TECHNISCHE UNIVERSITÄT DRESDEN

FOUNDATIONS OF DATABASES AND QUERY LANGUAGES

Lecture 4: Complexity of FO Query Answering

Markus Krötzsch

TU Dresden, 4 May 2015

Overview

How to Measure Query Answering Complexity

Query answering as decision problem
$\leadsto$ consider Boolean queries

Various notions of complexity:

- Combined complexity (complexity w.r.t. size of query and database instance)
- Data complexity (worst case complexity for any fixed query)
- Query complexity (worst case complexity for any fixed database instance)

Various common complexity classes:

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PS}\textsc{pace} \subseteq \mathrm{E}\textsc{xp}\mathrm{T}\textsc{ime}$$

An Algorithm for Evaluating FO Queries

function Eval($\varphi, \mathcal{I}$)

```
switch (φ) {
    case p(c_1, ..., c_n) : return ⟨c_1, ..., c_n⟩ ∈ p^I
    case ¬ψ : return ¬Eval(ψ, I)
    case ψ_1 ∧ ψ_2 : return Eval(ψ_1, I) ∧ Eval(ψ_2, I)
    case ∃x.ψ :
        for c ∈ Δ^I {
            if Eval(ψ[x ↦ c], I) then return true
        }
        return false
}
```

FO Algorithm Worst-Case Runtime

Let m be the size of φ, and let $n = |\mathcal{I}|$ (total table sizes)

- How many recursive calls of Eval are there?
  $\rightsquigarrow$ one per subexpression: at most m
- Maximum depth of recursion?
  $\rightsquigarrow$ bounded by total number of calls: at most m
- Maximum number of iterations of **`for`** loop?
  $\rightsquigarrow$ $|\Delta^{\mathcal{I}}| \leq n$ per recursion level
  $\rightsquigarrow$ at most n^m iterations
- Checking $\langle c_1, \ldots, c_n \rangle \in p^{\mathcal{I}}$ can be done in linear time w.r.t. n

Runtime in $m \cdot n^m \cdot n = m \cdot n^{m+1}$

Time Complexity of FO Algorithm

Let m be the size of φ, and let $n = |\mathcal{I}|$ (total table sizes)

Runtime in $m \cdot n^{m+1}$

Time complexity of FO query evaluation

- Combined complexity: in $\textsc{ExpTime}$
- Data complexity (m is constant): in P
- Query complexity (n is constant): in $\textsc{ExpTime}$

FO Algorithm Worst-Case Memory Usage

We can get better complexity bounds by looking at memory

Let m be the size of φ, and let $n = |\mathcal{I}|$ (total table sizes)

- For each (recursive) call, store pointer to current subexpression of φ: $\log m$
- For each variable in φ (at most m), store current constant assignment (as a pointer): $m \cdot \log n$
- Checking $\langle c_1, \ldots, c_n \rangle \in p^{\mathcal{I}}$ can be done in logarithmic space w.r.t. n

Memory in $m \log m + m \log n + \log n = m \log m + (m + 1) \log n$

Space Complexity of FO Algorithm

Let m be the size of φ, and let $n = |\mathcal{I}|$ (total table sizes)

Memory in $m \log m + (m+1) \log n$

Space complexity of FO query evaluation

- Combined complexity: in $\textsc{PSpace}$
- Data complexity (m is constant): in L
- Query complexity (n is constant): in $\textsc{PSpace}$

FO Combined Complexity

The algorithm shows that FO query evaluation is in $\textsc{PSpace}$.
Is this the best we can get?

Hardness proof: reduce a known $\textsc{PSpace}$-hard problem to FO query evaluation

FO Combined Complexity

The algorithm shows that FO query evaluation is in $\textsc{PSpace}$.
Is this the best we can get?

Hardness proof: reduce a known $\textsc{PSpace}$-hard problem to FO query evaluation
$\leadsto$ QBF satisfiability

Let $Q_1X_1.Q_2X_2.\cdots Q_nX_n.\varphi[X_1,\ldots,X_n]$ be a QBF (with $Q_i \in \{\forall, \exists\}$)

- Database instance $\mathcal{I}$ with $\Delta^{\mathcal{I}} = \{0, 1\}$
- One table with one row: true(1)
- Transform input QBF into Boolean FO query

$$Q_1x_1.Q_2x_2.\cdots Q_nx_n.\varphi[X_1 \mapsto \text{true}(x_1),\ldots,X_n \mapsto \text{true}(x_n)]$$

PSpace-hardness for DI Queries

The previous reduction from QBF may lead to a query that is not domain independent

Example: QBF $\exists p.\neg p$ leads to FO query $\exists x.\neg \mathrm{true}(x)$

PSPACE-hardness for DI Queries

The previous reduction from QBF may lead to a query that is not domain independent

Example: QBF $\exists p.\neg p$ leads to FO query $\exists x.\neg\text{true}(x)$

Better approach:

- Consider QBF $\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_n X_n.\varphi[X_1,\ldots,X_n]$ with φ in negation normal form: negations only occur directly before variables X_i (still PSPACE-complete: exercise)
- Database instance $\mathcal{I}$ with $\Delta^{\mathcal{I}} = \{0,1\}$
- Two tables with one row each: true(1) and false(0)
- Transform input QBF into Boolean FO query

$$\mathsf{Q}_1 x_1.\mathsf{Q}_2 x_2.\cdots\mathsf{Q}_n x_n.\varphi'$$

  where φ' is obtained by replacing each negated variable $\neg X_i$ with false(x_i) and each non-negated variable X_i with true(x_i).

Combined Complexity of FO Query Answering

Theorem

The evaluation of FO queries is $\textsc{PSpace}$-complete with respect to combined complexity.

We have actually shown something stronger:

Theorem

The evaluation of FO queries is $\textsc{PSpace}$-complete with respect to query complexity.

Data Complexity of FO Query Answering

The algorithm showed that FO query evaluation is in L
$\leadsto$ can we do any better?

What could be better than L?

$$? \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \ldots$$

$\leadsto$ we need to define circuit complexities first

Boolean Circuits

Definition

A Boolean circuit is a finite, directed, acyclic graph where

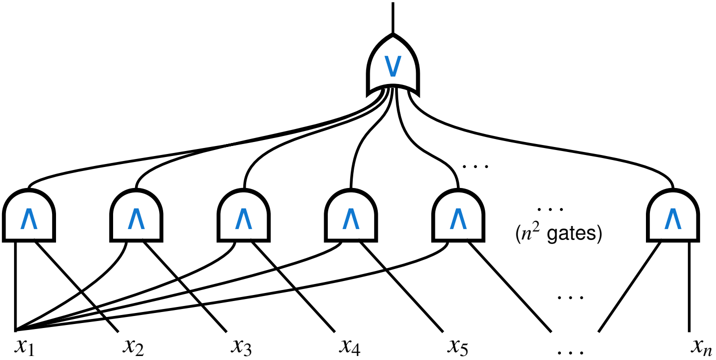
- each node that has no predecessors is an input node
- each node that is not an input node is one of the following types of logical gate: AND, OR, NOT
- one or more nodes are designated output nodes

$\rightsquigarrow$ we will only consider Boolean circuits with exactly one output

$\rightsquigarrow$ propositional logic formulae are Boolean circuits with one output and gates of fanout ≤ 1

Example

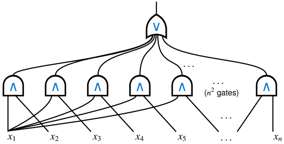
A Boolean circuit over an input string $x_1 x_2 \ldots x_n$ of length n



Corresponds to formula $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee \ldots \vee (x_{n-1} \wedge x_n)$
$\leadsto$ accepts all strings with at least two 1s

Circuits as a Model for Parallel Computation

Previous example:



$\leadsto$ n^2 processors working in parallel
$\leadsto$ computation finishes in 2 steps

- size: number of gates = total number of computing steps
- depth: longest path of gates = time for parallel computation

$\leadsto$ refinement of polynomial time taking parallelizability into account

Solving Problems With Circuits

Observation: the input size is “hard-wired” in circuits
$\leadsto$ each circuit only has a finite number of different inputs
$\leadsto$ not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?

Solving Problems With Circuits

Observation: the input size is "hard-wired" in circuits
$\rightsquigarrow$ each circuit only has a finite number of different inputs
$\rightsquigarrow$ not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?

Definition

A uniform family of Boolean circuits is a set of circuits C_n $(n \geq 0)$ that can be computed from n (usually in logarithmic space or time; we don't discuss the details here).

A language $\mathcal{L} \subseteq \{0, 1\}^*$ is decided by a uniform family $(C_n)_{n \geq 0}$ of Boolean circuits if for each word w of length $|w|$:

$$w \in \mathcal{L} \quad \text{if and only if} \quad C_{|w|}(w) = 1$$

Measuring Complexity with Boolean Circuits

How to measure the computing power of Boolean circuits?

Relevant metrics:

- size of the circuit: overall number of gates (as function of input size)
- depth of the circuit: longest path of gates (as function of input size)
- fan in: two inputs per gate or any number of inputs per gate?

Important classes of circuits: small-depth circuits

Definition

$(C_n)_{n\geq 0}$ is a family of small-depth circuits if

- the size of C_n is polynomial in n,
- the depth of C_n is poly-logarithmic in n, that is, $O(\log^k n)$.

The Complexity Classes NC and AC

Two important types of small-depth circuits

Definition

NC^k is the class of problems that can be solved by uniform families of circuits $(C_n)_{n\geq 0}$ of fan-in ≤ 2, size polynomial in n, and depth in $O(\log^k n)$.

The class NC is defined as $\mathrm{NC} = \bigcup_{k\geq 0} \mathrm{NC}^k$.

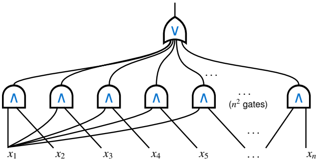
("Nick's Class" named after Nicholas Pippenger by Stephen Cook)

Definition

AC^k and AC are defined like NC^k and NC, respectively, but for circuits with arbitrary fan-in.

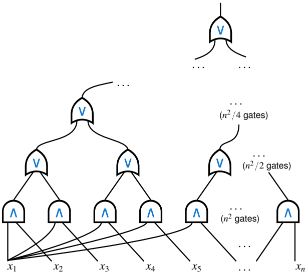
(A is for "Alternating": AND-OR gates alternate in such circuits)

Example



family of polynomial size,
constant depth,
arbitrary fan-in circuits
$\leadsto$ in AC^0

We can eliminate arbitrary fan-ins by using more layers of gates:



family of polynomial size,
logarithmic depth,
bounded fan-in circuits
$\leadsto$ in NC^1

Relationships of Circuit Complexity Classes

The previous sketch can be generalised:

$$\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{AC}^1 \subseteq \ldots \subseteq \mathrm{AC}^k \subseteq \mathrm{NC}^{k+1} \subseteq \ldots$$

Only few inclusions are known to be proper: $\mathrm{NC}^0 \subset \mathrm{AC}^0 \subset \mathrm{NC}^1$

Direct consequence of above hierarchy: $\mathrm{NC} = \mathrm{AC}$

Interesting relations to other classes:

$$\mathrm{NC}^0 \subset \mathrm{AC}^0 \subset \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{AC}^1 \subseteq \ldots \subseteq \mathrm{NC} \subseteq \mathrm{P}$$

Intuition:

- Problems in NC are parallelisable
- Problems in $\mathrm{P} \setminus \mathrm{NC}$ are inherently sequential

However: it is not known if $\mathrm{NC} \neq \mathrm{P}$

Back to Databases . . .

Theorem

The evaluation of FO queries is complete for (logtime uniform) AC^0 with respect to data complexity.

Proof:

- Membership: For a fixed Boolean FO query, provide a uniform construction for a small-depth circuit based on the size of a database
- Hardness: Show that circuits can be transformed into Boolean FO queries in logarithmic time (not on a standard TM . . . not in this lecture)

From Query to Circuit

Assumption:

- query and database schema is fixed
- database instance (and thus active domain) are variable

Construct circuit uniformly based on size of active domain

Sketch of construction:

- one input node for each possible database tuple (over given schema and active domain)
  $\leadsto$ true or false depending on whether tuple is present or not
- Recursively, for each subformula, introduce a gate for each possible tuple (instantiation) of this formula
  $\leadsto$ true or false depending on whether the subformula holds for this tuple or not
- Logical operators correspond to gate types: basic operators obvious, $\forall$ as generalised conjunction, $\exists$ as generalised disjunction
- subformula with n free variables $\leadsto$ $|\mathbf{adom}|^n$ gates
  $\leadsto$ especially: $|\mathbf{adom}|^0 = 1$ output gate for Boolean query

Example

We consider the formula

$$\exists z.(\exists x.\exists y.R(x, y) \land S(y, z)) \land \neg R(a, z)$$

Over the database instance:

R:

a	a
a	b

S:

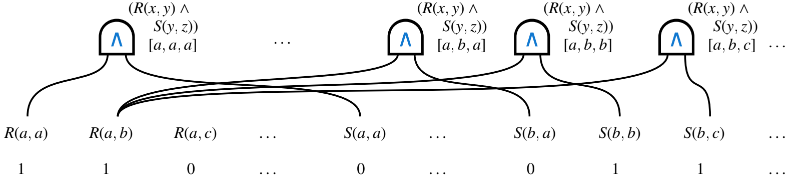
b	b
b	c

Active domain: $\{a, b, c\}$

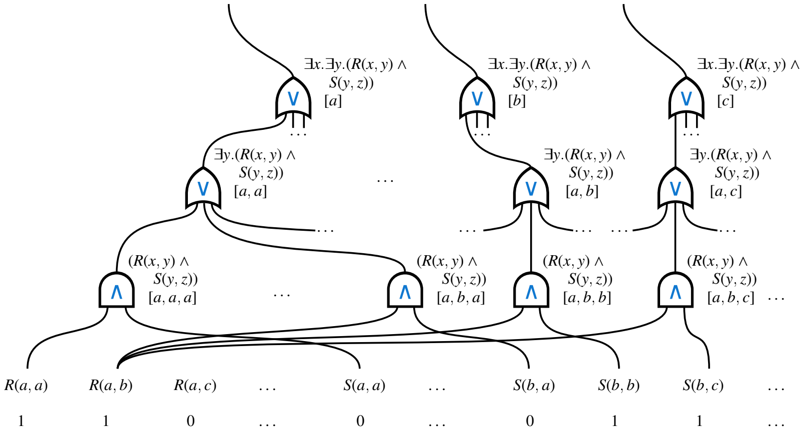
Example: $\exists z.(\exists x.\exists y.R(x, y) \wedge S(y, z)) \wedge \neg R(a, z)$

$R(a,a)$	$R(a,b)$	$R(a,c)$	$\ldots$	$S(a,a)$	$\ldots$	$S(b,a)$	$S(b,b)$	$S(b,c)$	$\ldots$
1	1	0	$\ldots$	0	$\ldots$	0	1	1	$\ldots$

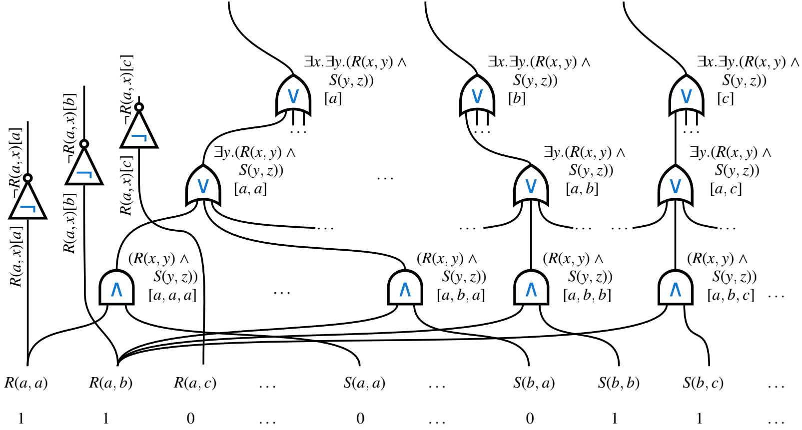
Example: $\exists z.(\exists x.\exists y.R(x,y) \land S(y,z)) \land \neg R(a,z)$



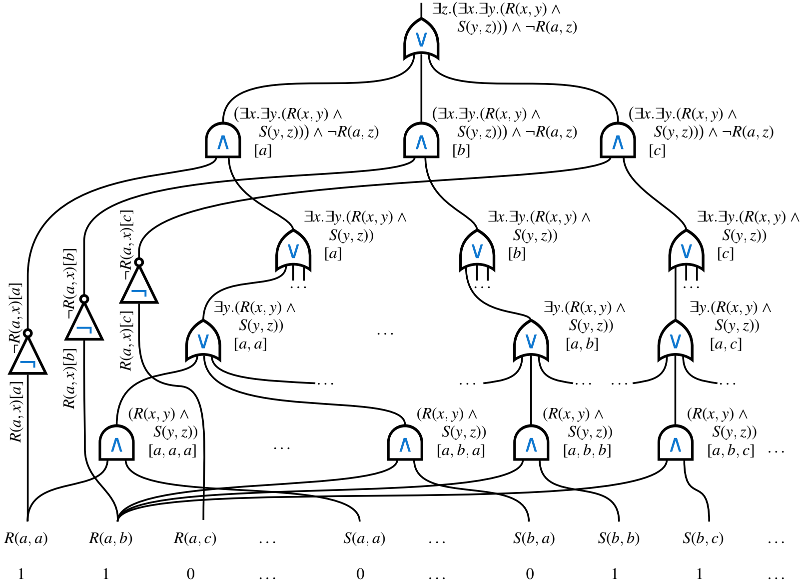
Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



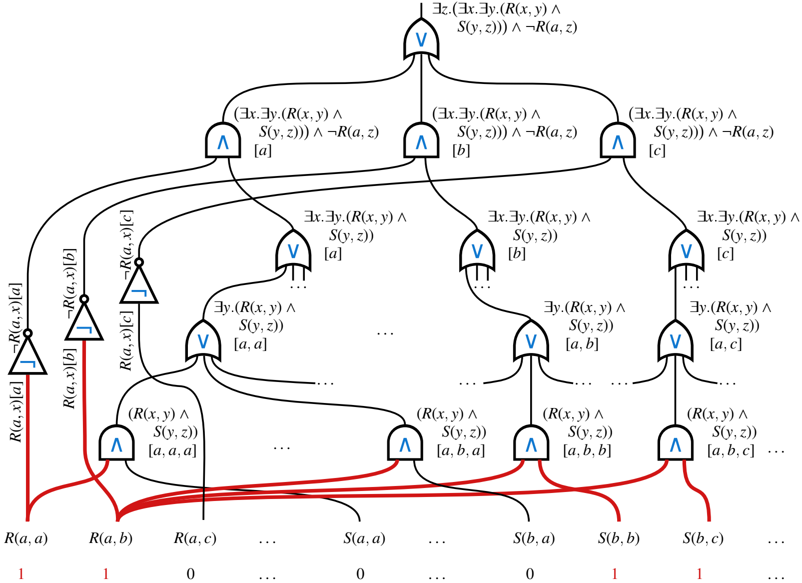
Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



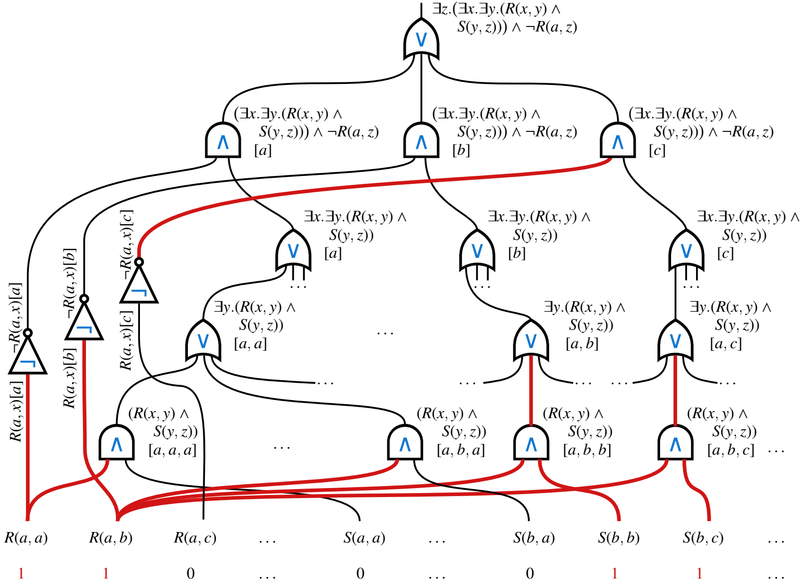
Example: $\exists z.(\exists x.\exists y.R(x,y) \land S(y,z)) \land \neg R(a,z)$



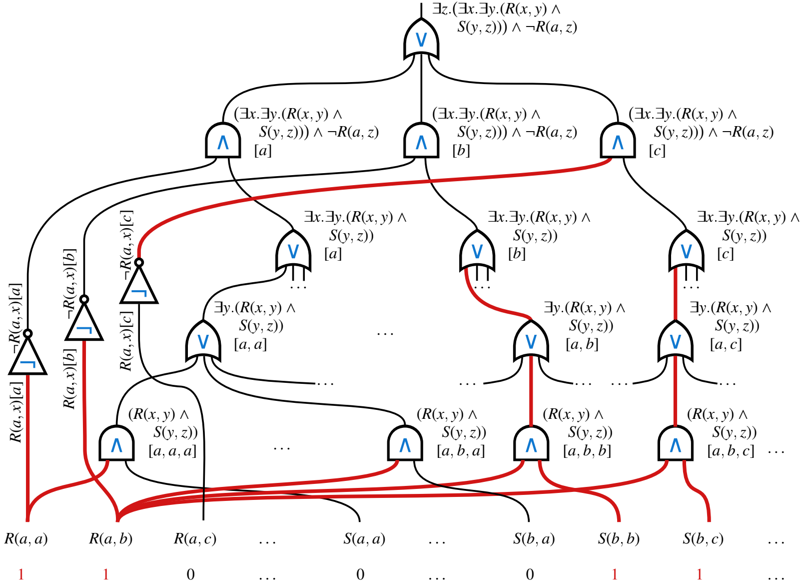
Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



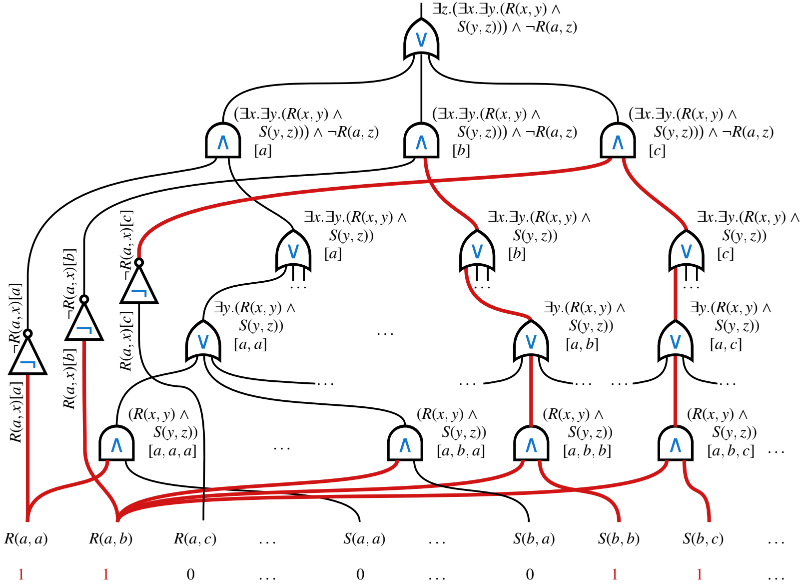
Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



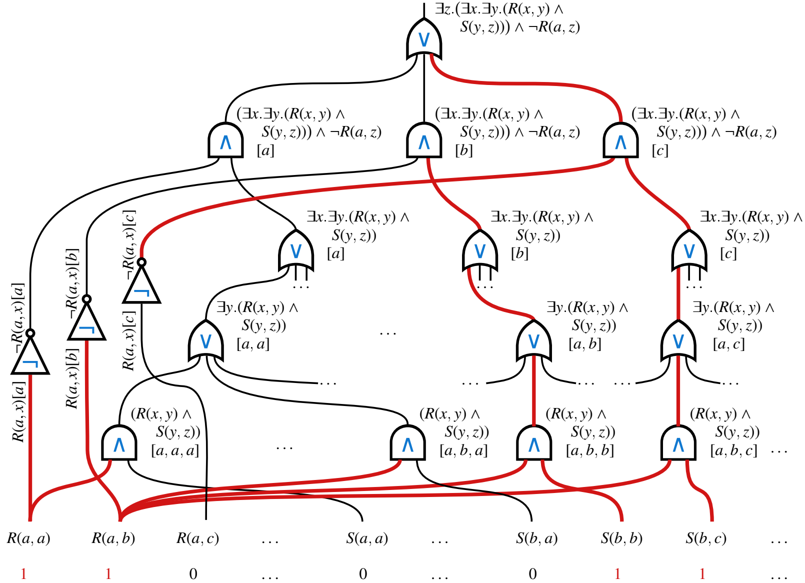
Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



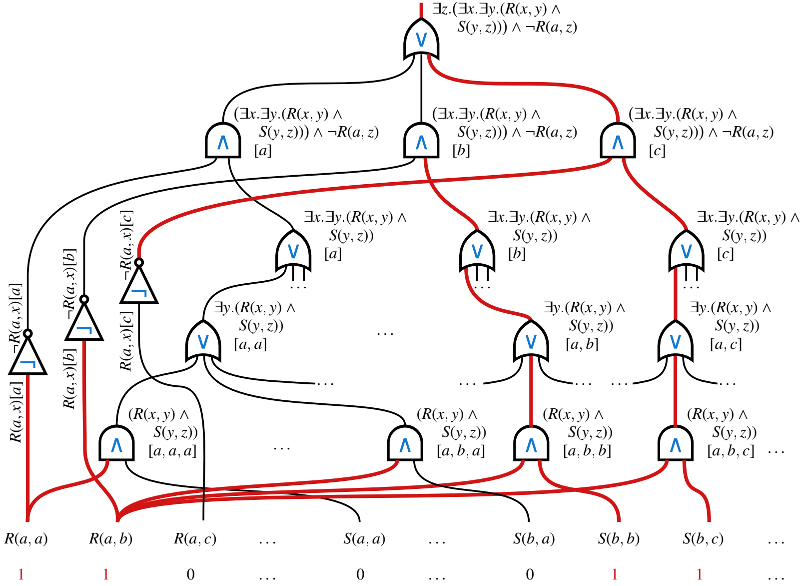
Example: $\exists z.(\exists x.\exists y.R(x,y) \land S(y,z)) \land \neg R(a,z)$



Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



Example: $\exists z.(\exists x.\exists y.R(x,y) \wedge S(y,z)) \wedge \neg R(a,z)$



Summary and Outlook

The evaluation of FO queries is

- $\textsc{PSpace}$-complete for combined complexity
- $\textsc{PSpace}$-complete for query complexity
- AC^0-complete for data complexity

Circuit complexities help to identify highly parallelisable problems in P

Open questions:

- Which other computing problems are interesting? (next lecture)
- Are there query languages with lower complexities?
- How can we study the expressiveness of query languages?