow$ `traceAncestors`(e, $\gamma(e)$);
          if $\mathsf{foundConstraintRemoval}'' = \mathtt{null}$ **then**
            $\mathsf{suitableBijection} \leftarrow \mathtt{false}$;
            break;
          if $\mathsf{foundConstraintRemoval}''$ **then** $\mathsf{foundConstraintRemoval}' \leftarrow \mathtt{true}$;
        if $\mathsf{suitableBijection}$ **then**
          $\rho'' \leftarrow$ `identifyRecursion`($\phi \setminus \{ c \}$, $\psi \setminus \{ d \}$, $\rho \cup \gamma$, $\mathsf{foundConstraintRemoval}'$);
          if $\rho'' \neq \mathtt{null}$ **then return** ρ'';
    return `null`;

Function `generateMaps`(c, d, ρ)**:**
  foreach *bijection* $\beta\colon \text{Vars}(c) \to \text{Vars}(d)$ **do**
    $\gamma \leftarrow$ `constructDomainMap`($\text{Vars}(c)$, δ_c, δ_d, β, ρ);
    if $\gamma \neq \mathtt{null}$ **then yield** (β, γ);

Function `constructDomainMap`(V, δ_c, δ_d, β, ρ)**:**
  $\gamma \leftarrow \emptyset$;
  foreach $v \in V$ **do**
    if $\delta_c(v) \in \text{dom}(\rho)$ **and** $\rho(\delta_c(v)) \neq \delta_d(\beta(v))$ **then return** `null`;
    if $\delta_c(v) \notin \text{dom}(\gamma)$ **then** $\gamma \leftarrow \gamma \cup \{ \delta_c(v) \mapsto \delta_d(\beta(v)) \}$;
    else if $\gamma(\delta_c(v)) \neq \delta_d(\beta(v))$ **then return** `null`;
  return γ;

Function `traceAncestors`(d, e)**:**
  $\mathsf{foundConstraintRemoval} \leftarrow \mathtt{false}$;
  while $e \neq d$ **and** $e \in \text{dom}(\pi)$ **do**
    if $e \in \mathcal{C}$ **then** $\mathsf{foundConstraintRemoval} \leftarrow \mathtt{true}$;
    $e \leftarrow \pi(e)$;
  if $e = d$ **then return** $\mathsf{foundConstraintRemoval}$;
  return `null`;

Algorithm 6: A generalised version of the compilation rule that uses Algorithm 5 to add Ref vertices (i.e., the edges that make the FCG no longer a tree) to the FCG.

Input: formula ϕ

Output: a vertex of type Ref (or `null`)

foreach $(\psi, v) \in \mathsf{compilationCache}(\#\psi)$ **do**

  $\rho \leftarrow$ `identifyRecursion(`ϕ, ψ`)`;

  if $\rho \neq$ `null` **then return** $\{\, \textsc{Ref}_\rho(v) \,\}$;

return `null`;

Algorithm 7: Formula transformation for domain recursion

Input: formula ϕ, domain $d \in \mathcal{D}$, and constant x

Output: formula ϕ'

$\phi' \leftarrow \emptyset$;

foreach *clause* $c = (L, C, \delta) \in \phi$ **do**

  $V \leftarrow \{\, v \in \mathrm{Vars}(L) \mid \delta(v) = d \,\}$;

  foreach *subset* $W \subseteq V$ *such that* $W^2 \cap C = \emptyset$ **and** $W \cap \{\, v \in \mathrm{Vars}(C) \mid (v, y) \in C$ *for some constant* $y \,\} = \emptyset$ **do**

    `/* Here, ` δ' ` is the restriction of ` δ ` to the new set of variables */`

    $\phi' \leftarrow \phi' \cup \{\, (L[x/W], C[x/W] \cup \{\, (v, x) \mid (v \in V \setminus W) \,\}, \delta') \,\}$;