Ideas about grounding

Johan Wittocx

Department of Computer Science, Katholieke Universiteit Leuven, Belgium

August 2011

Notations

- $\mathcal{L}$: a logic
- $\mathcal{G}$: some propositional (i.e., quantifier-free) logic
- Σ, Σ_g: vocabularies
- tr: a mapping from (three-valued) Σ_g-structures to (three-valued) Σ-structures

  Note 1: if J is two-valued, then $\mathrm{tr}(J)$ is not necessarily two-valued.

  Note 2: tr is not necessarily one-to-one.

Grounding

Definition (Grounding)

A *grounding* of a $\mathcal{L}$ theory T over Σ with respect to a (three-valued) Σ-structure I is a $\mathcal{G}$ theory T_g over Σ_g and translation function tr such that for every model J of T_g, $\mathrm{tr}(J)$ is more precise than I and every two-valued structure that is approximated by J, is a model of T.

Full grounding

Straightforward grounding of T and I over Σ:

- tr:
  1. Σ_g contains a 0-ary predicate P_a for each ground atom a of the form $P(d_1, \ldots, d_n)$ or $F(d_1, \ldots, d_n) = d_{n+1}$, where $P, F \in \Sigma$ and $d_1, \ldots, d_n \in \mathrm{dom}(I)$.
  2. $\mathrm{tr}(J)$ is the structure with the same domain as I, defined by $\mathrm{tr}(J)(a) = J(P_a)$.
- T_g:
  1. Remove all nested terms (e.g.: $P(C) \longrightarrow \forall x\ (C = x \Rightarrow P(x))$)
  2. Substitute variables by domain elements:
     - $\forall x\ \varphi[x] \longrightarrow \bigwedge_{d \in \mathrm{dom}(I)} \varphi[d]$
     - $\exists x\ \varphi[x] \longrightarrow \bigvee_{d \in \mathrm{dom}(I)} \varphi[d]$
     - $\{x : \varphi[x] : t[x]\} \longrightarrow \{(d, \varphi[d], t[d]) \mid d \in \mathrm{dom}(I)\}$
  3. Replace all atoms a by P_a.
  4. For every a that is true (false) in I, add the unit clause P_a ($\neg P_a$)
  5. Add function constraints

From ground atoms to integers

SAT-solver expects integers to denote atoms.
Question: how to efficiently translate ground atoms to unique numbers?

- Construct an explicit mapping (during grounding). Possible datastructures: hash map, red-black tree, AVL tree, ...

  Currently in GIDL: mapping per atom, implemented using STL's `map` (red-black tree).
- Offset system. E.g. if I has 10 domain elements, P has offset 50, and d_i is the ith domain element, $P(d_3, d_5)$ translates to 85 $(50 + 10 \times 3 + 5)$. Offsets are chosen such that two atoms over different predicates map to different numbers.

  Positive: efficient. Negative: large numbers needed, not all numbers appear in the ground theory but SAT datastructures are nevertheless initialized for all numbers between 1 and the largest number that occurs in its input.

Note: on some benchmarks, translation may take half of the total grounding time when an explicit mapping is created!

Complexity

- $\mathrm{qr}(\varphi)$ = quantifier rank of φ = maximal nesting depth of quantifiers in φ

  $\mathrm{qr}(\forall x \forall y\ P(x,y)) = 2$

  $\mathrm{qr}((\exists x\ P(x)) \wedge (\forall y\ Q(y))) = 1$

- Size of the full grounding $= O(|\mathrm{dom}(I)|^{\mathrm{qr}(T')})$ (T' is the theory in step 2)

  $\Rightarrow$ infinite domain $\Rightarrow$ infinite full grounding

  $\Rightarrow$ strategy for smaller grounding: reduce quantifier rank

Reduced grounding

- Replace in the grounding P_a by $\top$ ($\bot$) if a is true in I.
- Recursively replace
  - $\ldots \wedge \bot \wedge \ldots$ by $\bot$
  - $\ldots \vee \top \vee \ldots$ by $\top$
  - $\varphi_1 \wedge \top \wedge \varphi_2$ by $\varphi_1 \wedge \varphi_2$
  - $\varphi_1 \vee \bot \wedge \varphi_2$ by $\varphi_1 \vee \varphi_2$

Note: heads of rules cannot be replaced in this manner!

Note on step 4

"For every a that is true (false) in I, add the unit clause P_a ($\neg P_a$)"
This step is not necessary when the reduced grounding is created.
Then, atoms that are not unknown in I do not occur in the grounding anymore (unless they are defined by a rule in the grounding). Hence, they should not occur in Σ_g (tr should take this into account).
More precise input structure $I \Rightarrow$ smaller grounding.

Top-down, depth first grounding algorithm

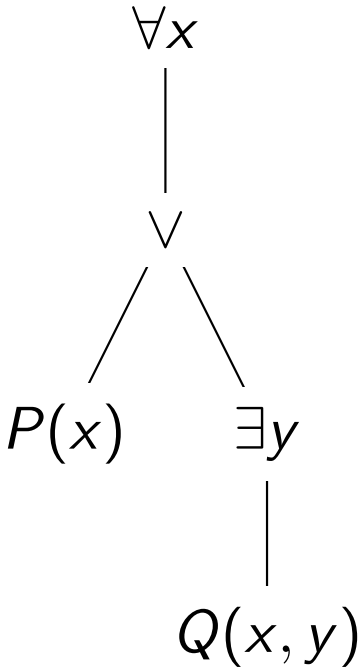
Input: Sentence φ, structure I
Output: Grounding of φ with respect to I
switch φ **do**
  case φ *is a ground atom* a **return** a;
  case $\varphi = \psi_1 \vee \ldots \vee \psi_n$
    $\mathcal{D} := \emptyset$;
    for $i = 1 \ldots n$ **do** $\mathcal{D} := \mathcal{D} \cup \texttt{ground}(\psi_i, I)$;
    return $\bigvee \mathcal{D}$;
  end
  case $\varphi = \psi_1 \wedge \ldots \wedge \psi_n$ `/* similar to disjunction */`;
  case $\varphi = \forall x\ \psi[x]$
    $\mathcal{C} := \emptyset$;
    for $d \in \mathrm{dom}(I)$ **do** $\mathcal{C} := \mathcal{C} \cup \texttt{ground}(\psi[d], I)$;
    return $\bigwedge \mathcal{C}$;
  end
  case $\varphi = \exists x\ \psi[x]$ `/* similar to univ. quantifier */`;
end

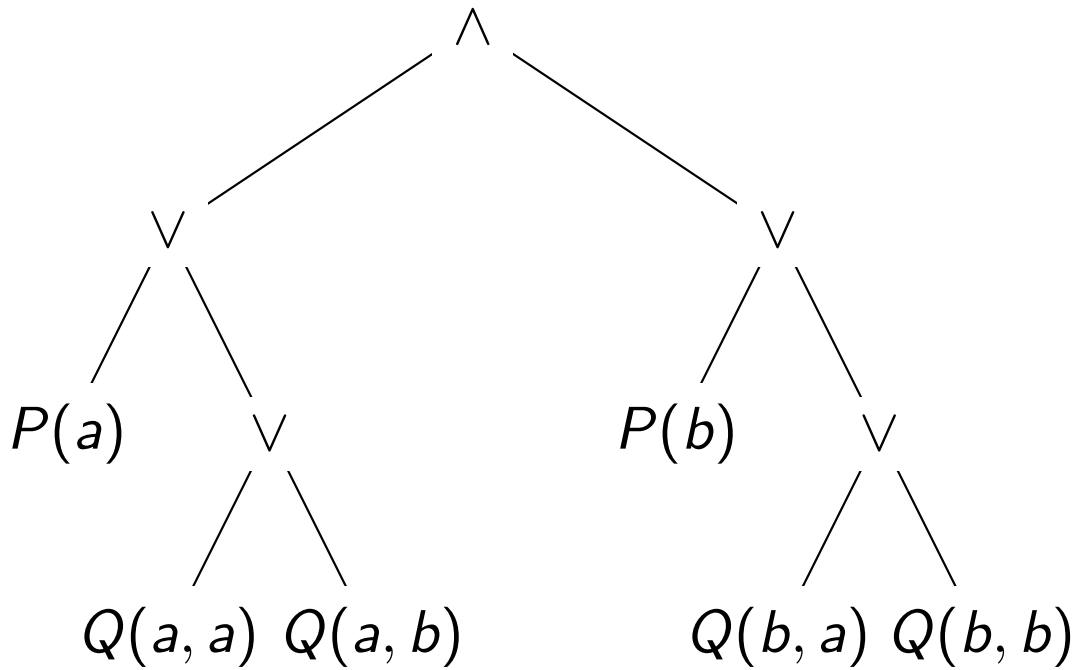
Main loop

Input: Theory T, structure I
Output: Grounding of T with respect to I
$T_g := \emptyset$;
for $\varphi \in T$ **do**
    $T_g := T_g \cup \texttt{ground}(\varphi, I)$;
end

Top-down, depth first grounding algorithm

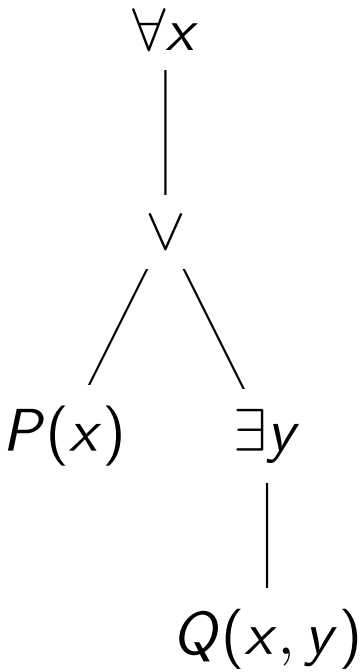
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

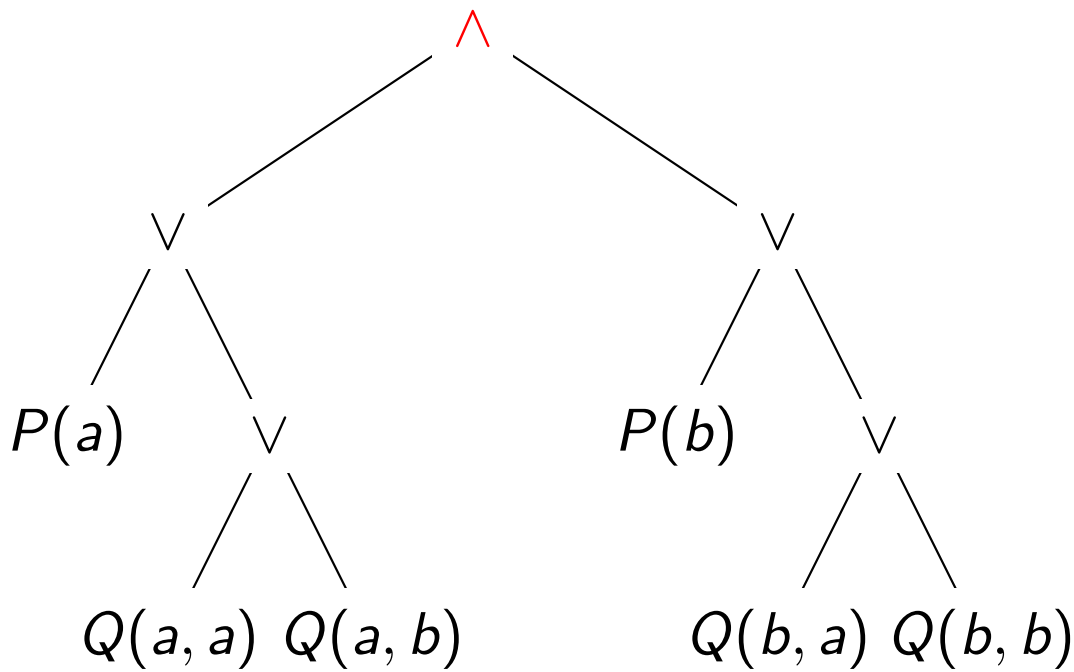




Top-down, depth first grounding algorithm

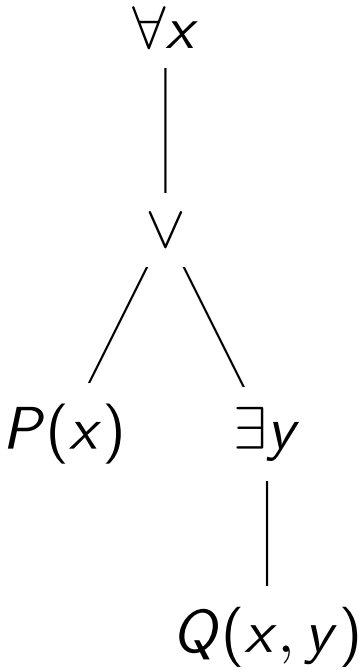
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

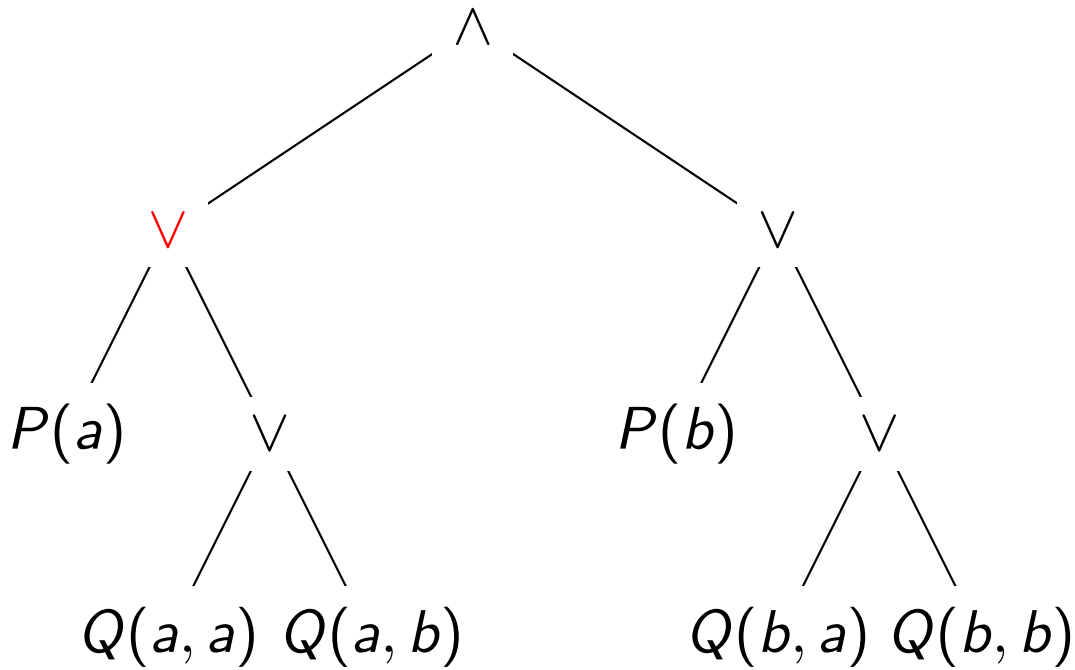




Top-down, depth first grounding algorithm

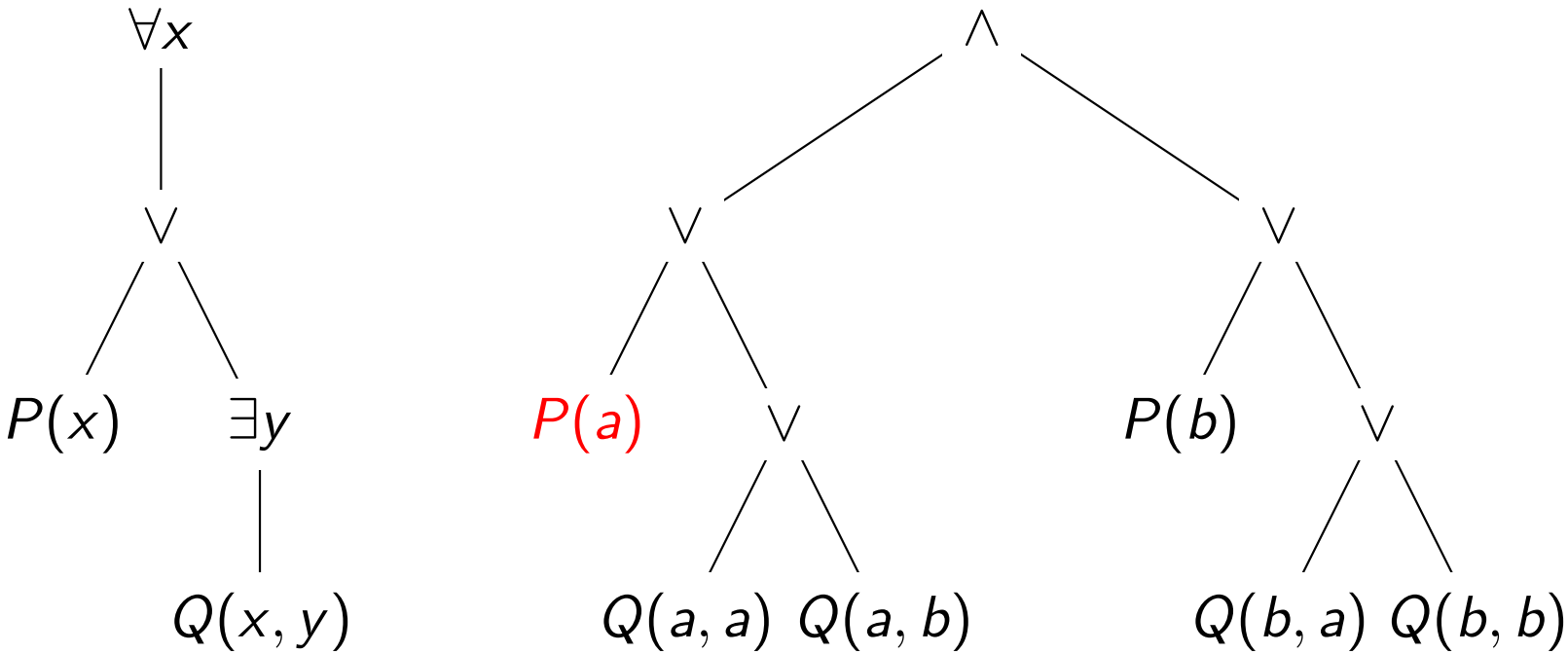
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:





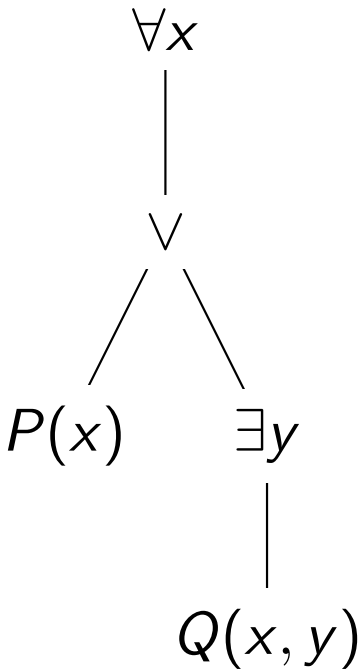
Top-down, depth first grounding algorithm

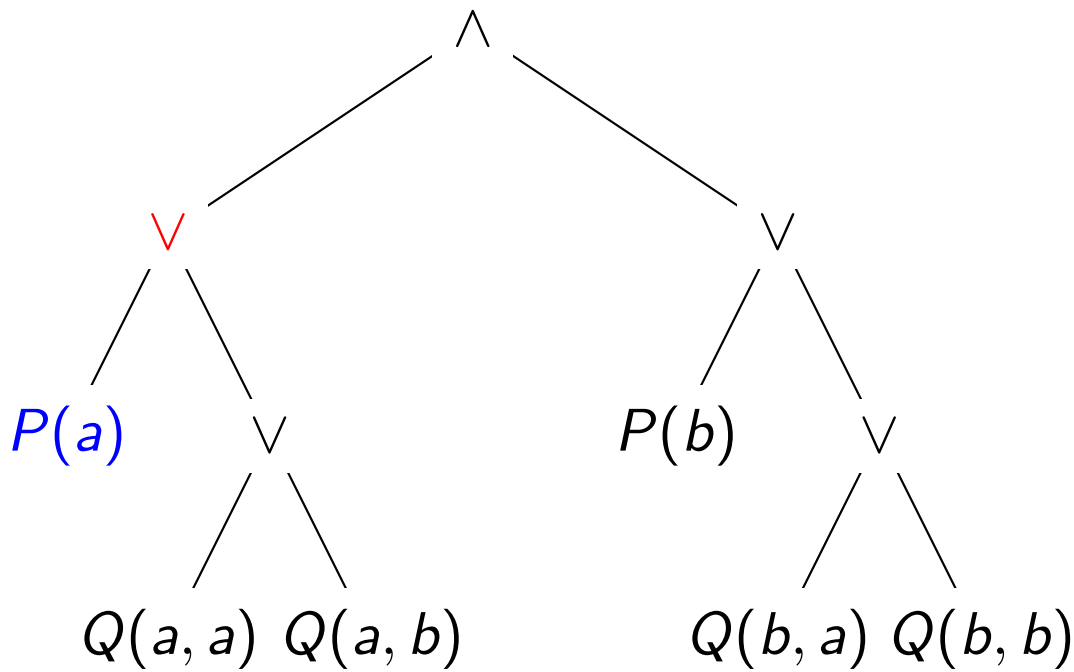
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:



Top-down, depth first grounding algorithm

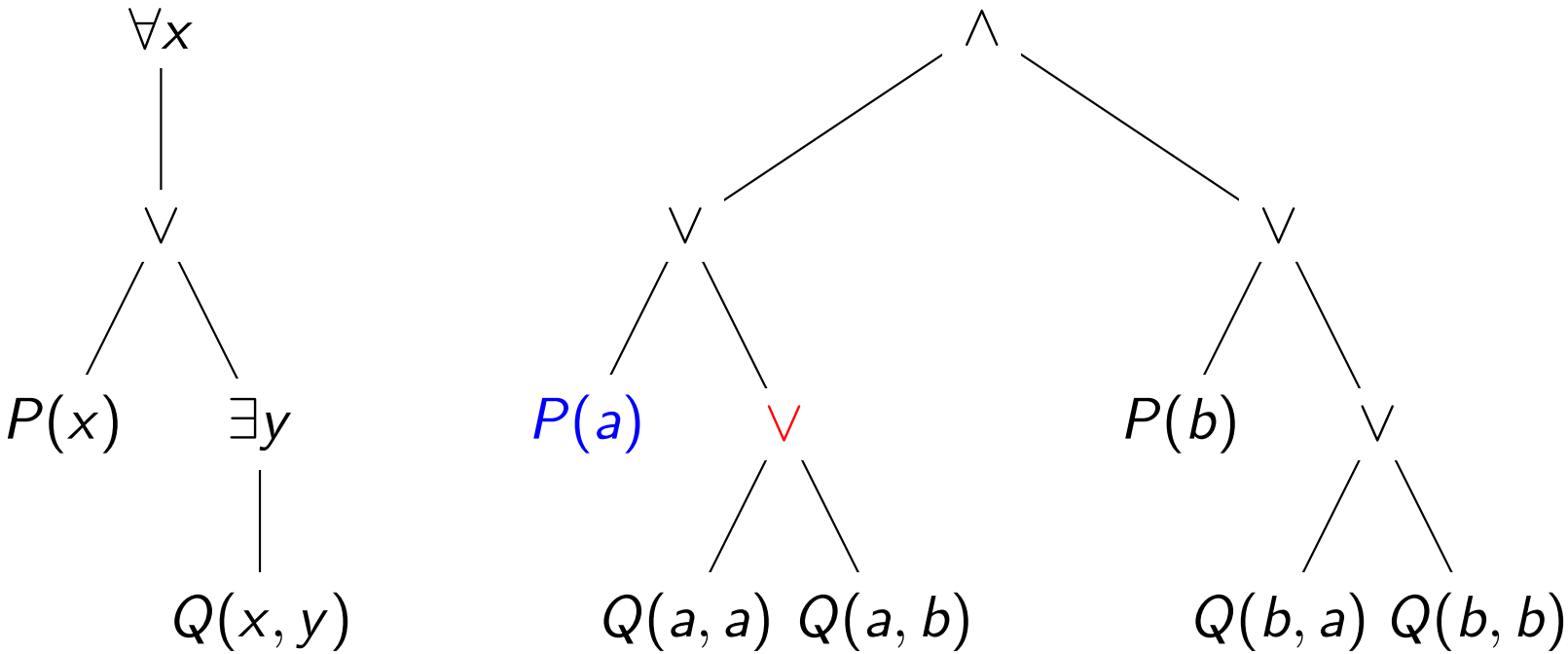
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:





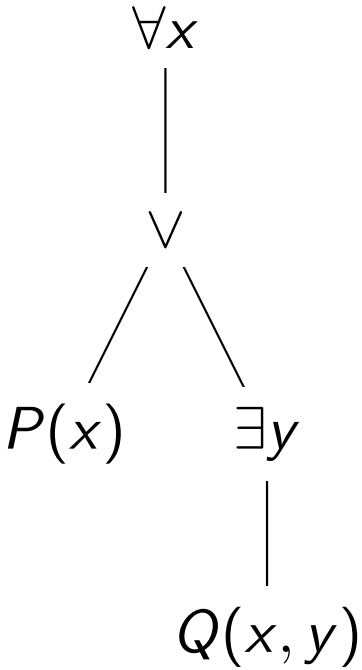
Top-down, depth first grounding algorithm

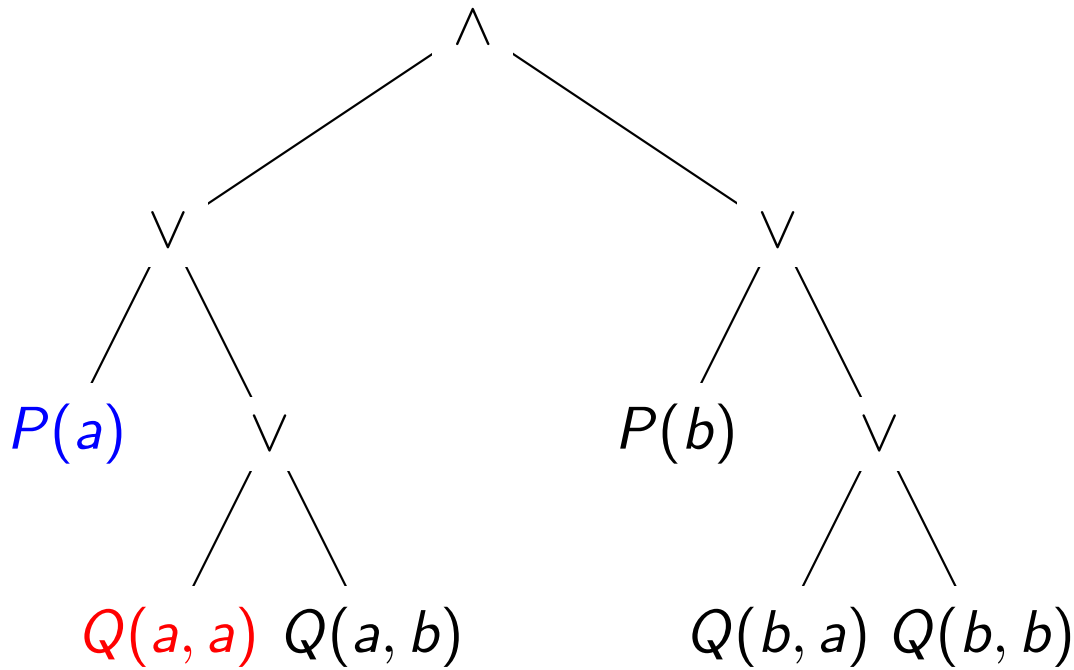
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:



Top-down, depth first grounding algorithm

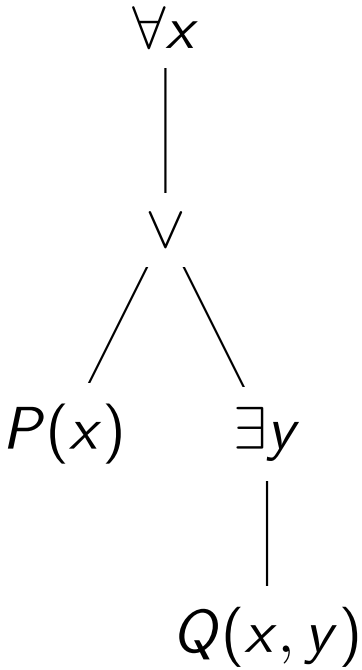
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

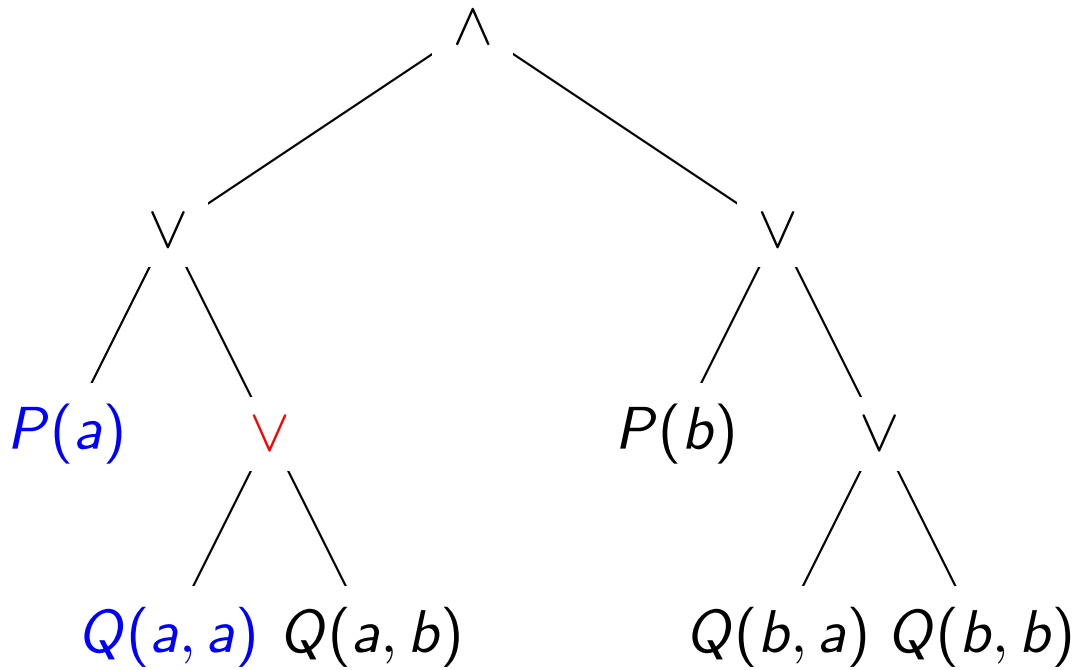




Top-down, depth first grounding algorithm

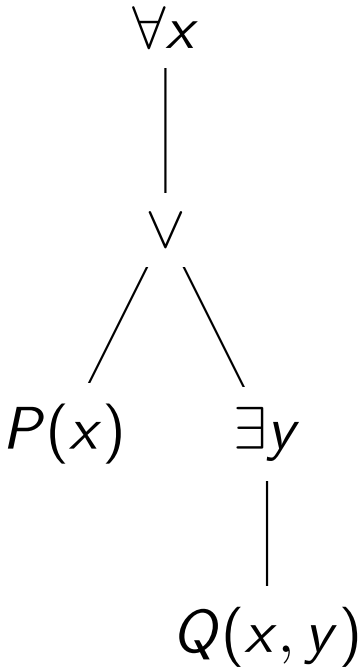
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

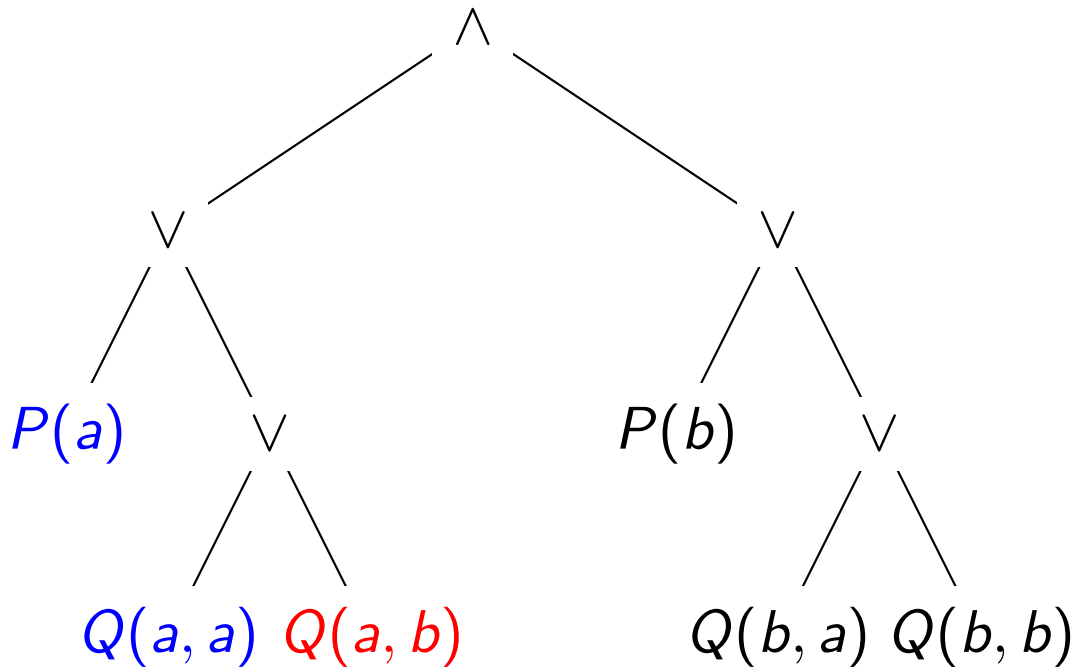




Top-down, depth first grounding algorithm

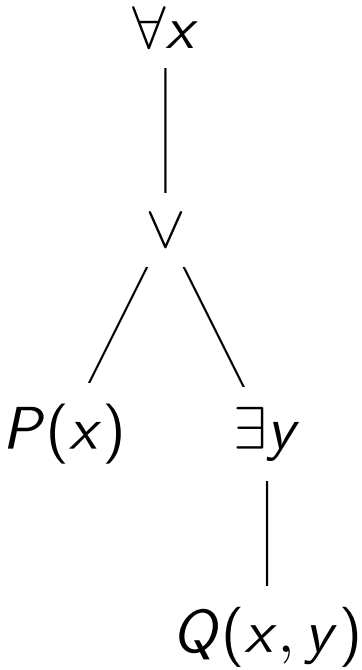
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

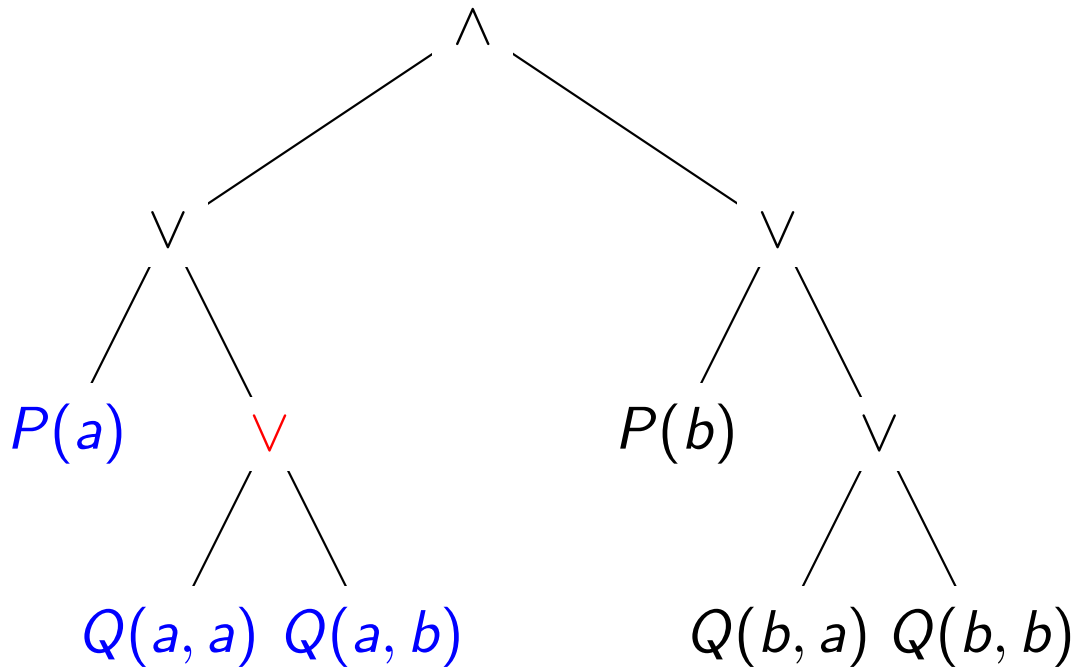




Top-down, depth first grounding algorithm

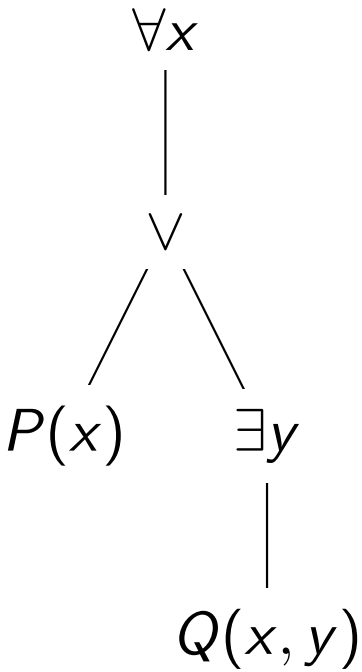
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

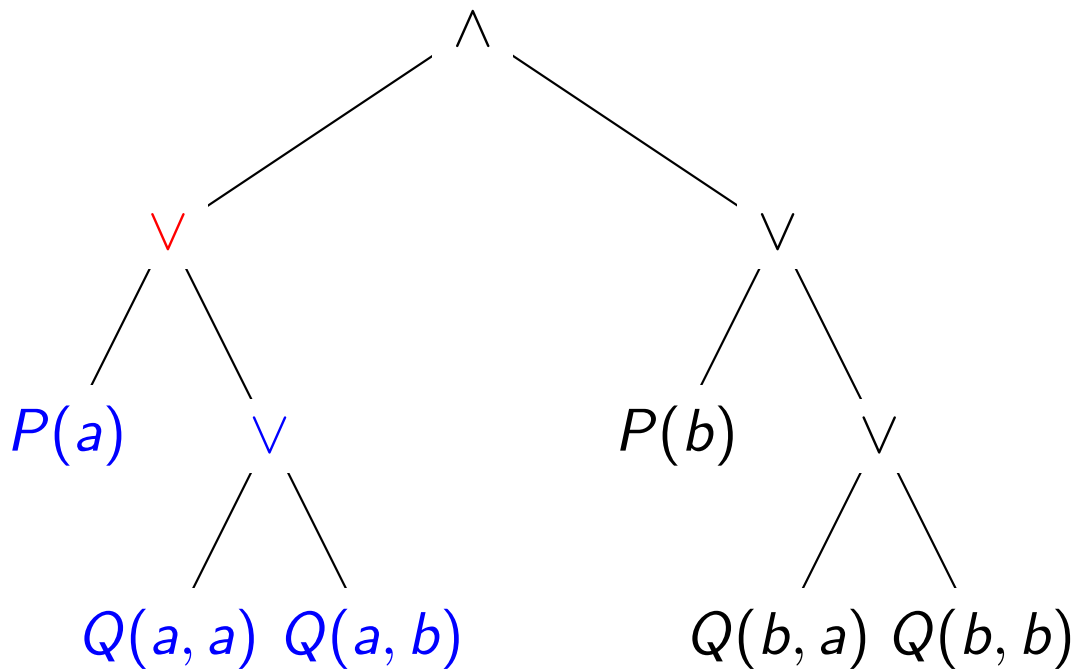




Top-down, depth first grounding algorithm

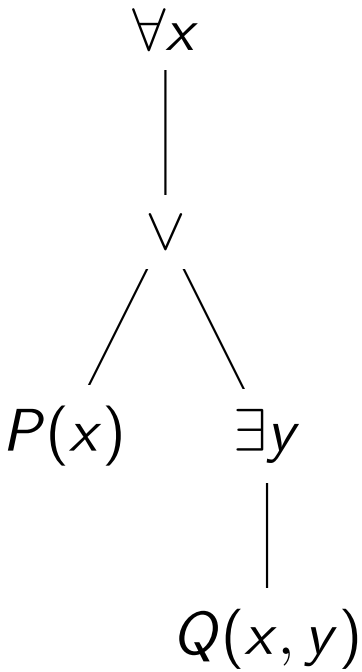
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

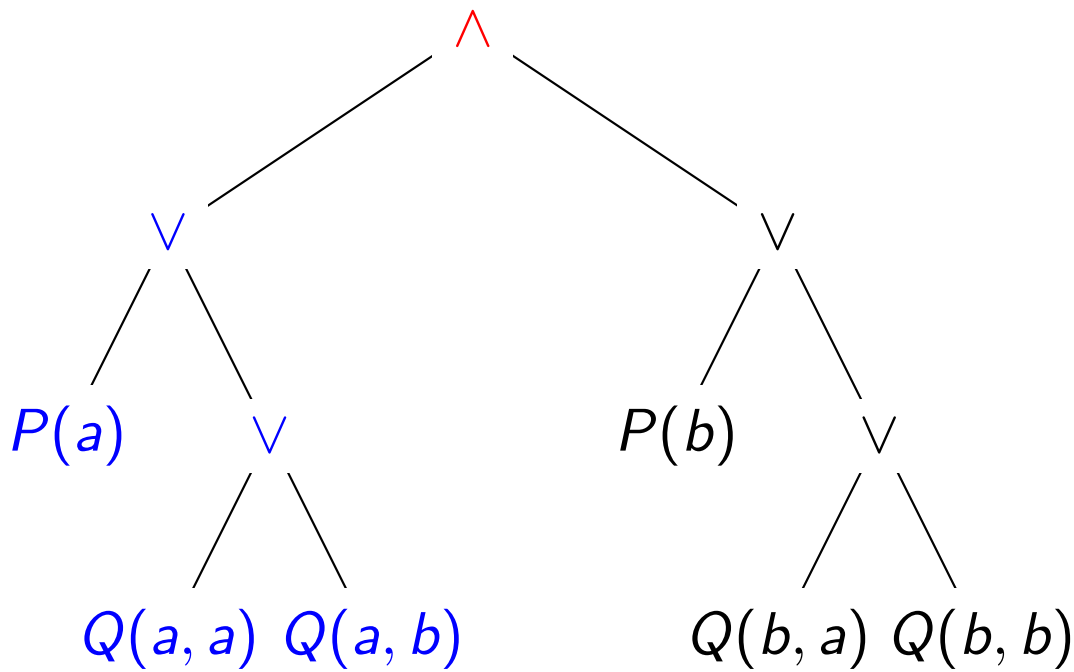




Top-down, depth first grounding algorithm

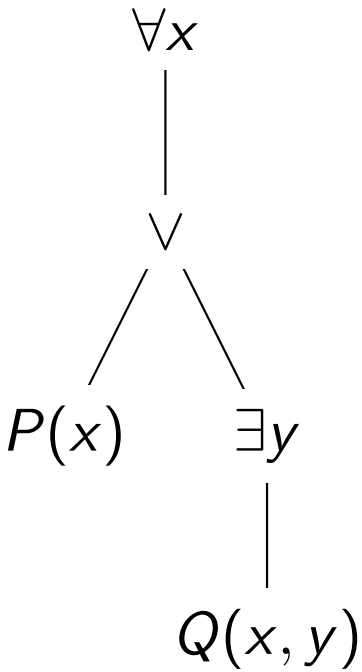
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

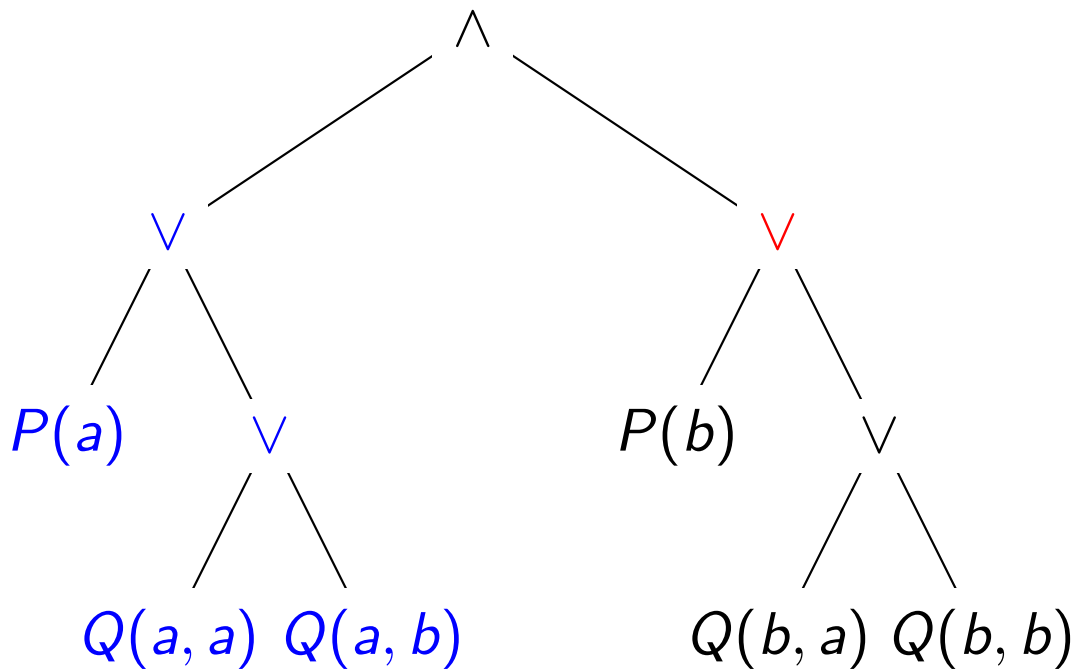




Top-down, depth first grounding algorithm

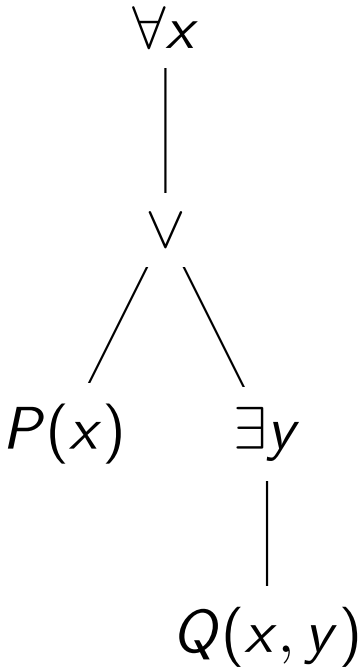
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

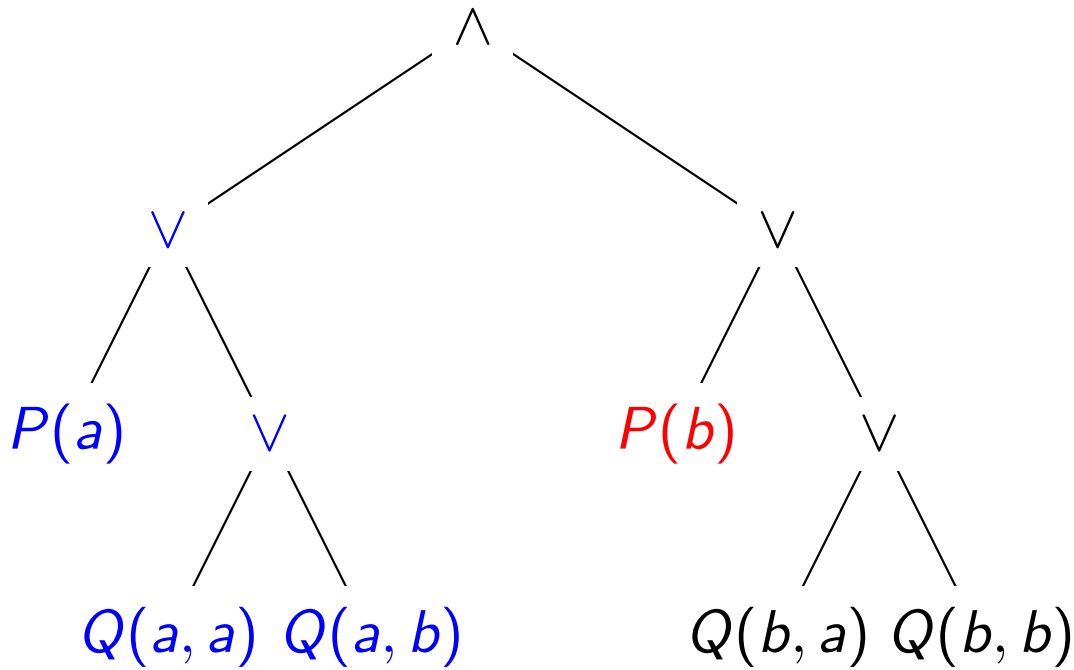




Top-down, depth first grounding algorithm

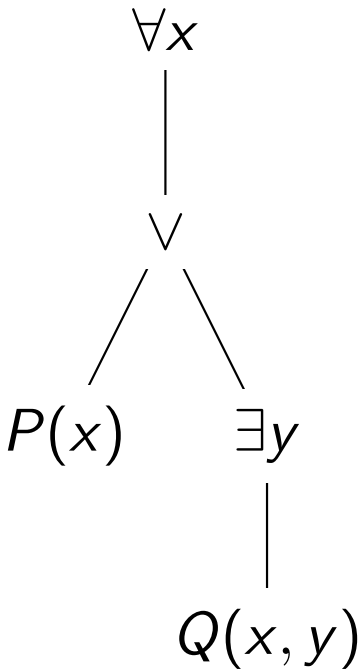
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:

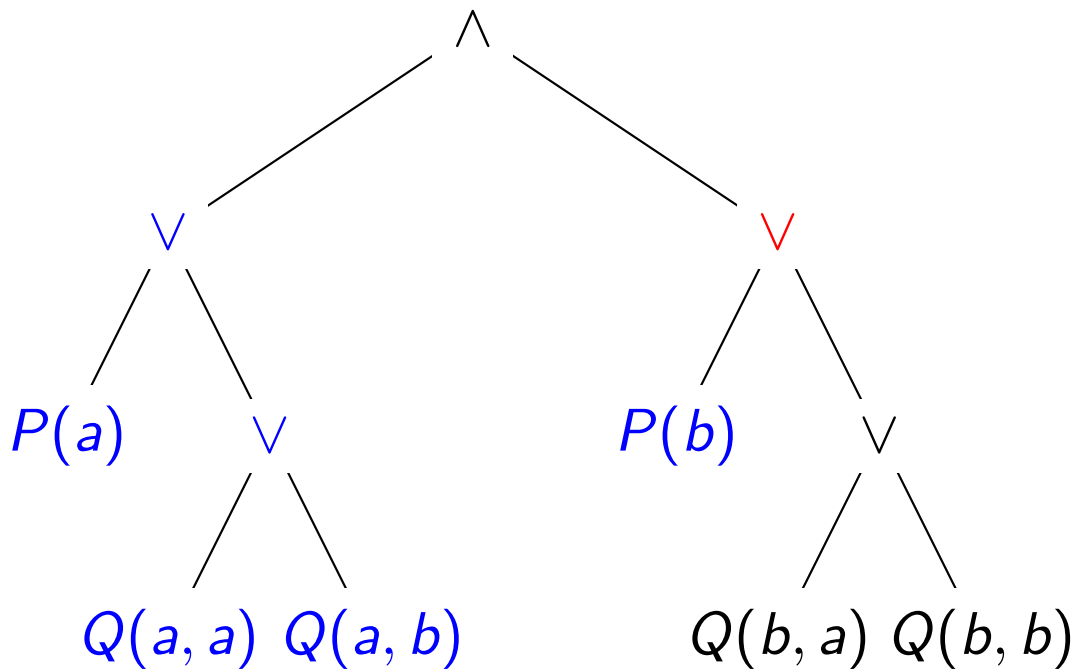




Top-down, depth first grounding algorithm

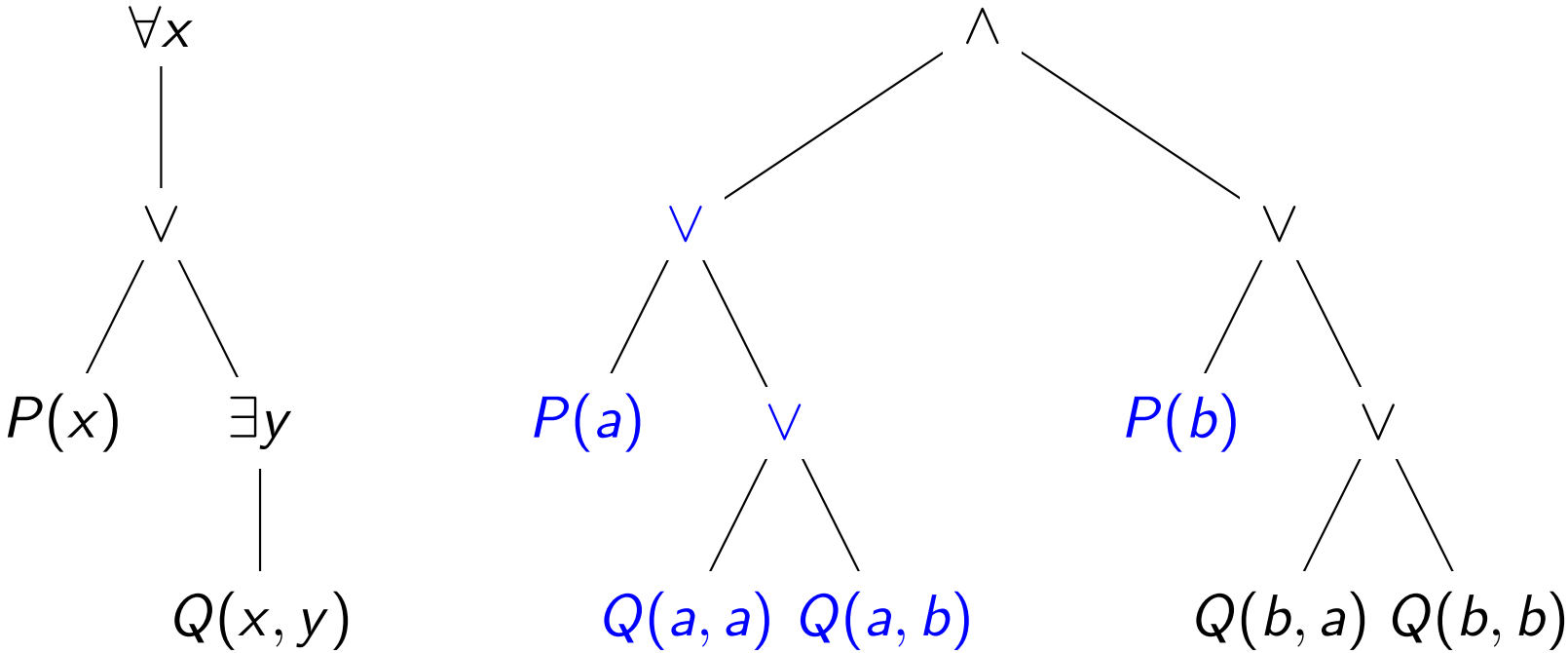
Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:





Top-down, depth first grounding algorithm

Grounding of $\forall x\ (P(x) \vee \exists y\ Q(x, y))$ over domain $\{a, b\}$:



Other algorithm: bottom-up (Vancouver style grounding)

Reduced grounding algorithm

case φ *is a ground atom* a
  if $I(a) = \mathit{true}$ **then return** $\top$;
  else if $I(a) = \mathit{false}$ **then return** $\bot$;
  else return a;
end
case $\varphi = \psi_1 \vee \ldots \vee \psi_n$
  $\mathcal{D} := \emptyset$;
  for $i = 1 \ldots n$ **do**
    $G := \mathtt{ground}(\psi_i)$;
    if $G = \top$ **then return** $\top$;
    else if $G \neq \bot$ **then** $\mathcal{D} := \mathcal{D} \cup \mathtt{ground}(\psi_i)$;
  end
  return $\bigvee \mathcal{D}$;
end
$\vdots$

Grounding definitions

To ground $\forall x \forall y\ (P(x, y) \leftarrow \varphi[y])$:

$\Delta := \emptyset$;
for $d \in \mathrm{dom}(I)$ **do**
    $\varphi_g := \texttt{ground}(\varphi[d], I)$;
    if $\varphi_g \neq \bot$ **then**
        for $d' \in \mathrm{dom}(I)$ **do**
            Add $P(d', d) \leftarrow \varphi_g$ to Δ;
        end
    end
end
return Δ;

Optimization (not yet implemented): Ground $P(x, y) \leftarrow \varphi[y]$ to

$$
\begin{aligned}
ts &\leftarrow \varphi[d_y] \\
P(d_{x1} &\leftarrow ts \\
&\vdots \\
P(d_{xn} &\leftarrow ts
\end{aligned}
$$

Grounding aggregates

To ground $\mathrm{F}\{x \mid \varphi[x] \mid t[x]\}$:
$S := \emptyset$;
$S_t := \emptyset$;
for $d \in \mathrm{dom}(I)$ **do**
    $\varphi_g := \mathtt{ground}(\varphi[d], I)$;
    if $\varphi_g \neq \bot$ **then**
        $t_g = I(t[d])$;
        if $\varphi_g = \top$ **then** Add t_g to S_t;
        else Add t_g to S;
    end
end
return S *and* $\mathrm{F}(S_t)$;

Tseitin tranformation

- SAT solvers expect grounding in CNF
- Applying distributivity on a propositional formula to get CNF leads to an exponential blow-up.
- Better technique: introduce auxiliary atoms.

$$P_1 \vee (P_2 \wedge P_3)$$
$$\downarrow$$
$$P_1 \vee Aux$$
$$Aux \Leftrightarrow P_2 \wedge P_3$$
$$\downarrow$$
$$P_1 \vee Aux$$
$$\neg Aux \vee P_2$$
$$\neg Aux \vee P_3$$
$$Aux \vee \neg P_2 \vee \neg P_3$$

Transformation becomes linear.

Improved Tseitin transformation

Instead of adding (clausal form of) $Aux \Leftrightarrow \varphi$, add

- $Aux \Rightarrow \varphi$ in a positive context (scope of an even number of negations)
- $Aux \Leftarrow \varphi$ in a negative context

Grounding is two times smaller.
One-to-many correspondence between models of original theory and grounding (instead of one-to-one).

Incorporating the Tseitin transformation

- Keep a set of introduced "Tseitin clauses" (formulas like $Aux \Leftrightarrow \varphi$)
- Replace lines like **Return** $\bigvee \mathcal{D}$ by

  if $\mathcal{D}$ *is a singleton* $\{P_a\}$ **then**
  return P_a;
  end
  else
  Create a new auxiliary predicate Aux;
  Store the Tseitin clauses corresponding to $Aux \Leftrightarrow \bigvee \mathcal{D}$;
  return Aux;
  end

- When a clause C is added to the ground theory, also (recursively) add all clauses in the store associated to tseitin atoms that occur in C.

Note: It is possible that some Tseitin atoms are created, but are never added to ground theory (because, e.g., $Aux \vee \top$ is simplified to $\top$ when creating a reduced grounding)

Unfolding and sharing of Tseitin atoms

Tseitin atoms can sometimes be removed by unfolding them
Example:

$$P_1 \vee Aux$$
$$Aux \Leftrightarrow P_2 \vee P_3$$
$$\downarrow$$
$$P_1 \vee P_2 \vee P_3$$

Not yet implemented? Not in GIDL 1 (GIDL 1 has no store, but immediately prints Tseitin clauses)
Implement by applying unfolding before the clause $P_1 \vee Aux$ is added to the grounding.

Sharing of Tseitin atoms

If two Tseitin atoms Aux_1 and Aux_2 have similar associated clauses (e.g., $Aux_1 \Rightarrow \varphi$ and $Aux_2 \Rightarrow \varphi$), Aux_2 can be replaced by Aux_1 in the whole grounding.

How to detect Tseitin atoms with similar clauses?

- Order clauses in the store and use binary search. Expensive?
- Similar clauses come from similar first-order formulas $\Rightarrow$ search similar formulas in the input theory. (Use simple syntactic check? Convert all subformulas of the input theory to BDDs and check if some created BDDs are equal?)
- Easy case to detect: the same formula is grounded more than once.

  Example: $\forall x\ (\varphi[x] \vee \forall y\ \psi[y])$

  For each substitution of x by a $d \in \mathrm{dom}(I)$, $\forall y\ \psi[y]$ is grounded $|\mathrm{dom}(I)|$ times.
  - First time: ground and create Tseitin atom Aux.
  - Second time: immediately return Aux.

  Not yet implemented. Implemented in GIDL 1

Papers: [tacas/TorlakJ07] [Exploiting subformula sharing in automatic analysis of quantified formulas, SAT 2003]

Clause splitting

- $\forall x \forall y \forall z\ (P(x,y) \vee Q(y,z))$ $\longrightarrow$ quantifier rank 3.
- Equivalent theory:
  $\forall x \forall y\ (P(x,y) \vee S(y))$
  $\forall y \forall z\ (\neg S(y) \vee Q(y,z))$
  $\longrightarrow$ quantifier rank 2, hence smaller grounding!
- Same effect is obtained by our grounding algorithm if quantifiers are moved (and Tseitin atoms are shared using the easy case of the previous slide):
  $\forall x \forall y\ (P(x,y) \vee \forall z\ Q(y,z))$
- Which quantifiers to move inside? See paper on Paradox [model/ClaessenS03]

Not yet implemented

Partition based reasoning

Partition based reasoning:

- Leads to smaller groundings
- Some models are lost, but the grounding is satisfiable iff the original theory/structure was.
- If the grounding has a model, then this model can easily be translated to a model of the original theory.
- Interesting is only one model needs to be computed or only deciding satisfiability is important.

Paper: aaai/RamachandranA05

Propagation

If input structure becomes more precise, then grounding becomes smaller.
$\Rightarrow$ Use any propositional propagation algorithm on partially constructed grounding to find atoms that are certainly true/false. This information can be added to the input structure.

- Possible propagation methods: unit propagation, one-atom lookahead.
- Used in PSGRND to simplify the grounding afterwards (and timely detect unsatisfiability), but not to make the input structure more precise.
- Difficulties:
  - Order of grounding matters a lot. How to find a good order is not known.
  - Tables used to reduce the grounding need to be updated dynamically.

Calculating known definitions

If all open symbols in a definition are two-valued in the input structure, then the interpretation of the defined symbols is fixed in all models of the grounding.

$\Rightarrow$ Compute this interpretation in advance, and make the input structure more precise.

Algorithms:

- Ground and solve. Probably the fastest method when the interpretation of the defined atoms is of the same order of magnitude as the grounding of the definition (e.g.: $\{Even(0) \leftarrow \top. Even(x+1) \leftarrow \neg Even(x).\}$.
  Currently used in GIDL.
- Semi-naive evaluation (see [aw/AbiteboulHV95]).
  Implemented in GIDL 1
- Translate to Prolog and use tabled logic programming engine.
  IMPORTANT: This creates the possibility to compute the interpretation on demand: only calculate the interpretation of defined atoms that occur in the grounding of the rest of the input theory.

Symmetry breaking

- Read master thesis of Jo Devriendt. Breaks symmetries that arise from interchangeable domain elements. May improve the overall solving time, but (the static variant) slows down grounding.
- Other kind of symmetry breaking:
  $\forall x \forall y\ (P(x) \wedge P(y) \Rightarrow x = y)$. Reduced grounding over a structure with domain $\{1, 2\}$:

$$\begin{array}{ll} \neg P(1) \vee \neg P(2). & \neg P(1) \vee \neg P(3). \\ \neg P(2) \vee \neg P(1). & \neg P(2) \vee \neg P(3). \\ \neg P(3) \vee \neg P(1). & \neg P(3) \vee \neg P(2). \end{array}$$

  Can be factor two smaller by grounding
  $\forall x \forall y\ (P(x) \wedge P(y) \wedge x \leq y \Rightarrow x = y)$ instead.
  Detection:
  - Search for formulas of the form $\forall x \forall y\ \varphi[x, y]$ or $\exists x \exists y\ \varphi[x, y]$.
  - Check if $\varphi[x, y]$ and $\varphi[y, x]$ are equivalent (e.g., using BDDs).
  - If they are, replace the formula by $\forall x \forall y\ x \leq y \Rightarrow \varphi[x, y]$, respectively $\exists x \exists y\ x \leq y \wedge \varphi[x, y]$.

Grounding terms

- Default strategy to ground $P(t)$:
  1. transform to $\forall x\ (t = x \Rightarrow P(x))$.
  2. replace equations of the form $F(y) = z$ by $P_F(y, z)$, where P_F is a new predicate.
  3. add constraint $\forall y\ \exists! z\ P_F(y, z)$.
  4. repeat until no functions are left, then apply standard grounding algorithm
- If all functions in t are two-valued in the input structure, $P(t)$ can be grounded by grounding $P(t^I)$. That is, no transformation is needed.
- If P is a supported CP constraint, $P(F(d))$ not transformed, but grounded as follows:
  1. Create a new CP variable v
  2. Adapt tr such that $\mathrm{tr}(v) = F(d)$
  3. Add $\mathrm{dom}(I)$ as domain for v
  4. Return $P(v)$

Grounding CP tricks

Introducing CP variables is similar to introducing Tseitin atoms

- Do not immediately add the domains of CP variables to the grounding, but store them. Only add when the CP variable occurs in the grounding.
- Unfolding is possible. Assume constraint $v_1 + \cdots + v_n \leq m$ is supported

$$
\begin{gathered}
\mathrm{tr}(aux_2) = v_1 + v_2 \\
\mathrm{tr}(aux_3) = aux_2 + v_3 \\
\vdots \\
\mathrm{tr}(aux_n) = aux_{n-1} + v_n \\
aux_n \leq m \\
\downarrow \\
v_1 + \cdots + v_n \leq m
\end{gathered}
$$

Binary encoding of functions

Ask Broes about this idea . . .

Basic terminology and results

Top-down, depth-first grounding

Grounding with bounds

Approximation

BDDs

Grounding with bounds

- Basic idea: instead of instantiating and simplifying afterwards, do not create unnecessary instances.
- Example: $\forall x\ P(x) \vee Q(x)$ and P two-valued in I
  $\longrightarrow$ Only instantiate x by domain elements d such that $P^I(d) = \text{false}$.
- Kind of bottom-up run before starting top-down grounding!

Code

- Old code:

case $\varphi = \forall x\ \psi[x]$
  $\mathcal{C} := \emptyset$;
  for $d \in \mathrm{dom}(I)$ **do**
    $\psi_g :=$ `ground`$(\psi[d], I)$;
    if $\psi_g \neq \top$ **then**
      if $\psi_g = \bot$ **then** **return** $\bot$;
      else $\mathcal{C} := \mathcal{C} \cup$ `ground`$(\psi[d], I)$;
    end
  end
  return *Tseitin atom associated to* $\bigwedge \mathcal{C}$;
end

- New code

case $\varphi = \forall x\ \psi[x]$
  $\mathcal{C} := \emptyset$;
  for $d \in \mathrm{dom}(I)$ *such that* $\psi[d]$ *is possibly false* **do**
    if $\psi[d]$ *is certainly false* **then** **return** $\bot$;
    else
      `/* for-loop body of old code goes here */`
    end
  end
  return *Tseitin atom associated to* $\bigwedge \mathcal{C}$;
end

- Tests in red should be fast and can be incomplete
- Similarly for existential quantifier

Create the tests

- Create two predicates P_{ct} and P_{cf} for each $P \in \Sigma$.
- Create two-valued structure J from I such that
  - $P^J_{ct}(d) = \mathit{true}$ iff $P^I(d) = \mathit{true}$
  - $P^J_{cf}(d) = \mathit{true}$ iff $P^I(d) = \mathit{false}$
- Define $\varphi_{ct} =$
  - $P_{ct}(x)$ if $\varphi = P(x)$
  - ψ_{cf} if $\varphi = \neg\psi$
  - $\psi_{ct} \wedge \chi_{cf}$ if $\varphi = \psi \wedge \chi$
  - $\psi_{ct} \vee \chi_{ct}$ if $\varphi = \psi \vee \chi$
  - $\forall x\ \psi_{ct}$ if $\varphi = \forall x\ \psi$
  - $\exists x\ \psi_{ct}$ if $\exists = \forall x\ \psi$

  Similarly for φ_{cf}. Define $\varphi_{pt} = \neg(\varphi_{cf})$, $\varphi_{pf} = \neg(\varphi_{ct})$.
- Implement test "$d \in \mathrm{dom}(I)$ such that $\psi[d]$ is possibly false" by solving query $\{x \mid \psi_{pf}[x]\}$ over J.
- Similarly for other tests

Query algorithm

- Represent query as BDD, then use our BDD query algorithm (see below)
- Translate query to Prolog and use (tabled) Prolog engine
- Any other query algorithm ...

Papers of DLV about indexing, magic set transformation, backjumping, join-ordering

Notes

- Tests do not need to be exact.
  $\Rightarrow$ simplify queries so that they can be solved more efficiently, but may have more answers.
  See below for simplification algorithm for BDDs
- If the tests in all for-loops generate finite number of instances
  $\longrightarrow$ finite grounding (even if $\mathrm{dom}(I)$ is infinite!)

Main idea

- Propagation on first-order theory such that input structure becomes more precise
- Result 1: smaller grounding
- Result 2: grounding with bounds becomes more efficient

Variants

- Main algorithm: see my PhD thesis
- Result of algorithm can take many forms:
  1. Concrete refined input structure
  2. Definition Δ over vocabulary with ct and cf predicates such that interpretation of this definition in I corresponds to a refined input structure.
  3. Non-recursive FO queries for each predicate P_{ct} and P_{cf} such that evaluating the queries corresponds to a refined input structure.
- In case (2) grounding with bounds possible by evaluating queries with respect to ($\Delta \cup$ facts representing I)
- In case (3) grounding with bounds by replacing $P_{ct}(x)$ in a query by the query associated to P_{ct} (similarly for $P_{cf}(x)$).

Basic terminology and results

Top-down, depth-first grounding

Grounding with bounds

Approximation

BDDs

Binary Decision Trees (BDTs)

$\rightarrowtriangle$: if-then-else operator

That is: $\varphi \rightarrowtriangle \psi_t ; \psi_f$ is equivalent to $(\varphi \wedge \psi_t) \vee (\neg\varphi \wedge \psi_f)$

Binary Decision Trees (BDTs)

$\rightarrowtriangle$: if-then-else operator

That is: $\varphi \rightarrowtriangle \psi_t ; \psi_f$ is equivalent to $(\varphi \wedge \psi_t) \vee (\neg\varphi \wedge \psi_f)$

Definition (Kernel, BDT)

- An atom is a kernel
- If φ is a BDT and x a variable, then $(\exists x\ \varphi)$ is a kernel
- $\top$ and $\bot$ are BDTs
- If φ is a kernel and ψ_t and ψ_f are BDTs, then $\varphi \rightarrowtriangle \psi_t ; \psi_f$ are BDTs

Binary Decision Trees (BDTs)

$\rightarrowtriangle$: if-then-else operator

That is: $\varphi \rightarrowtriangle \psi_t ; \psi_f$ is equivalent to $(\varphi \wedge \psi_t) \vee (\neg\varphi \wedge \psi_f)$

Definition (Kernel, BDT)

- An atom is a kernel
- If φ is a BDT and x a variable, then $(\exists x\ \varphi)$ is a kernel
- $\top$ and $\bot$ are BDTs
- If φ is a kernel and ψ_t and ψ_f are BDTs, then $\varphi \rightarrowtriangle \psi_t ; \psi_f$ are BDTs

Every FO formula has an equivalent BDT

Binary Decision Diagrams (BDD)

- Ordered BDT: fixed order on nodes in BDT. Every branch is according to that order.
- Reduced: $\varphi \rightarrowtriangle \psi; \psi$ is always replaced by ψ
- Shared: if B_1 and B_2 are syntactically the same BDDs, they are stored at the same address in memory.

BDD = Reduced, ordered, shared BDT.
Important result: if two propositional formulas are equivalent, their BDD form is the same.
This result does (of course) not hold for first-order logic.

Further reduction of first-order BDDs

- $\exists x(\ldots P(x)\ldots \exists y\ (\ldots Q(x,y)\ldots))$ is replaced by $\exists(\ldots P(<0>)\ldots \exists\ (\ldots Q(<1>,<0>)\ldots))$ ($<0>$ and $<1>$ are called *De Bruijn indices*)
- $\exists\ (\varphi \rightarrowtriangle \psi_t;\psi_f) \rightarrowtriangle \chi_t;\chi_f$ is replaced by $\varphi \rightarrowtriangle (\exists\ \psi_t \rightarrowtriangle \chi_t;\chi_f);(\exists\ \psi_f \rightarrowtriangle \chi_t;\chi_f)$ if φ does not contain $<0>$
- In the fixed order on kernels, kernels containing $<0>$ are always deeper in the BDDs than kernels that do not contain $<0>$ (to be able to apply the above as much as possible).
- NOTE: $\exists\top$ cannot be reduced to $\top$ if empty domain is allowed!

  Solution? Introduce a predicate *EmptyDomain* and replace $\exists\top$ by $\mathit{EmptyDomain} \rightarrowtriangle \bot;\top$

Further reduction of first-order BDDs

- $S(x) \quad \longrightarrow \quad \top$, if S is a sort predicate and the sort of x is a subsort of S.
- $x = x \twoheadrightarrow \psi_t; \psi_f \quad \longrightarrow \quad \psi_t$
- $(\exists\, (\varphi \twoheadrightarrow \top; \bot)) \twoheadrightarrow \psi_t; \psi_f \quad \longrightarrow \quad \psi_t$
  if φ is an equation that can be solved to $< 0 >$
  (i.e., φ is equivalent to $t =< 0 >$, t is total does not contain $< 0 >$)
- $\varphi[x] \twoheadrightarrow \psi_t; \psi_f \quad \longrightarrow \quad \varphi[x] \twoheadrightarrow \psi_t[x/t]; \psi_f$
  if φ can be rewritten to $t = x$, t does not contain x
  (Note: interesting if x is index $< 0 >$!)[1]
- Use normal form for arithmetic equations:
  $(s_1 \cdot t_{11} \cdot \cdots \cdot t_{1n_1}) + \cdots + (s_m \cdot t_{m1} \cdot \cdots \cdot t_m n_m)$
  $s_i \in \mathbb{R}$, t_{ij} terms that are neither additions nor multiplications

[1] In kernel $\exists\varphi$, check that there exists a kernel ψ, solvable to $< 0 >$ that occurs in every branch of φ. Then $< 0 >$ can be replaced by its 'solution' in all other kernels of φ. Not yet implemented ...

Interpretation dependent reduction

If an input interpretation I is given, a BDD / kernel without free variables can be replaced by $\top$, respectively $\bot$ if it is true, respectively false in I.
$\Rightarrow$ According to experiments, this gave no advantage in GIDL 1.
Maybe use this to remove *EmptyDomain*?

Querying

if $\varphi = \top$ **then**
  return $[\overline{x}/\overline{d}]$ *for some tuple* $\overline{d}$ *of domain elements*;
end
else if $\varphi = \psi[\overline{v}] \twoheadrightarrow \psi_1; \bot$ **then**
  for *every tuple* $\overline{d}$ *such that* $I[\overline{v}/\overline{d}] \models \psi$ **do**
    $\theta := \texttt{query}(I, \{\overline{x} \setminus \overline{v} \mid \psi_1[\overline{v}/\overline{d}]\})$;
    if $\theta \neq$ *FAIL* **then return** $\theta[\overline{v}/\overline{d}]$
  end
end
else if $\varphi = \psi[\overline{v}] \twoheadrightarrow \bot; \psi_2$ **then**
  for *every tuple* $\overline{d}$ *such that* $I[\overline{v}/\overline{d}] \not\models \psi$ **do**
    $\theta := \texttt{query}(I, \{\overline{x} \setminus \overline{v} \mid \psi_2[\overline{v}/\overline{d}]\})$;
    if $\theta \neq$ *FAIL* **then return** $\theta[\overline{v}/\overline{d}]$;
  end
end
else if φ *is of the form* $\psi[\overline{v}] \twoheadrightarrow \psi_1; \psi_2$ **then**
  for *every tuple* $\overline{d} \in \mathrm{dom}(I)$ **do**
    if $I[\overline{v}/\overline{d}] \models \psi$ **then** $\theta := \texttt{query}(I, \{\overline{x} \setminus \overline{v} \mid \psi_1[\overline{v}/\overline{d}]\})$;
    else $\theta := \texttt{query}(I, \{\overline{x} \setminus \overline{v} \mid \psi_2[\overline{v}/\overline{d}]\})$;
    if $\theta \neq$ *FAIL* **then return** $\theta[\overline{v}/\overline{d}]$;
  end
end
return *FAIL*;

Function $\texttt{query}(I, \{\overline{x} \mid \varphi\})$

Backjumping

Backjumping algorithm for conjunctive queries described in [amai/PerriSCL07]
Note: error in paper (described algorithm does not terminate. . .)
Can backjumping be generalized to BDD querying?

Optimize query

- Algorithm:
  1. Reorder nodes in BDD representing the query
  2. For each reordering, estimate cost of query
  3. Choose the best reordering
- Reordering by swapping nodes (efficient operation on BDDs). Currently not all orderings are tried. Instead, the sifting algorithm is applied. See *Dynamic Variable Ordering fo Ordered Binary Decision Diagrams (Richard Rudell)*

Estimators

Given structure I and query $Q = \{\overline{x} \mid \varphi\}$ estimate the cost of evaluating Q in I.

- Chance that an arbitrary $\overline{d}$ is answer to BDD φ:
  - $\varphi = \{\overline{x} \mid \bot\} \longrightarrow 0$
  - $\varphi = \{\overline{x} \mid \top\} \longrightarrow 1$
  - $\varphi = \{\overline{x} \mid \psi \rightarrowtriangle \chi_t; \chi_f\}$
    $\longrightarrow$
    $\text{chance}(\psi) \cdot \text{chance}(\chi_t) + (1 - \text{chance}(\psi)) \cdot \text{chance}(\chi_f)$
- Chance that an arbitrary $\overline{d}$ is answer to kernel φ:
  - $\varphi = P(\overline{x}) \longrightarrow \frac{|P^I|}{|\text{dom}(I)|^{|\overline{x}|}}$
  - $\varphi = \exists\psi.$

```
C := 0;
for 1..N do
    curr := 1;
    for 1..|dom(I)| do
        cumul_chance := Σ_{P is a path in ψ to ⊥} chance(P);
        if cumul_chance > 0 then
            curr := curr · cumul_chance;
            Randomly choose a path P to ⊥ in ψ with right probability;
            Adapt probabilities of paths in ψ;
        end
        else curr := 0; break;
    end
    C := C + curr;
end
return 1 - C/N
```

Estimators

Estimated number of answers = chance · max_nr_answers
Estimated cost for computing all answers to $\{\overline{x} \mid \varphi\}$:

- $\varphi = \bot \quad \longrightarrow 1$
- $\varphi = \top \quad \longrightarrow |\mathrm{dom}(I)|^{|\overline{x}|}$
- $\varphi = \psi \twoheadrightarrow \chi_t; \bot \quad \longrightarrow \mathsf{cost}(\psi) + \mathsf{nr\_ans}(\psi) \cdot \mathsf{cost}(\chi_t)$
- $\varphi = \psi \twoheadrightarrow \bot; \chi_f \quad \longrightarrow \mathsf{cost}(\neg\psi) + \mathsf{nr\_ans}(\neg\psi) \cdot \mathsf{cost}(\chi_f)$
- $\varphi = \psi \twoheadrightarrow \chi_t; \chi_f \quad \longrightarrow$
  $|\mathrm{dom}(I)|^{|\text{free variables of } \psi|} \cdot \mathsf{cost}(\{\emptyset \mid \psi\}) + \mathsf{nr\_ans}(\psi) \cdot \mathsf{cost}(\chi_t) + \mathsf{nr\_ans}(\neg\psi) \cdot \mathsf{cost}(\chi_f)$
- $\varphi = (\neg)P(\overline{y}) \quad \longrightarrow$ specific cost estimator based on the internal representation of the interpretation for $(\neq)P$ in I
- $\varphi = (\neg)\exists\psi \quad \longrightarrow \mathsf{cost}(\psi)$

Approximating queries

- To get more/less answers at a lower cost: try to replace nodes $\bot/\top$ by $\top/\bot$ and see if the estimated cost improves.
- Three possible points to apply this:
  - During approximation
  - On the result of approximation
  - On the queries used for grounding

  Third one can be made context sensitive:

  Example:

  $\forall x$ (some formula with a huge grounding for each instance of x)

  $\longrightarrow$ high cost to avoid instances of x is acceptable!

Avoiding double tests

When grounding $\forall x\ (\ldots \forall y\ (\ldots P(x, y) \ldots))$, queries used to ground

- instantiate x,
- instantiate y,
- check if $P(x, y)$ is certainly true/false

are similar. Avoid double checks:

- if some query Q_2 is executed in a context where query Q_1 has a succesfull answer ...
- ...then simplify Q_2 with respect to Q_1 by the algorithm `simplify` described in *An Introduction to Binary Decision Diagrams (Henrik Reif Andersen)*
- Note: check if the simplified query is indeed cheaper!

Avoid infinite grounding

TODO: Adapt query algorithm handle BDD like
$(x > 3) \rightarrowtriangle ((x < 5) \rightarrowtriangle \ldots; \bot); \bot$

Lifted clause learning

Idea:

1. SAT solver learns a clause by resolution
2. Clauses that are used in resolution are instances of first-order formulas
3. Apply first-order resolution on this first-order formulas to obtain learned first-order clause
4. Ground the learned first-order clause

E-mail

johan.wittocx@cs.kuleuven.be
johan.wittocx@gmail.com