Recursive Solutions to First-Order Model Counting

15th 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. $\mathrm{Doms}(c) = \operatorname{Im} \delta_c$, and $\mathrm{Doms}(\phi) = \bigcup_{c \in \phi} \mathrm{Doms}(c)$.
- maybe: notation for partial function, notation for powerset, notation for disjoint union, domain size
- 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 $\sigma \colon \mathscr{D} \to \mathbb{N}_0$.
- $\sqcup$ for both sets and functions

TODO

- should I always say 'compilation rule' instead of 'operator'?
- maybe mathcal instead of mathscr
- maybe π is global enough to have a more unique name
- maybe $c[\beta, \gamma]$ is better than $c * (\beta, \gamma)$
- capitalise variable names.
- introduce/use notation for vertices, edges of a graph

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 $\mathscr{D}$ be the set of all domains and $\mathscr{C} \subset \mathscr{D}$ be the subset of domains introduced as a consequence of constraint removal. Note that both sets (can) expand during the compilation phase.

Let $\pi\colon \mathscr{D} \nrightarrow \mathscr{D}$ be a partial endomorphism on $\mathscr{D}$ that denotes the *parent* relation, i.e., if $\pi(d) = e$ for some $d, e \in \mathscr{D}$, 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 $\delta_c\colon \operatorname{Vars}(c) \to \mathscr{D}$ 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 $\delta_c(x) = \delta_c(y)$.

Two clauses c and $d = (L', C', \delta_d)$ are *isomorphic* (written $c \cong d$) if there is a bijection $\beta\colon \operatorname{Vars}(c) \to \operatorname{Vars}(d)$ such that $c\beta = d\beta$. TODO: we will always use this subscript notation for the δ's. Equality of clauses is defined in the usual way (i.e., all variables, domains, etc. must match).

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 $|\operatorname{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted $\#\phi$.

Definition 4. Let $\operatorname{gr}(\cdot; \sigma)$ 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 $v \in \operatorname{Vars}(c)$, let $C_v = \{w \mid (u, w) \in C \setminus \operatorname{Vars}(c)^2,\ u \neq v\}$ be the set of (explicitly named) constants that v is permitted to be equal to. Then

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

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

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

2 Search

Definition 5. A *first-order deterministic decomposable negation normal form computational graph* (FCG) is a (weakly connected) directed graph with a a single source, node labels, and ordered direct successors. We denote an FCG as $G = (V, s, N^+)$, where V is the set of nodes, $s \in V$ is the source, and N^+ is the direct successor function that maps each node in V to a list that contains either other nodes in V or $\star$, which means that the target of the edge is yet to be determined. For each $v \in V$, the length of the list $N^+(v)$ is determined by the label of v.

TODO.

- maybe add a (vertex) labelling function and its codomain (and use it in the algorithm)? But it's only used once...

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

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

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 solutions

```
1  solutions ← ∅;
2  C_0 ← ∅;
3  (G_0, C_0, L_0) ← applyGreedyRules(φ_0, C_0);
4  if L_0 = [] then solutions ← { G_0 };
5  else
6      q ← an empty queue of states;
7      q.put((G_0, C_0, L_0));
8      while not q.empty() do
9          foreach state (G, C, L) ∈ applyNonGreedyRules(q.get()) do
10             if L = [] then solutions ← solutions ∪ { G };
11             else q.put((G, C, L));
```

- explain the connection to the algebraic formalism (or just don't use it).
- define 'an FCG for a formula'
- vertex or node, v or n?
- what notation for lists (a.k.a. sequences) am I going to use? $\oplus$ for concatenation (of both two lists and an element to a list), $\in$ for (in-order) enumeration, [] for an empty list, $[x]$ for a list with one element, $|L|$ for the length of the list, and $h : t$ to denote a list with first element (a.k.a. head) h and remaining list (a.k.a. tail) t.
- use Greek letters for compilation rules

Definition 6. A *state* (of the search for an FCG for a given formula) is a tuple (G, C, L), where:

- G is an FCG (can be `null`),
- C is a compilation cache that maps integers to sets of pairs (ϕ, n), where ϕ is a formula, and n is a node 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 7. 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 node and a bunch of $\star$'s marking a fixed number of outedges.

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 nodes.

TODO

- maybe 'applyAllRules' is a better name
- explain the 'tail' part of the algorithm

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.

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* $r \in \Gamma$ **do**
    $S \leftarrow r(\phi)$;
    if $S \neq \emptyset$ **then**
      $(G, L) \leftarrow$ any element of S;
      if $G =$ `null` **then return** `applyGreedyRules`(*the only element of* L, C);
      else
        $(V, s, N^+) \leftarrow G$;
        $C \leftarrow$ `updateCache`(C, ϕ, s);
        return `applyGreedyRulesToAllFormulas`(G, C, L);
  return $(\texttt{null}, C, [\phi])$;
Function `applyGreedyRulesToAllFormulas`(G, C, L)**:**
  $(V, s, N^+) \leftarrow G$;
  $N^+(s) \leftarrow []$;
  $L' \leftarrow []$;
  foreach *formula* $\phi \in L$ **do**
    $(G', C, L'') \leftarrow$ `applyGreedyRules`(ϕ, C);
    $L' \leftarrow L' \oplus L''$;
    if $G' =$ `null` **then** $N^+(s) \leftarrow N^+(s) \oplus [\star]$;
    else
      $(V', s', N') \leftarrow G'$;
      $V \leftarrow V \sqcup V'$;
      $N^+ \leftarrow N^+ \sqcup N'$;
      $N^+(s) \leftarrow N^+(s) \oplus [s']$;
  return $((V, s, N^+), C, L')$;
Function `applyNonGreedyRules`(s)**:**
  $(G, C, L) \leftarrow s$;
  $\phi : T \leftarrow L$;
  $(G', C', L') \leftarrow$ a copy of s;
  newStates $\leftarrow []$;
  foreach *rule* $r \in \Delta$ **do**
    foreach $(G'', L'') \in r(\phi)$ **do**
      if $G'' =$ `null` **then** newStates $\leftarrow$ newStates $\oplus$ `applyNonGreedyRules`$((G', C', L''))$;
      else
        $(V, s, N^+) \leftarrow G''$;
        $C' \leftarrow$ `updateCache`(C', ϕ, s);
        $(G'', C', L'') \leftarrow$ `applyGreedyRulesToAllFormulas`(G'', C', L'');
        if $G' =$ `null` **then** newStates $\leftarrow$ newStates $\oplus [(G'', C', L'' \oplus T)]$;
        else newStates $\leftarrow$ newStates $\oplus$ [(`mergeFcgs`(G', G''), $C', L'' \oplus T$)];
    $(G', C', L') \leftarrow$ a copy of s;
  return newStates;

Algorithm 3: Helper functions used by Algorithm 2

```
1  Function updateCache(C, φ, v):
2      if #φ ∉ dom(C) then return C ∪ { #φ ↦ (φ, v) };
3      if there is no (φ′, v′) ∈ C(#φ) such that v′ = v then C(#φ) ← (φ, v) ⊕ C(#φ);
4      return C;
5  Function mergeFcgs(G = (V, s, N⁺), G′ = (V′, s′, N′), r = s):
6      if r has label REF then return null;
7      foreach t ∈ N⁺(r) do
8          if t = ⋆ then
9              replace t with s′ in N⁺(r);
10             return (V ⊔ V′, s, N⁺ ⊔ N′);
11         G″ ← mergeFcgs(G, G′, t);
12         if G″ ≠ null then return G″;
13     return null;
```

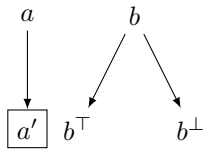

Figure 1: The forest of domains from Example 1. The directed edges correspond to the pairs of domains in π but in the opposite direction, e.g., $\pi(a') = a$. The domain enclosed in a rectangle is the only domain created by constraint removal, i.e., $\mathscr{C} = \{ a' \}$.

3 Identifying Possibilities for Recursion

Definition 8 (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 $\gamma\colon \mathrm{Doms}(c) \to \mathscr{D}$, 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 $\delta_d\colon V \to \mathscr{D}$ defined as $\delta_d := \gamma \circ \delta_c \circ \beta^{-1}$. In other words, δ_d is the unique function that makes the following diagram commute:

$$\begin{array}{ccc} \mathrm{Vars}(c) & \xrightarrow{\ \beta\ } & V = \mathrm{Vars}(d) \\ {\scriptstyle \delta_c}\downarrow & & \vdots\ {\scriptstyle \exists! \delta_d} \\ \mathrm{Doms}(c) & \xrightarrow[\ \gamma\ ]{} & \mathscr{D}. \end{array}$$

The function `traceAncestors` returns `null` if domain $d \in \mathscr{D}$ is not an ancestor of domain $e \in \mathscr{D}$. 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

- introduce/describe Algorithm 4 and Algorithm 5 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 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):

$$\begin{array}{ccc} \mathrm{Vars}(c) & \xrightarrow{\ \beta\ } & \mathrm{Vars}(d) \\ {\scriptstyle \delta_c}\downarrow & & \downarrow{\scriptstyle \delta_d} \\ \mathrm{Doms}(c) & \overset{\gamma}{\dashrightarrow} & \mathrm{Doms}(d) \\ \downarrow & & \downarrow \\ \mathscr{D} & \xrightarrow[\ \rho\ ]{} & \mathscr{D}. \end{array}$$

Example 1. Let ϕ be the formula

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

and ψ be the formula

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

The relevant domains and the definition of π is in Fig. 1. 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 the $c * (\beta, \gamma) = d$ condition.

Since $\pi(a') = a$, and $a' \in \mathscr{C}$, `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 $\text{Doms}(c)$ and $\text{Doms}(d)$, γ exists and is equal to ρ. Again, the returned pair (β, γ) satisfies the condition $c * (\beta, \gamma) = d$. 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 a circuit for $\text{WMC}(\phi)$ to compute $\text{WMC}(\psi)$ by replacing every mention of a with a' and every mention of b with $b^\perp$.

3.1 Evaluation

$\text{WMC}(\text{Ref}_\rho(n); \sigma) = \text{WMC}(n; \sigma')$ (n is the target circuit node), where σ' is defined as

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

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

4 New Operations

4.1 Constraint Removal

TODO: introduce the Forclift-style notation (e.g., the compile operator) and the arbitrary formula ϕ.

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

- for each clause $c = (L, C, \delta_c) \in \phi$ and variable $v \in \text{Vars}(c)$, 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., $\mathscr{D} \coloneqq \mathscr{D} \sqcup \{ d' \}$), add it to $\mathscr{C}$ (i.e., $\mathscr{C} \coloneqq C \sqcup \{ d' \}$), 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)

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

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

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

Evaluation.

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

4.2 A Generalisation of Domain Recursion

TODO: Compare with the original. The original version of domain recursion is here [1].

TODO: introduce ϕ.

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

Operator. First, introduce a new constant x. Then $\mathrm{DR}(\phi) = \mathrm{DR}_d(\textsc{Compile}(\phi'))$, where ϕ' is generated by Algorithm 6.

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.

5 Other Topics

- domains, smoothing, and avoiding infinite cycles
- new rules that don't create nodes (e.g., duplicate removal, unconditional contradiction detection, etc.)
- my search algorithm
- 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.

6 Circuit Nodes Types and Their Evaluations

Along with the three circuit node 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 nodes, c refers to a clause, and describe each node type in a bit more detail.

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

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

unit clause

$$\mathrm{WMC}(①c; \sigma) = \begin{cases} \mathrm{w}(p)^{\mathrm{gr}(c;\sigma)} & \text{if the literal is positive} \\ \overline{\mathrm{w}}(p)^{\mathrm{gr}(c;\sigma)} & \text{otherwise,} \end{cases}$$

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 $\mathrm{WMC}(x \mathbin{⩓} y; \sigma) = \mathrm{WMC}(x; \sigma) \times \mathrm{WMC}(y; \sigma)$

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

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

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

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

TODO: define 'circuit' (or something similar). Note that circuits are rooted.

6.1 Observations and Theoretical Results

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

Observation 1. Let n be a positive integer and $d \in \mathscr{D}$ a domain. Then one can construct a circuit 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 circuit node of arbitrary type. Indeed, $(⨁ c) \mathbin{⩓} y$ is such a circuit, 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 (sub)circuits 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.

7 Examples

For both one domain and two domains:

- bijections
- injections
- partial injections

8 Conclusions and Future Work

- Transform circuits 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.)

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

```
Function identifyRecursion(φ, ψ, ρ = ∅, foundConstraintRemoval = false):
    if |φ| ≠ |ψ| or #φ ≠ #ψ then return null;
    if φ = ∅ then
        if foundConstraintRemoval then return ρ;
        return null;
    foreach clause c ∈ φ do
        foreach clause d ∈ ψ such that #d = #c do
            foreach (β, γ) ∈ generateMaps(c, d, ρ) such that c ∗ (β, γ) = d do
                foundConstraintRemoval′ ← foundConstraintRemoval;
                suitableBijection ← true;
                foreach e ∈ Doms(c) do
                    foundConstraintRemoval″ ← traceAncestors(e, γ(e));
                    if foundConstraintRemoval″ = null then
                        suitableBijection ← false;
                        break;
                    if foundConstraintRemoval″ then foundConstraintRemoval′ ← true;
                if suitableBijection then
                    ρ″ ← identifyRecursion(φ \ {c}, ψ \ {d}, ρ ∪ γ, foundConstraintRemoval′);
                    if ρ″ ≠ null then return ρ″;
        return null;
Function generateMaps(c, d, ρ):
    foreach bijection β: Vars(c) → Vars(d) do
        γ ← constructDomainMap(Vars(c), δ_c, δ_d, β, ρ);
        if γ ≠ null then yield (β, γ);
Function constructDomainMap(V, δ_c, δ_d, β, ρ):
    γ ← ∅;
    foreach v ∈ V do
        if δ_c(v) ∈ dom(ρ) and ρ(δ_c(v)) ≠ δ_d(β(v)) then return null;
        if δ_c(v) ∉ dom(γ) then γ ← γ ∪ {δ_c(v) ↦ δ_d(β(v))};
        else if γ(δ_c(v)) ≠ δ_d(β(v)) then return null;
    return γ;
Function traceAncestors(d, e):
    foundConstraintRemoval ← false;
    while e ≠ d and e ∈ dom(π) do
        if e ∈ 𝒞 then foundConstraintRemoval ← true;
        e ← π(e);
    if e = d then return foundConstraintRemoval;
    return null;
```

Algorithm 5: A generalised version of the compilation rule that uses Algorithm 4 to add REF nodes (i.e., the edges that make the circuit no longer a tree) to the circuit. TODO: refactor to return a set of solutions.

Input: formula ϕ
Output: a REF circuit node (or `null`)

```
1 foreach (ψ, n) ∈ compilationCache(#ψ) do
2     ρ ← identifyRecursion(φ, ψ);
3     if ρ ≠ null then return REF_ρ(n);
4 return null;
```

Algorithm 6: Formula transformation for domain recursion

Input: formula ϕ, domain $d \in \mathscr{D}$, 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** $W \cap \{ v \in \mathrm{Vars}(C) \mid (v, y) \in C \text{ for some constant } y \} = \emptyset$ **do**
    $\phi' \leftarrow \phi' \cup \{ (L[x/W], C[x/W] \cup \{ (v, x) \mid (v \in V \setminus W) \}, \delta') \}$;
    /* δ' is the restriction of δ to the new set of variables */