# A Circuit-Based Approach to Efficient Enumeration

**Antoine Amarilli**[1], Pierre Bourhis[2], Louis Jachiet[3], Stefan Mengel[4]

May 9th, 2017

[1]Télécom ParisTech

[2]CNRS CRIStAL

[3]Université Grenoble-Alpes

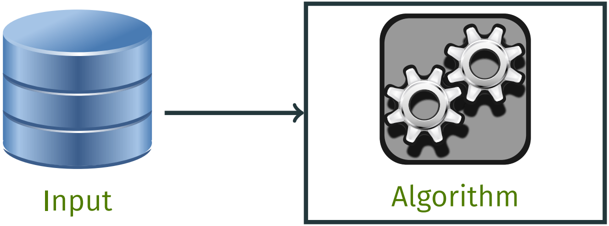
[4]CNRS CRIL

# Problem statement

# Problem: Enumerating large result sets

Input

# Problem: Enumerating large result sets



# Problem: Enumerating large result sets

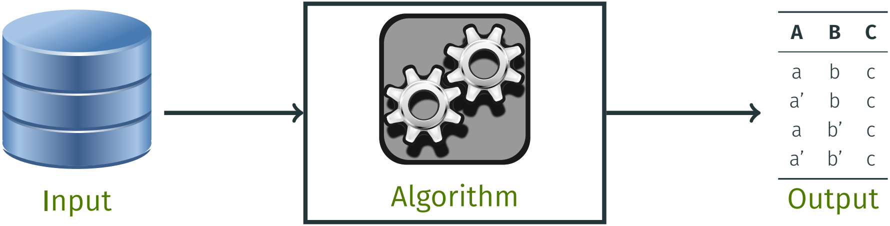
Input

Algorithm

| A | B | C |
|---|---|---|
| a | b | c |
| a’ | b | c |
| a | b’ | c |
| a’ | b’ | c |

Output

# Problem: Enumerating large result sets




- Problem: The output may be too large to compute efficiently

# Problem: Enumerating large result sets

Input

Algorithm

| A | B | C |
|---|---|---|
| a | b | c |
| a’ | b | c |
| a | b’ | c |
| a’ | b’ | c |

Output

- Problem: The output may be too large to compute efficiently

# Problem: Enumerating large result sets

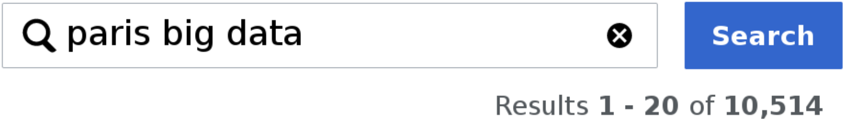
Input

Algorithm

| A | B | C |
|---|---|---|
| a | b | c |
| a' | b | c |
| a | b' | c |
| a' | b' | c |

Output

- Problem: The output may be too large to compute efficiently



# Problem: Enumerating large result sets

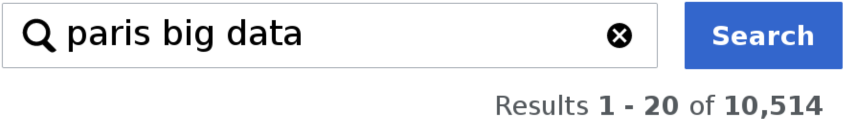
Input

Algorithm

| A | B | C |
|---|---|---|
| a | b | c |
| a’ | b | c |
| a | b’ | c |
| a’ | b’ | c |

Output

- Problem: The output may be too large to compute efficiently




...

# Problem: Enumerating large result sets

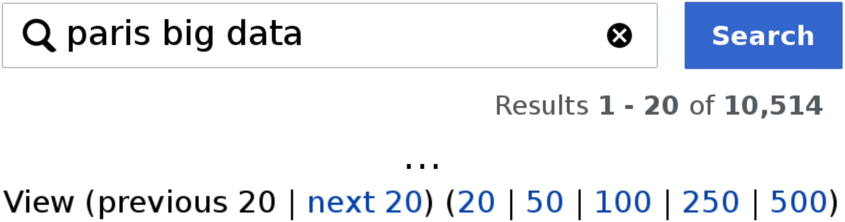
Input

Algorithm

| A | B | C |
|---|---|---|
| a | b | c |
| a’ | b | c |
| a | b’ | c |
| a’ | b’ | c |

Output

- Problem: The output may be too large to compute efficiently



# Problem: Enumerating large result sets

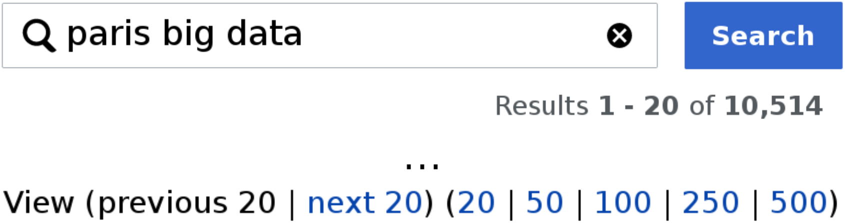
Input

Algorithm

| A | B | C |
|---|---|---|
| a | b | c |
| a’ | b | c |
| a | b’ | c |
| a’ | b’ | c |

Output

- Problem: The output may be too large to compute efficiently




→ Solution: Enumerate solutions one after the other

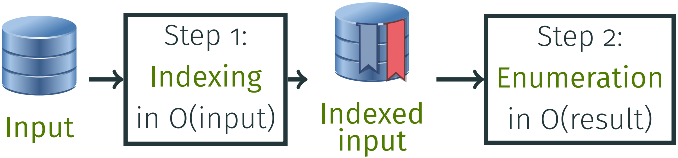
# Enumeration algorithm

Input

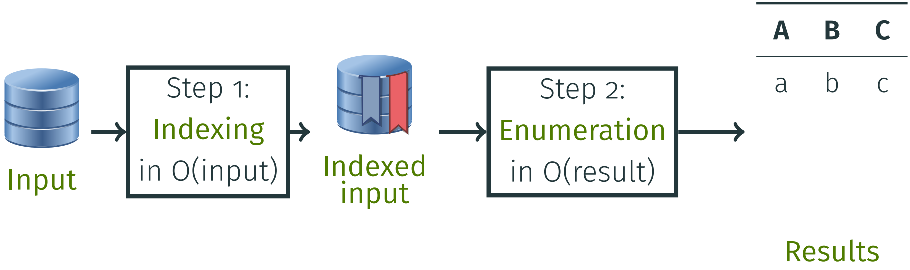
# Enumeration algorithm

# Enumeration algorithm

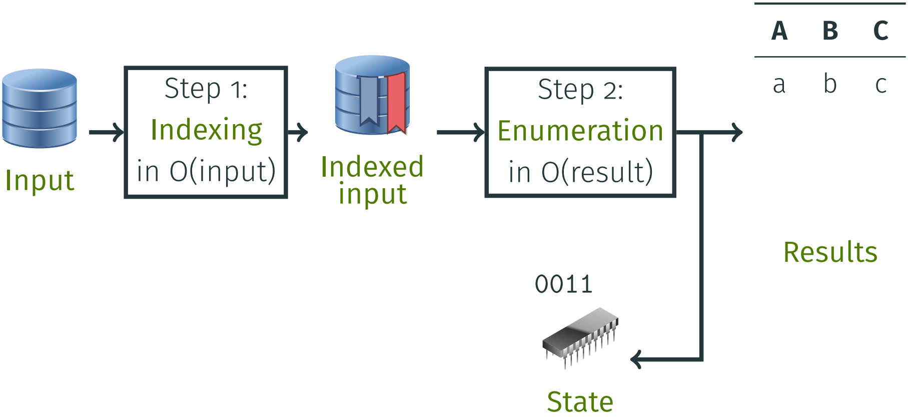
# Enumeration algorithm



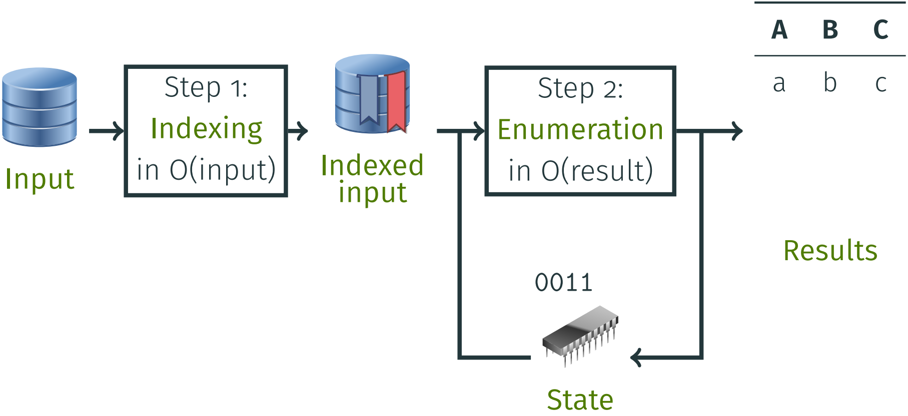
# Enumeration algorithm



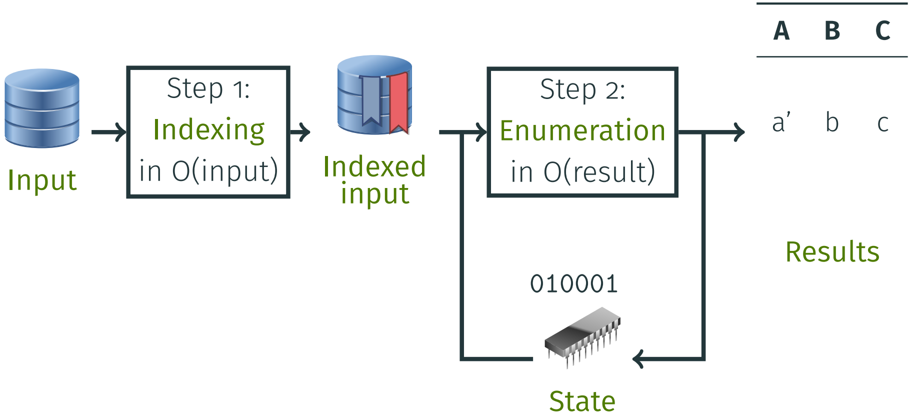
# Enumeration algorithm



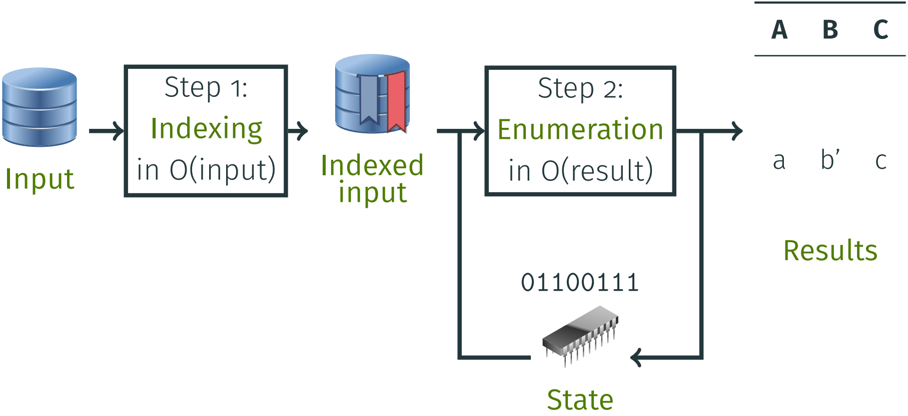
# Enumeration algorithm



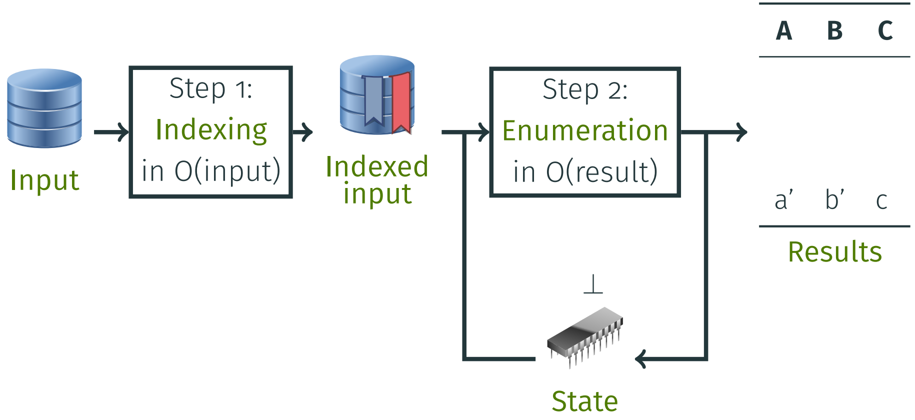
# Enumeration algorithm



# Enumeration algorithm

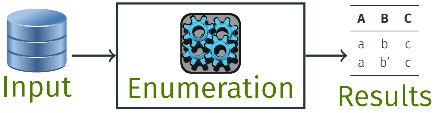


# Enumeration algorithm



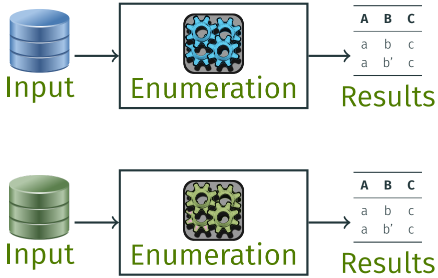
# General idea for enumeration

**Currently:**



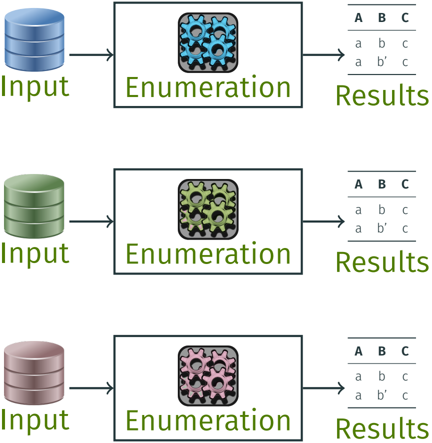
# General idea for enumeration

**Currently:**



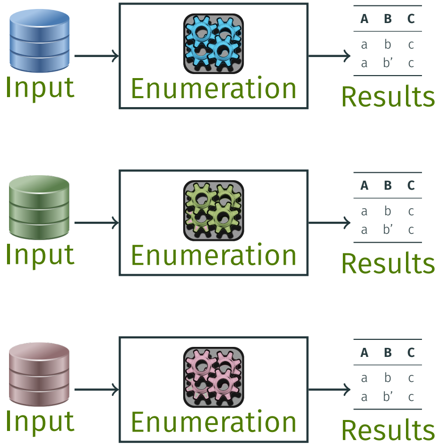
# General idea for enumeration

**Currently:**



# General idea for enumeration

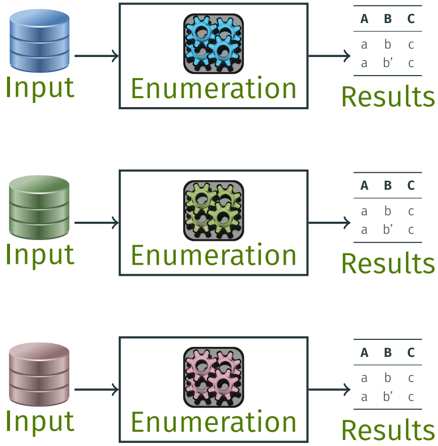
## Currently:



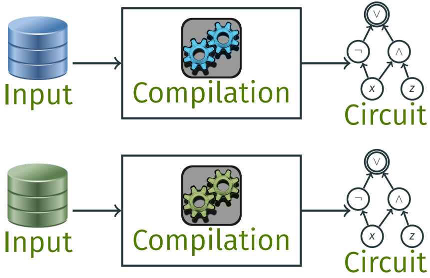

## Our idea:

# General idea for enumeration

## Currently:

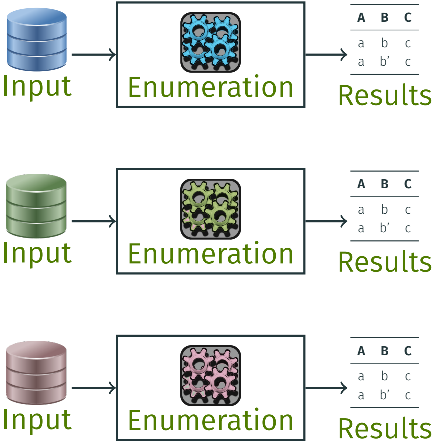



## Our idea:

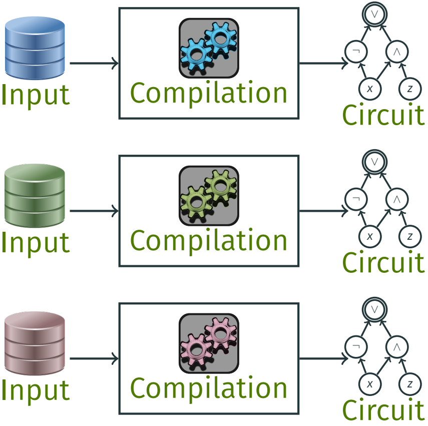


# General idea for enumeration

## Currently:

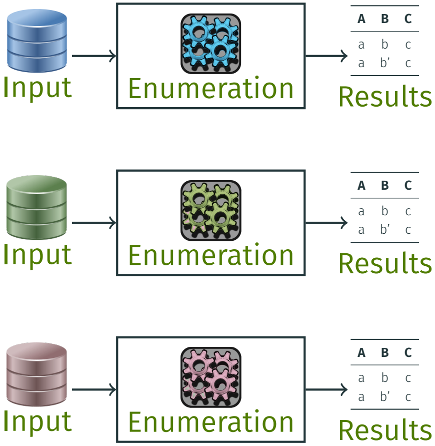



## Our idea:

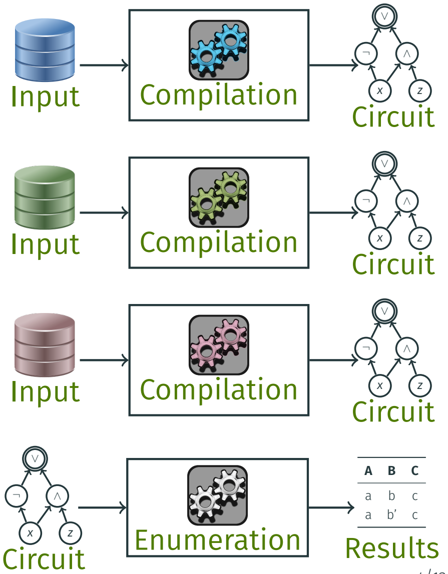


# General idea for enumeration

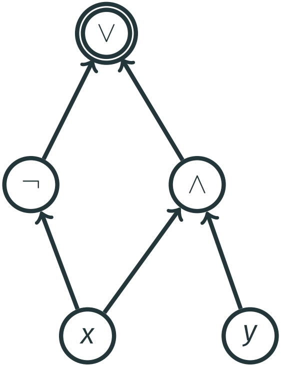
## Currently:




## Our idea:

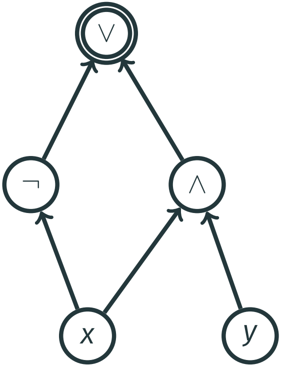


## Boolean circuits



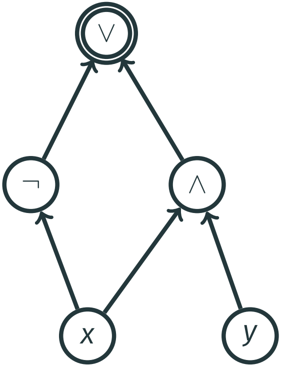

- Directed acyclic graph of gates

# Boolean circuits



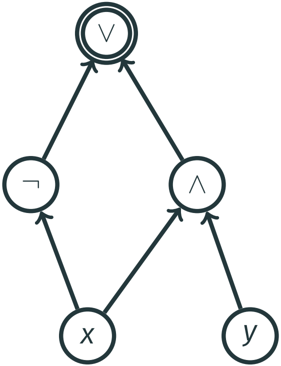

- Directed acyclic graph of gates
- Output gate: 

# Boolean circuits



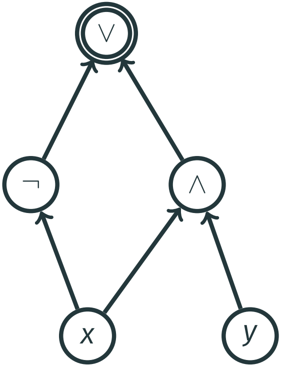

- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates: 

# Boolean circuits



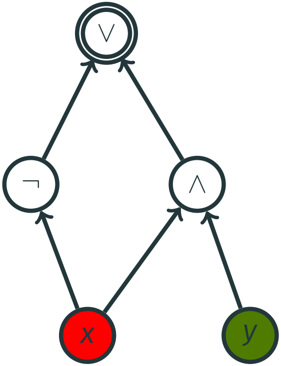

- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates: $x$
- **Internal** gates: $\lor$ $\land$ $\neg$

# Boolean circuits




- Directed acyclic graph of gates
- Output gate: 
- Variable gates: $x$
- Internal gates: $\vee$ $\wedge$ $\neg$
- Valuation: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$...

## Boolean circuits

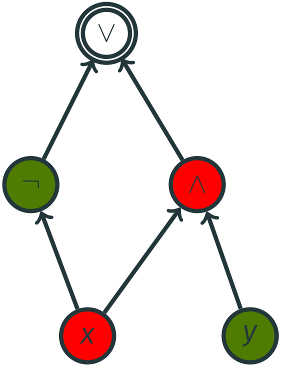



- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates: 

- **Internal** gates: 

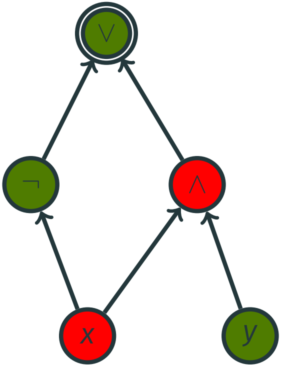
- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$...

# Boolean circuits




- Directed acyclic graph of gates
- Output gate:
- Variable gates: $x$
- Internal gates: $\vee$ $\wedge$ $\neg$
- Valuation: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$...

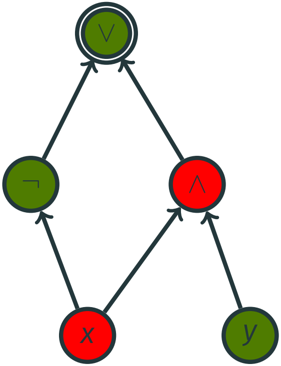
## Boolean circuits




- Directed acyclic graph of gates
- Output gate:
- Variable gates: $x$
- Internal gates: $\lor$ $\land$ $\neg$
- Valuation: function from variables to $\{0, 1\}$

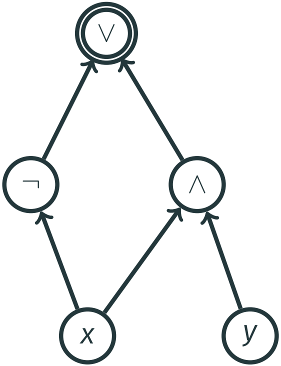
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$... mapped to 1

# Boolean circuits




- Directed acyclic graph of gates
- Output gate:
- Variable gates: $x$
- Internal gates: $\vee$ $\wedge$ $\neg$
- Valuation: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$... mapped to 1
- Assignment: set of variables mapped to 1
  Example: $S_\nu = \{y\}$; more concise than $\nu$

# Boolean circuits




- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates: 

- **Internal** gates: 

- **Valuation**: function from variables to $\{0, 1\}$
  Example: $\nu = \{x \mapsto 0,\ y \mapsto 1\}$... mapped to 1
- **Assignment**: set of variables mapped to 1
  Example: $S_\nu = \{y\}$; more concise than $\nu$

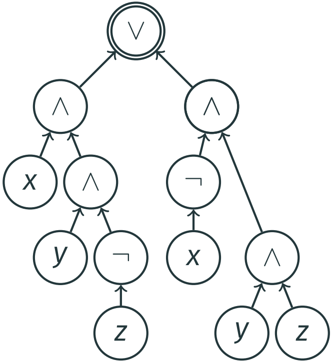
**Our task:** Enumerate all **satisfying assignments** of an input circuit

# Circuit restrictions

**d-DNNF:**

- $\vee$ are all deterministic:

The inputs are mutually exclusive (= no valuation $\nu$ makes two inputs simultaneously evaluate to 1)



# Circuit restrictions

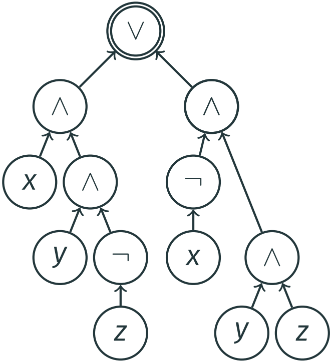
**d-DNNF:**

- $\lor$ are all deterministic:

The inputs are mutually exclusive
(= no valuation $\nu$ makes two inputs simultaneously evaluate to 1)

- $\land$ are all decomposable:

The inputs are independent
(= no variable *x* has a path to two different inputs)



# Circuit restrictions

**d-DNNF:**

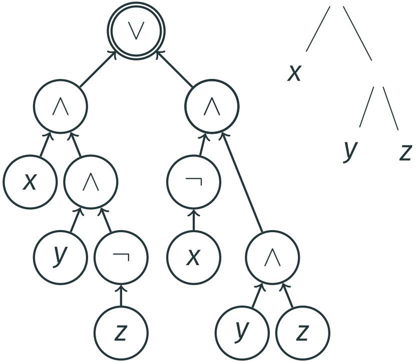
- $\vee$ are all deterministic:

The inputs are mutually exclusive (= no valuation $\nu$ makes two inputs simultaneously evaluate to 1)

- $\wedge$ are all decomposable:

The inputs are independent (= no variable $x$ has a path to two different inputs)

**v-tree:** $\wedge$-gates follow a tree on the variables



# Main results

**Theorem**

*Given a d-DNNF circuit $C$ with a v-tree $T$, we can enumerate its satisfying assignments with preprocessing linear in $|C| + |T|$ and delay linear in each assignment*

# Main results

**Theorem**

*Given a d-DNNF circuit $C$ with a v-tree $T$, we can enumerate its satisfying assignments with preprocessing linear in $|C| + |T|$ and delay linear in each assignment*

Also: restrict to assignments of constant size $k \in \mathbb{N}$ (at most $k$ variables are set to $1$):

**Theorem**

*Given a d-DNNF circuit $C$ with a v-tree $T$, we can enumerate its satisfying assignments of size $\leq k$ with preprocessing linear in $|C| + |T|$ and constant delay*

# Application 1: Factorized databases

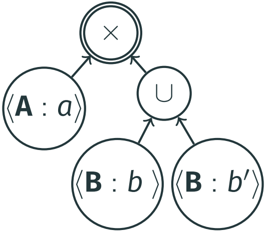
- Factorized databases: implicit representation of database tables

# Application 1: Factorized databases

- Factorized databases: implicit representation of database tables

| **A** | **B** |
|---|---|
| $a$ | $b$ |
| $a$ | $b'$ |

represented by

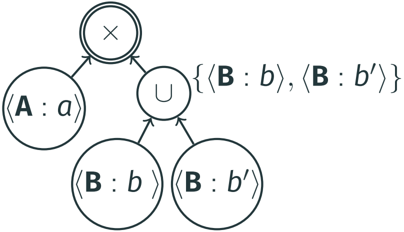


# Application 1: Factorized databases

- Factorized databases: implicit representation of database tables

| **A** | **B** |
|---|---|
| $a$ | $b$ |
| $a$ | $b'$ |

represented by

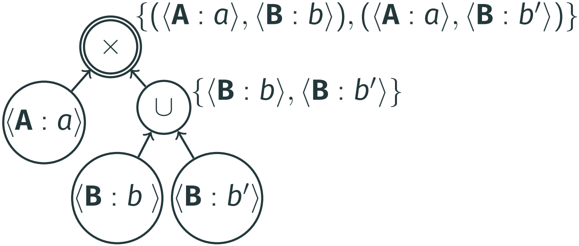


# Application 1: Factorized databases

- Factorized databases: implicit representation of database tables

| **A** | **B** |
|---|---|
| $a$ | $b$ |
| $a$ | $b'$ |

represented by

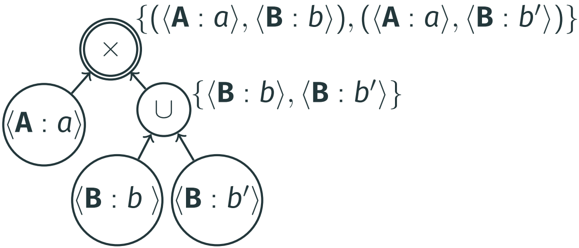


# Application 1: Factorized databases

- Factorized databases: implicit representation of database tables

| A | B |
|---|---|
| $a$ | $b$ |
| $a$ | $b'$ |

represented by



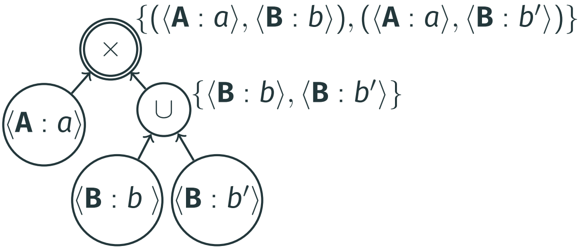

- Relational product $\times$
- Relational union $\cup$

# Application 1: Factorized databases

- **Factorized databases:** implicit representation of database tables

| **A** | **B** |
|---|---|
| $a$ | $b$ |
| $a$ | $b'$ |

represented by

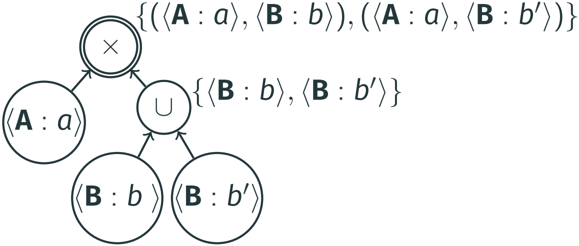



- Relational product $\times$
- Relational union $\cup$
- Deterministic: We do not obtain the same tuple multiple times

- Factorized databases: implicit representation of database tables

| **A** | **B** |
|---|---|
| $a$ | $b$ |
| $a$ | $b'$ |

represented by




- Relational product $\times$
- Relational union $\cup$
- Deterministic: We do not obtain the same tuple multiple times

**Theorem (Strenghtens result of [Olteanu and Závodnỳ, 2015])**
*Given a deterministic factorized representation, we can enumerate its tuples with linear preprocessing and constant delay*

# Application 2: Query evaluation

- Compute the results $(a, b, c)$ of a query $Q(x, y, z)$ on a database $D$

# Application 2: Query evaluation

- Compute the results $(a, b, c)$ of a query $Q(x, y, z)$ on a database $D$
- Assumption: the database has bounded treewidth
  - $\rightarrow$ Captures trees, words, etc.

# Application 2: Query evaluation

- Compute the results $(a, b, c)$ of a query $Q(x, y, z)$ on a database $D$
- Assumption: the database has bounded treewidth
  - → Captures trees, words, etc.
- Query given as a deterministic tree automaton
  - → Captures monadic second-order (data-independent translation)
  - → Captures conjunctive queries, SQL, etc.

# Application 2: Query evaluation

- Compute the results $(a, b, c)$ of a query $Q(x, y, z)$ on a database $D$
- Assumption: the database has bounded treewidth
  - $\rightarrow$ Captures trees, words, etc.
- Query given as a deterministic tree automaton
  - $\rightarrow$ Captures monadic second-order (data-independent translation)
  - $\rightarrow$ Captures conjunctive queries, SQL, etc.

$\rightarrow$ We can construct a d-DNNF that describes the query results

# Application 2: Query evaluation

- Compute the results $(a, b, c)$ of a query $Q(x, y, z)$ on a database $D$
- Assumption: the database has bounded treewidth
  $\rightarrow$ Captures trees, words, etc.
- Query given as a deterministic tree automaton
  $\rightarrow$ Captures monadic second-order (data-independent translation)
  $\rightarrow$ Captures conjunctive queries, SQL, etc.

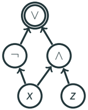
$\rightarrow$ We can construct a d-DNNF that describes the query results

**Theorem (Recaptures [Bagan, 2006], [Kazana and Segoufin, 2013])**
*Given a MSO query $Q$ and a database $D$, the results of $Q$ on $D$ can be enumerated with linear preprocessing in $D$ and linear delay in each answer ($\rightarrow$ constant delay for free first-order variables)*

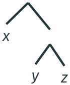
# Proof techniques

# Proof overview

## Preprocessing phase:



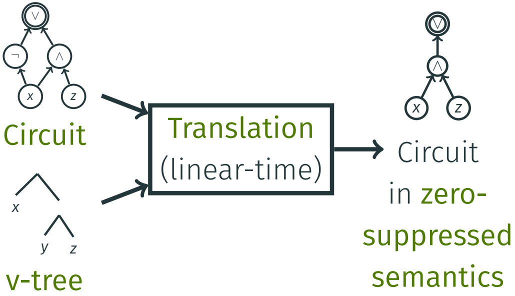

Circuit




v-tree

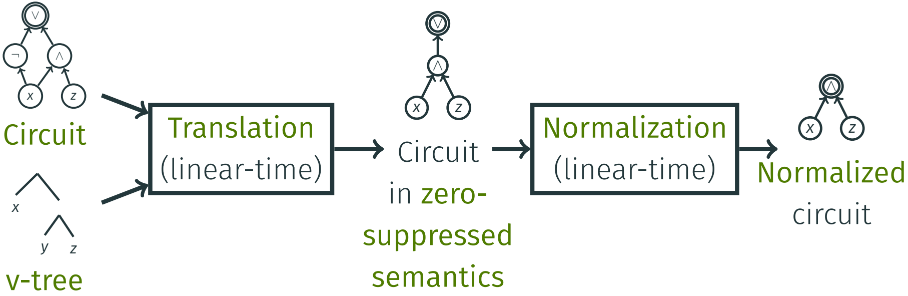
# Proof overview

## Preprocessing phase:



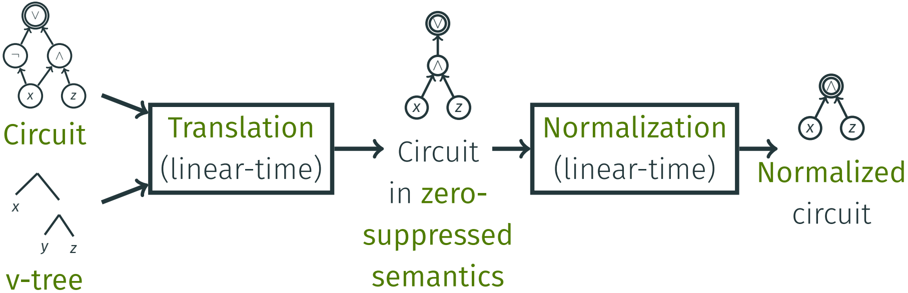
# Proof overview

## Preprocessing phase:

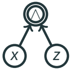


# Proof overview

## Preprocessing phase:



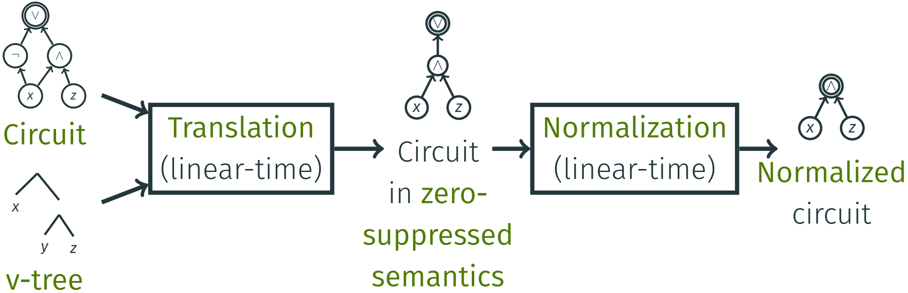

## Enumeration phase:



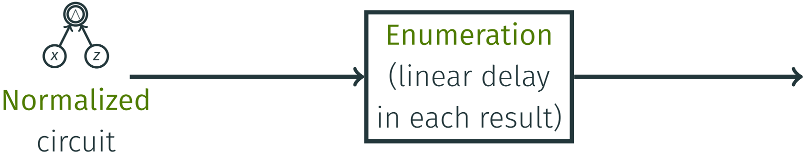

Normalized
circuit

# Proof overview

## Preprocessing phase:



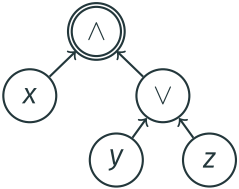

## Enumeration phase:




| A | B | C |
|---|---|---|
| a | b | c |
| a | b’ | c |

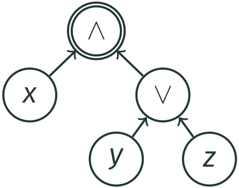
Results

# Zero-suppressed semantics




Special zero-suppressed semantics for circuits:

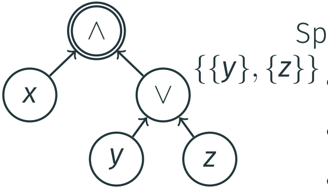
# Zero-suppressed semantics




Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

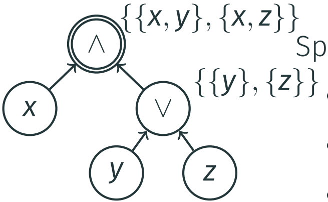
# Zero-suppressed semantics




Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

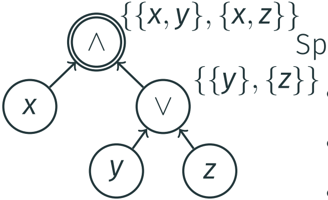
# Zero-suppressed semantics




Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

## Zero-suppressed semantics

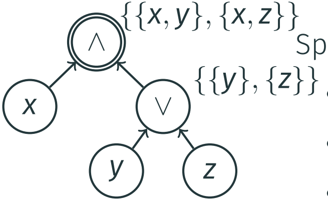



Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

- d-DNNF: $\cup$ are disjoint, $\times$ are on disjoint sets

# Zero-suppressed semantics




Special zero-suppressed semantics for circuits:

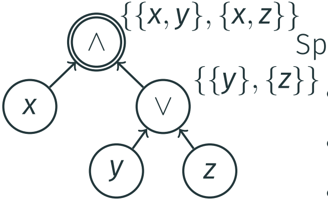
- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

- d-DNNF: $\cup$ are disjoint, $\times$ are on disjoint sets

Many equivalent ways to understand this:

- Generalization of factorized representations
- Analogue of zero-suppressed OBDDs (implicit negation)
- Arithmetic circuits: $\times$ and $+$ on polynomials

# Zero-suppressed semantics




Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

- d-DNNF: $\cup$ are disjoint, $\times$ are on disjoint sets

Many equivalent ways to understand this:

- Generalization of factorized representations
- Analogue of zero-suppressed OBDDs (implicit negation)
- Arithmetic circuits: $\times$ and $+$ on polynomials

Simplification: rewrite circuits to arity-two (fan-in $\leq 2$)

# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

Base case: variable $x$ :

# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

$\rightarrow$ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

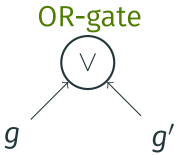
Base case: variable $x$ : enumerate $\{x\}$ and stop

# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

$\rightarrow$ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

Base case: variable ($x$) : enumerate $\{x\}$ and stop



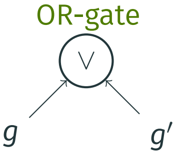

Concatenation: enumerate $S(g)$
and then enumerate $S(g')$

# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

$\rightarrow$ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

Base case: variable $x$ : enumerate $\{x\}$ and stop




Concatenation: enumerate $S(g)$
and then enumerate $S(g')$

Determinism: no duplicates

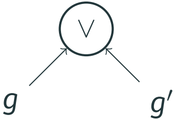
# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

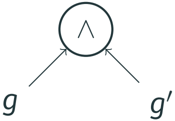
$\rightarrow$ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

Base case: variable $(x)$ : enumerate $\{x\}$ and stop

OR-gate




AND-gate




Concatenation: enumerate $S(g)$ and then enumerate $S(g')$

Determinism: no duplicates

Lexicographic product: enumerate $S(g)$ and for each result $t$ enumerate $S(g')$ and concatenate $t$ with each result

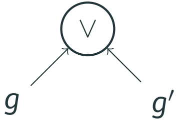
# Enumerating assignments in the zero-suppressed semantics

Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$

Base case: variable $x$ : enumerate $\{x\}$ and stop

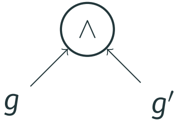
OR-gate




Concatenation: enumerate $S(g)$ and then enumerate $S(g')$

Determinism: no duplicates

AND-gate




Lexicographic product: enumerate $S(g)$ and for each result $t$ enumerate $S(g')$ and concatenate $t$ with each result

Decomposability: no duplicates

# Conclusion

## Summary and conclusion

Summary:

- Usual approach: develop enumeration algorithms by hand

# Summary and conclusion

Summary:

- Usual approach: develop enumeration algorithms by hand
- Proposed approach:

# Summary and conclusion

Summary:

- Usual approach: develop enumeration algorithms by hand
- Proposed approach:
  - Develop linear-time compilation algorithm to circuits

# Summary and conclusion

Summary:

- Usual approach: develop enumeration algorithms by hand
- Proposed approach:
  - Develop linear-time compilation algorithm to circuits
  - Use restricted circuit classes (structured d-DNNF)

## Summary and conclusion

Summary:

- Usual approach: develop enumeration algorithms by hand
- Proposed approach:
  - Develop linear-time compilation algorithm to circuits
  - Use restricted circuit classes (structured d-DNNF)
  - Develop general enumeration results on circuits

# Summary and conclusion

Summary:

- Usual approach: develop enumeration algorithms by hand
- Proposed approach:
  - Develop linear-time compilation algorithm to circuits
  - Use restricted circuit classes (structured d-DNNF)
  - Develop general enumeration results on circuits

Future work:

- Theory: handle updates on the structure
- Practice: implement the technique with automata

# Summary and conclusion

Summary:

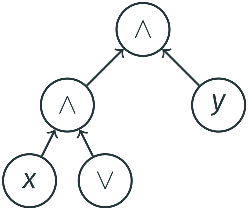
- Usual approach: develop enumeration algorithms by hand
- Proposed approach:
  - Develop linear-time compilation algorithm to circuits
  - Use restricted circuit classes (structured d-DNNF)
  - Develop general enumeration results on circuits

Future work:

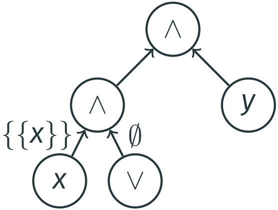
- Theory: handle updates on the structure
- Practice: implement the technique with automata

Thanks for your attention!

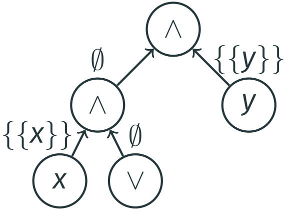
# Normalization: handling $\emptyset$



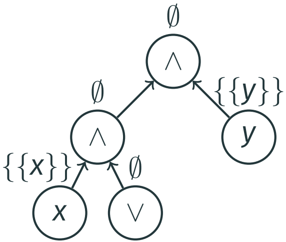
# Normalization: handling ∅



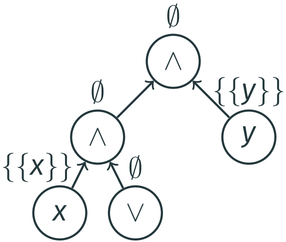
# Normalization: handling $\emptyset$



# Normalization: handling ∅

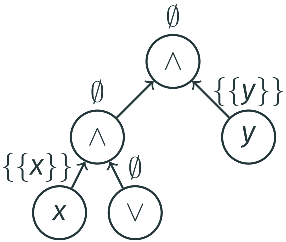


# Normalization: handling $\emptyset$



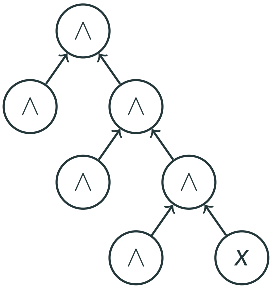

- Problem: if $S(g) = \emptyset$ we waste time

# Normalization: handling $\emptyset$

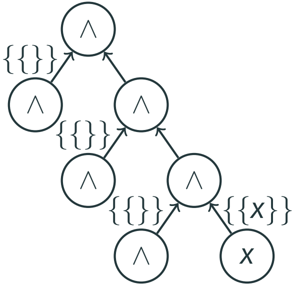



- Problem: if $S(g) = \emptyset$ we waste time
- Solution: compute bottom-up if $S(g) = \emptyset$

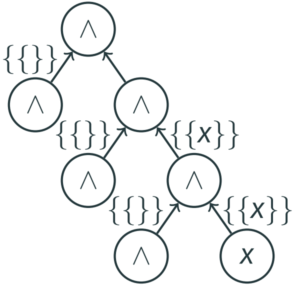
# Normalization: handling empty assignments



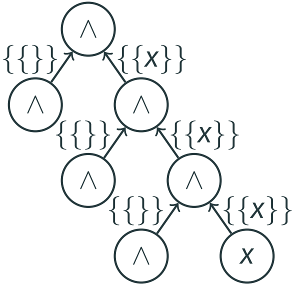
# Normalization: handling empty assignments



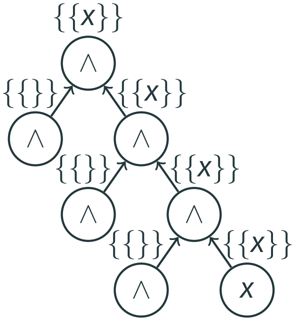
# Normalization: handling empty assignments



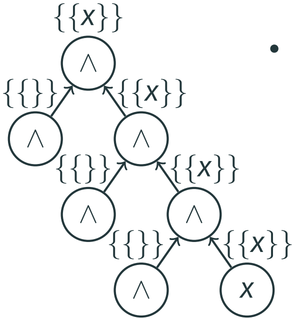
# Normalization: handling empty assignments



# Normalization: handling empty assignments

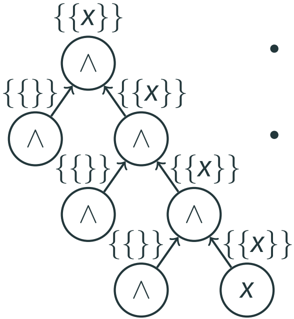


# Normalization: handling empty assignments



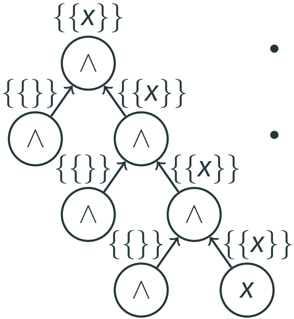

- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates

# Normalization: handling empty assignments



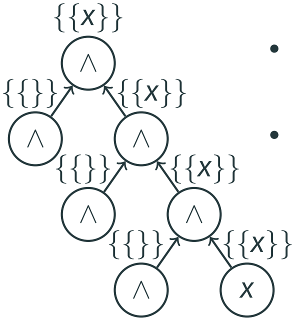

- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:

# Normalization: handling empty assignments



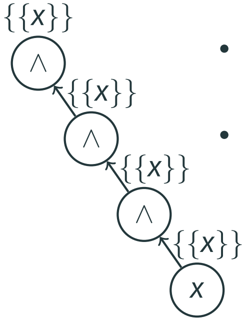

- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \{\{\}\}$ and $S(g) \setminus \{\{\}\}$ (homogenization)

# Normalization: handling empty assignments



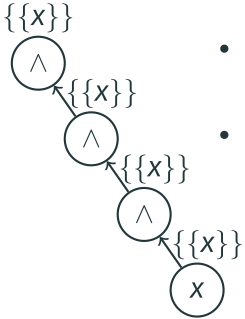

- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \{\{\}\}$ and $S(g) \setminus \{\{\}\}$ (homogenization)
  - remove inputs with $S(g) = \{\{\}\}$ for AND-gates

# Normalization: handling empty assignments



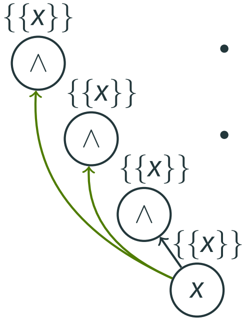

- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \{\{\}\}$ and $S(g) \setminus \{\{\}\}$ (homogenization)
  - remove inputs with $S(g) = \{\{\}\}$ for AND-gates

# Normalization: handling empty assignments



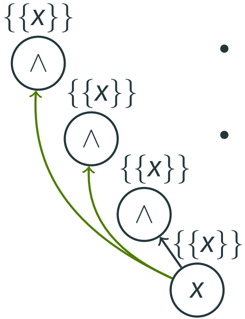

- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \{\{\}\}$ and $S(g) \setminus \{\{\}\}$ (homogenization)
  - remove inputs with $S(g) = \{\{\}\}$ for AND-gates
  - collapse AND-chains with fan-in 1

# Normalization: handling empty assignments




- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \{\{\}\}$ and $S(g) \setminus \{\{\}\}$ (homogenization)
  - remove inputs with $S(g) = \{\{\}\}$ for AND-gates
  - collapse AND-chains with fan-in 1

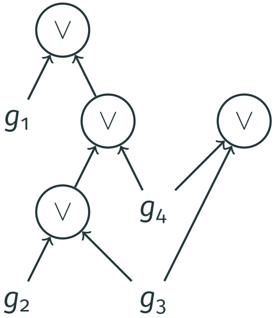
# Normalization: handling empty assignments




- Problem: if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \{\{\}\}$ and $S(g) \setminus \{\{\}\}$ (homogenization)
  - remove inputs with $S(g) = \{\{\}\}$ for AND-gates
  - collapse AND-chains with fan-in 1

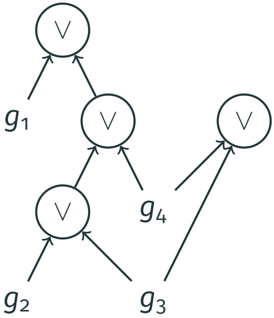
$\rightarrow$ Now, traversing an AND-gate ensures that we make progress: it splits the assignments non-trivially

# Normalization: handling OR-hierarchies



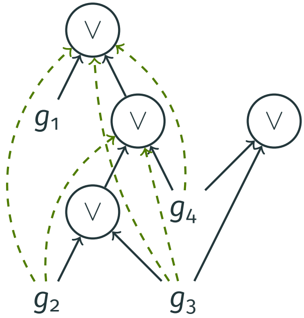

- Problem: we waste time in OR-hierarchies to find a reachable exit (non-OR gate)

# Normalization: handling OR-hierarchies



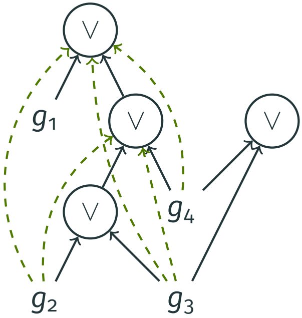

- Problem: we waste time in OR-hierarchies to find a reachable exit (non-OR gate)
- Solution: compute reachability index

# Normalization: handling OR-hierarchies



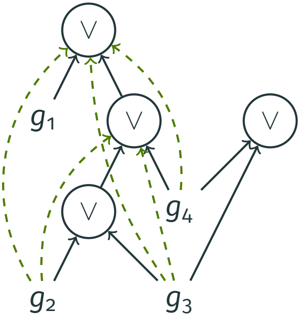

- **Problem:** we waste time in OR-hierarchies to find a **reachable exit** (non-OR gate)
- **Solution:** compute **reachability index**

# Normalization: handling OR-hierarchies




- **Problem:** we waste time in OR-hierarchies to find a **reachable exit** (non-OR gate)
- **Solution:** compute **reachability index**
- **Problem:** must be done in **linear time**

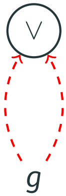
# Normalization: handling OR-hierarchies




- **Problem:** we waste time in OR-hierarchies to find a **reachable exit** (non-OR gate)
- **Solution:** compute **reachability index**
- **Problem:** must be done in **linear time**

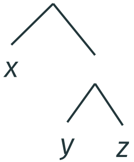
Solution:

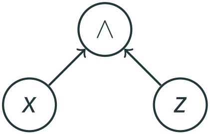
- **Determinism** ensures we have a **multitree** (we cannot have the pattern at the right)
- **Custom** constant-delay reachability index for multitrees



# Translating to zero-suppressed semantics

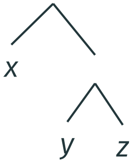
- This is where we use the v-tree

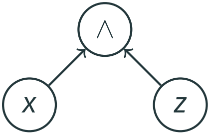


# Translating to zero-suppressed semantics

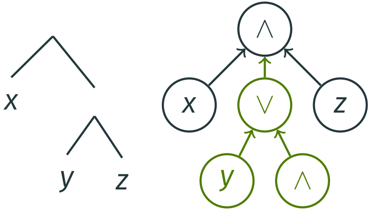
- This is where we use the v-tree
- Add explicitly untested variables



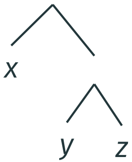
# Translating to zero-suppressed semantics

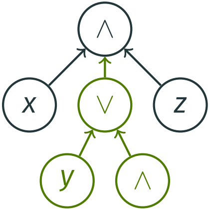
- This is where we use the **v-tree**
- Add explicitly **untested variables**

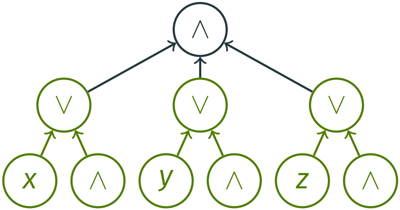


# Translating to zero-suppressed semantics

- This is where we use the **v-tree**
- Add explicitly **untested variables**



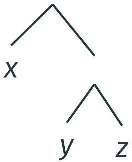



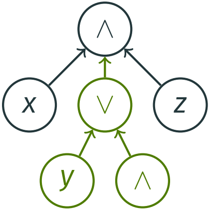



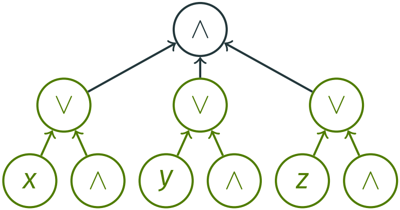

- **Problem:** quadratic blowup

# Translating to zero-suppressed semantics

- This is where we use the **v-tree**
- Add explicitly **untested variables**




- **Problem:** quadratic blowup
- **Solution:**
  - **Order** $<$ on variables in the v-tree ($x < y < z$)
  - **Interval** $[x, z]$
  - **Range gates** to denote $\bigvee[x, z]$ in constant space

# References


Bagan, G. (2006).
**MSO queries on tree decomposable structures are computable with linear delay.**
In *CSL*.

Kazana, W. and Segoufin, L. (2013).
**Enumeration of monadic second-order queries on trees.**
*TOCL*, 14(4).

Olteanu, D. and Závodnỳ, J. (2015).
**Size bounds for factorised representations of query results.**
*TODS*, 40(1).