Empowering Domain Recursion in Symmetric Weighted First-Order Model Counting

19th January 2022

1 Basic Definitions

TODO (later): maybe mathcal instead of mathscr

Things I might need to explain.

- notation: Im
- atom, literal
- inequality constraint
- Vars, $\mathrm{Vars}(c) = \mathrm{Vars}(P) \cup \mathrm{Vars}(N) \cup \mathrm{Vars}(C)$
- Doms on both formulas and clauses. $\mathrm{Doms}(c) = \mathrm{Im}\,\delta_c$, and $\mathrm{Doms}(\phi) = \bigcup_{c \in \phi} \mathrm{Doms}(c)$.
- the hash codes of clauses and formulas. Introduce the # notation.
- substitution
- size of a domain, how each domain is partitioned into two during compilation.
- maybe: notation for partial function, notation for powerset, domain size
- notation for projection (or avoid it?)
- WMC
- constraint removal operation
- two parts: compilation and inference.
- introduce and use arrows for bijections, injections, set inclusions, etc.

Let $\mathscr{V}$ be the set of circuit nodes.

TODO: maybe π is global enough to have a more unique name.

Definition 1. A *domain* is a set with elements not used anywhere else.[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.

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

Definition 2. A *clause* is a triple $c = (P, N, C, \delta_c)$, where P and N are sets of atoms interpreted as positive and negative literals respectively, C is a set of inequality constraints, and $\delta_c \colon \mathrm{Vars}(c) \to \mathscr{D}$ is a function that maps all variables in c to their domains. Two clauses c and $d = (P', N', C', \delta_d)$ are *isomorphic* (written $c \cong d$) if there is a bijection $\beta \colon \mathrm{Vars}(c) \to \mathrm{Vars}(d)$ 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.

2 Identifying Possibilities for Recursion

Definition 3 (Notation)**.** For any clause $c = (P, N, C, \delta_c)$, 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 P, N, and C replaced with $\beta(v)$ (so $\mathrm{Vars}(d) = V$) and $\delta_d \colon V \to \mathscr{D}$ 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:

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

The function `traceAncestors` returns `null` if domain $c \in \mathscr{D}$ is not an ancestor of domain $d \in \mathscr{D}$. Otherwise, it returns `true` if the size of d is guaranteed to be strictly smaller than the size of c (i.e., there is domain created by the constraint removal rule on the path from c to d) 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 why $\rho \cup \gamma$ is possible.

TODO: explain what the second return statement is about and why a third one is not necessary.

TODO: mention which one is the main function, what each function takes and returns.

Example 1.

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

to

$$\forall X \in a. \forall Y \in b. \forall Z \in b. Y \neq Z \implies \neg p(X, Y) \vee \neg p(X, Z)$$
$$\forall X \in a. \forall Y \in b. \forall Z \in a. X \neq Z \implies \neg p(X, Y) \vee \neg p(Z, Y)$$

and mention Fig. 1. Solution map (stored as the edge label):

$$\rho(a') = (a, \{ (\diamondsuit, 0) \})$$
$$\rho(b^{\perp}) = (b, \{ (\heartsuit, 1) \})$$

TODO: conclude with a description of the inference rule and the node/edge type.

3 New Node Types

3.1 Improved Domain Recursion

The original version of domain recursion is here [1].

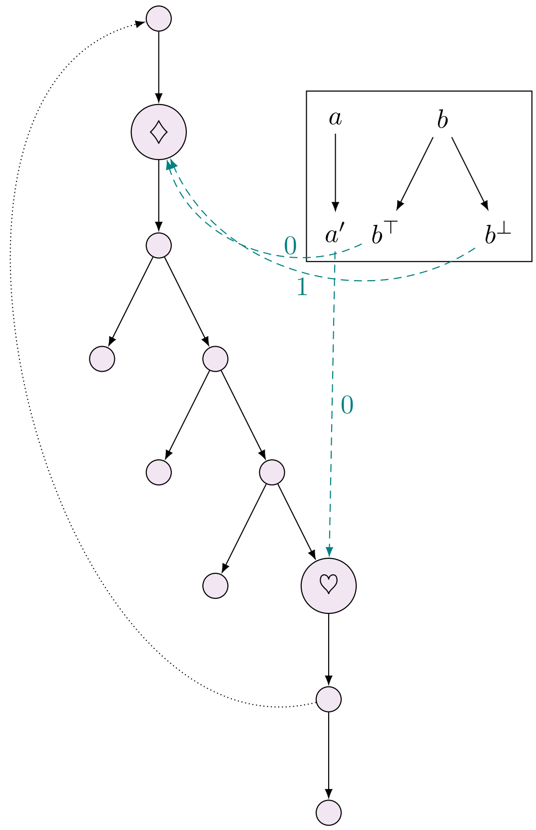

Figure 1: A circuit (outside the box) and a forest of domains (inside the box). The dotted line on the left will be added if `identifyRecursion` returns a non-`null` mapping. The dashed edges with their labels represent the κ function. The circuit nodes with symbols in them are the nodes that introduce new (sub)domains.

3.2 Constraint Removal

4 Other Topics

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

5 Circuit Evaluation

TODO: new node types, their algebraic/graphical representation, what info they hold, and how they're created.

TODO: describe evaluation of: and, counting, constraint removal, ref, unit, contradiction, improved domain recursion. Most of this will be from [2].

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

Function `identifyRecursion`(ϕ, ψ, $\rho = \emptyset$, foundConstraintRemoval = `false`)**:**
  if $|\phi| \neq |\psi|$ **or** $\#\phi \neq \#\psi$ **then return** `null`;
  if $\phi = \psi = \emptyset$ **then**
    if 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**
        foundConstraintRemoval′ ← foundConstraintRemoval;
        suitableBijection ← `true`;
        foreach $v \in \mathrm{Vars}(c)$ **do**
          foundConstraintRemoval″ ← `traceAncestors`($\delta_c(v)$, $\delta_d(\beta(v))$);
          if foundConstraintRemoval″ = `null` **then**
            suitableBijection ← `false`;
            break;
          if foundConstraintRemoval″ **then** foundConstraintRemoval′ ← `true`;
        if suitableBijection **then**
          ρ'' ← `identifyRecursion`($\phi \setminus \{ c \}$, $\psi \setminus \{ d \}$, $\rho \cup \gamma$, foundConstraintRemoval′);
          if $\rho'' \neq$ `null` **then return** ρ'';
    return `null`;

Function `generateMaps`(c, d, ρ)**:**
  $M \leftarrow \emptyset$;
  foreach *bijection* $\beta\colon \mathrm{Vars}(c) \to \mathrm{Vars}(d)$ **do**
    if $\forall v \in \mathrm{Vars}(c).(\delta_c(v) \notin \mathrm{dom}(\rho) \lor \rho(\delta_c(v)) = \delta_d(\beta(v)))$ **then**
      $\gamma \leftarrow$ `constructDomainMap`($\mathrm{Vars}(c)$, δ_c, δ_d, β);
      if $\gamma \neq$ `null` **then** $M \leftarrow M \cup \{ (\beta, \gamma) \}$;
  return M;

Function `constructDomainMap`(V, δ_c, δ_d, β)**:**
  $\gamma \leftarrow \emptyset$;
  foreach $v \in V$ **do**
    if $\delta_c(v) \notin \mathrm{dom}(\gamma)$ **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`(c, d)**:**
  foundConstraintRemoval ← `false`;
  while $d \neq c$ **and** $d \in \mathrm{dom}(\pi)$ **do**
    if $d \in \mathscr{C}$ **then** foundConstraintRemoval ← `true`;
    $d \leftarrow \pi(d)$;
  if $d = c$ **then return** foundConstraintRemoval;
  return `null`;