

Recursive Solutions to First-Order Model Counting

Paulius Dilkas, Vaishak Belle

AAI Seminar, 28th March 2022

Some Elementary Counting

A Counting Problem

Suppose we were meeting in person, the room had n seats, and there were $m \leq n$ attendees. How many ways would there be to seat everyone?

Some Elementary Counting

A Counting Problem

Suppose we were meeting in person, the room had n seats, and there were $m \leq n$ attendees. How many ways would there be to seat everyone?

More explicitly, we assume that:

- each attendee gets one seat (i.e., at least one **and** at most one),
- and a seat can accommodate at most one person.

Some Elementary Counting

A Counting Problem

Suppose we were meeting in person, the room had n seats, and there were $m \leq n$ attendees. How many ways would there be to seat everyone?

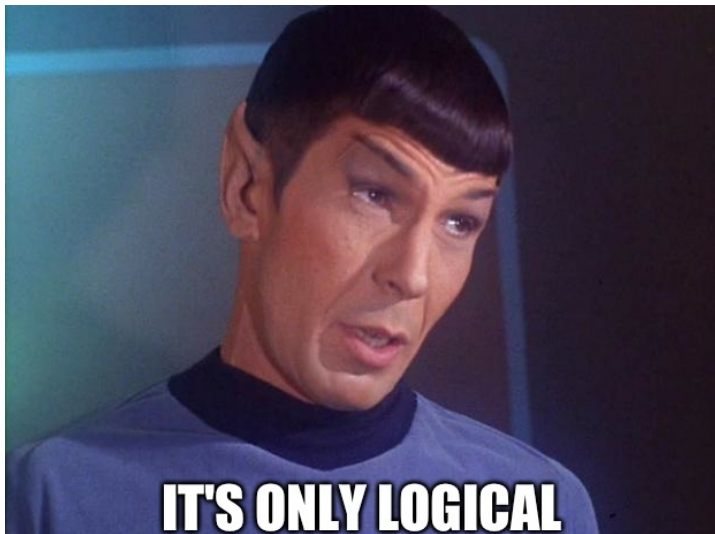
More explicitly, we assume that:

- each attendee gets one seat (i.e., at least one **and** at most one),
- and a seat can accommodate at most one person.

Answer: $n^{\underline{m}} = n \cdot (n - 1) \cdots (n - m + 1)$.

Note: this problem is equivalent to counting $[m] \rightarrow [n]$ injections.

Let's Express This Problem in Logic!



Let's Express This Problem in Logic!

- Let M and N be sets (i.e., **domains**) such that $|M| = m$, and $|N| = n$
- Let $P \subseteq M \times N$ be a relation (i.e., **predicate**) over sets M and N
- We can describe all of the constraints in first-order logic:

Let's Express This Problem in Logic!

- Let M and N be sets (i.e., **domains**) such that $|M| = m$, and $|N| = n$
- Let $P \subseteq M \times N$ be a relation (i.e., **predicate**) over sets M and N
- We can describe all of the constraints in first-order logic:
 - each attendee gets a seat (i.e., at least one seat)

$$\forall x \in M. \exists y \in N. P(x, y)$$

Let's Express This Problem in Logic!

- Let M and N be sets (i.e., **domains**) such that $|M| = m$, and $|N| = n$
- Let $P \subseteq M \times N$ be a relation (i.e., **predicate**) over sets M and N
- We can describe all of the constraints in first-order logic:
 - each attendee gets a seat (i.e., at least one seat)

$$\forall x \in M. \exists y \in N. P(x, y)$$

- one person cannot occupy multiple seats

$$\forall x \in M. \forall y, z \in N. P(x, y) \wedge P(x, z) \Rightarrow y = z$$

Let's Express This Problem in Logic!

- Let M and N be sets (i.e., **domains**) such that $|M| = m$, and $|N| = n$
- Let $P \subseteq M \times N$ be a relation (i.e., **predicate**) over sets M and N
- We can describe all of the constraints in first-order logic:
 - each attendee gets a seat (i.e., at least one seat)

$$\forall x \in M. \exists y \in N. P(x, y)$$

- one person cannot occupy multiple seats

$$\forall x \in M. \forall y, z \in N. P(x, y) \wedge P(x, z) \Rightarrow y = z$$

- one seat cannot accommodate multiple attendees

$$\forall w, x \in M. \forall y \in N. P(w, y) \wedge P(x, y) \Rightarrow w = x$$

Let's Express This Problem in Logic!

- Let M and N be sets (i.e., **domains**) such that $|M| = m$, and $|N| = n$
- Let $P \subseteq M \times N$ be a relation (i.e., **predicate**) over sets M and N
- We can describe all of the constraints in first-order logic:
 - each attendee gets a seat (i.e., at least one seat)

$$\forall x \in M. \exists y \in N. P(x, y)$$

- one person cannot occupy multiple seats

$$\forall x \in M. \forall y, z \in N. P(x, y) \wedge P(x, z) \Rightarrow y = z$$

- one seat cannot accommodate multiple attendees

$$\forall w, x \in M. \forall y \in N. P(w, y) \wedge P(x, y) \Rightarrow w = x$$

The first two sentences constrain P to be a function, and the last one makes it injective.

Let's Express This Problem in Logic!

- Let M and N be sets (i.e., **domains**) such that $|M| = m$, and $|N| = n$
- Let $P \subseteq M \times N$ be a relation (i.e., **predicate**) over sets M and N
- We can describe all feasible assignments in first order logic:

- each attendee



- one person can

 $\forall x$
 $z) \Rightarrow y = z$

- one seat cannot accommodate multiple attendees

$$\forall w, x \in M. \forall y \in N. P(w, y) \wedge P(x, y) \Rightarrow w = x$$

The first two sentences constrain P to be a function, and the last one makes it injective.

Overview of the Problem

- **First-order model counting** is the problem of counting the models of a sentence in first-order logic.
- The **(symmetric) weighted** variation of the problem adds weights (e.g., probabilities) to predicates.
- Thus, SWFOMC can also be used for efficient **probabilistic inference** in relational models.
- None of the (implemented) (SW)FOMC algorithms are able to count, e.g., **injective** and **bijjective** functions.

Claim

This shortcoming can be addressed via support for (almost arbitrary) **recursive functions**.

Back to Our Example

For instance, the following function counts injections

$$f(m, n) = \begin{cases} 1 & \text{if } m = 0 \text{ and } n = 0 \\ 0 & \text{if } m > 0 \text{ and } n = 0 \\ f(m, n - 1) + mf(m - 1, n - 1) & \text{otherwise.} \end{cases}$$

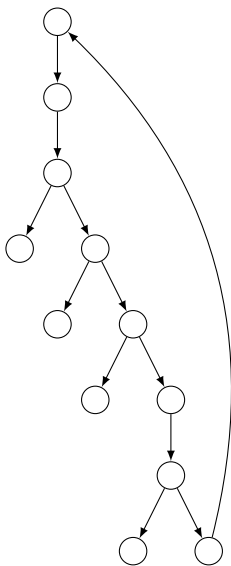
Back to Our Example

For instance, the following function counts injections

$$f(m, n) = \begin{cases} 1 & \text{if } m = 0 \text{ and } n = 0 \\ 0 & \text{if } m > 0 \text{ and } n = 0 \\ f(m, n - 1) + mf(m - 1, n - 1) & \text{otherwise.} \end{cases}$$

- f can be computed in $\Theta(mn)$ time (via dynamic programming).
- Optimal time complexity to compute n^m is $\Theta(\log m)$.
- But $\Theta(mn)$ is still much better than solving an equivalent **#P-complete** problem in propositional logic.
- The rest of this talk is about how such functions can be found automatically.

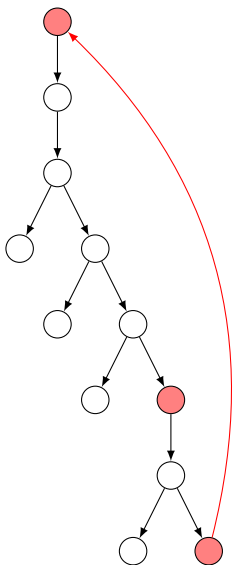
First-Order Knowledge Compilation



Workflow Before

- ① Compile the formula to a **circuit**
- ② Evaluate to get the answer

First-Order Knowledge Compilation



Workflow Before

- ① Compile the formula to a **circuit**
- ② Evaluate to get the answer

Workflow After

- ① Compile the formula to a **graph**
- ② Extract the definitions of functions
- ③ Simplify
- ④ Supplement with **base cases**
- ⑤ Evaluate to get the answer

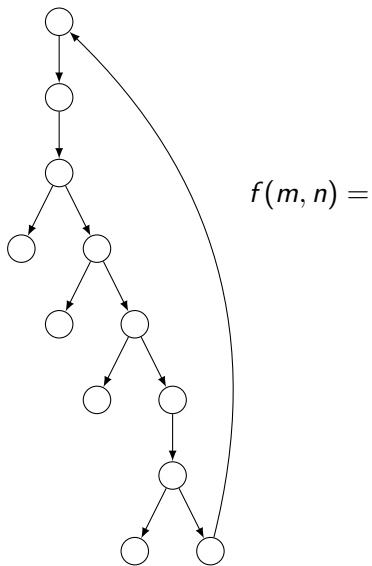
More Formally...

Definition

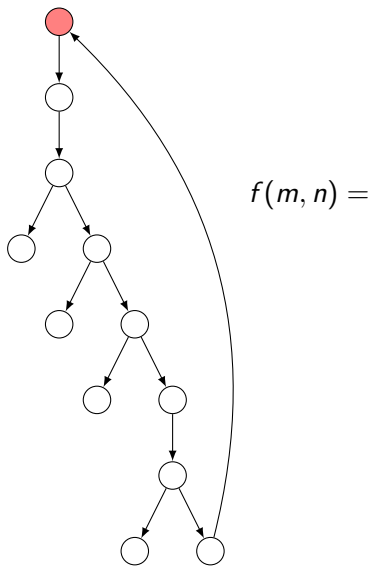
A first-order deterministic decomposable negation normal form computational graph (FCG) is a

- directed graph
- (which is weakly connected)
- with a single source,
- labelled vertices,
- and ordered outgoing edges.

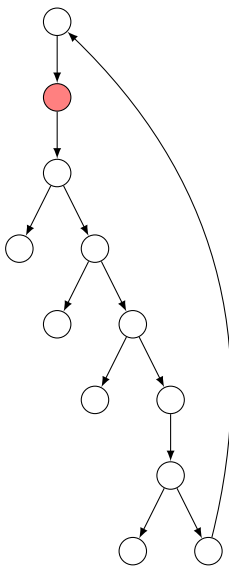
How to Interpret an FCG



How to Interpret an FCG

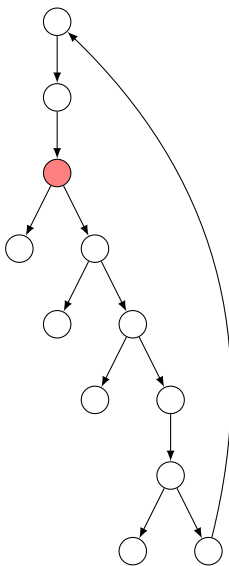


How to Interpret an FCG



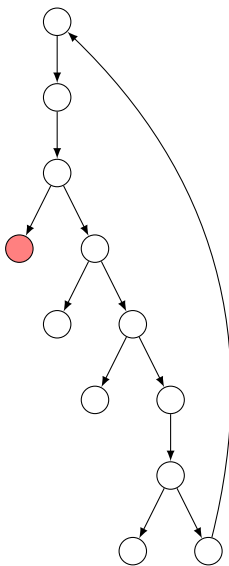
$$f(m, n) = \sum_{l=0}^m \binom{m}{l}$$

How to Interpret an FCG



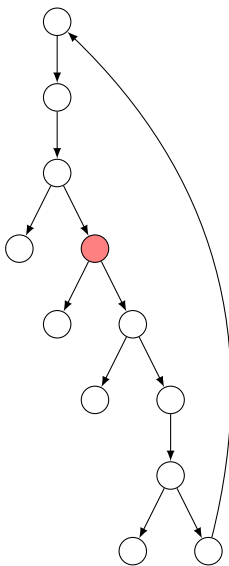
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} \times$$

How to Interpret an FCG



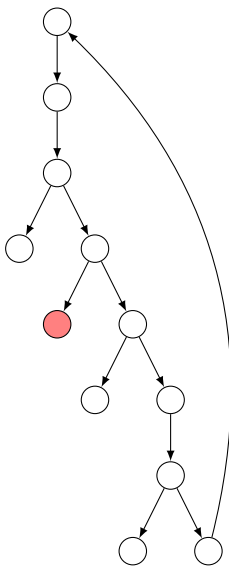
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times$$

How to Interpret an FCG



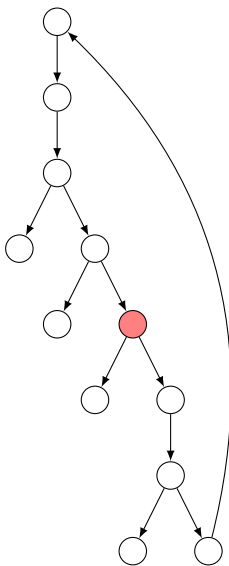
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times \times$$

How to Interpret an FCG



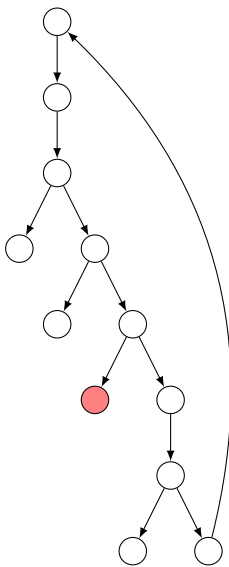
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times$$

How to Interpret an FCG



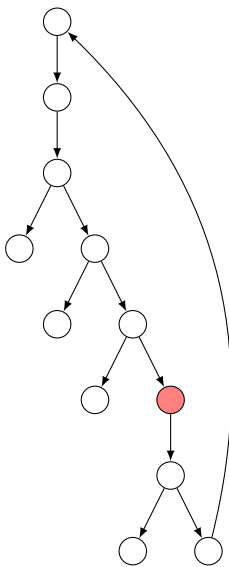
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times \dots \times$$

How to Interpret an FCG



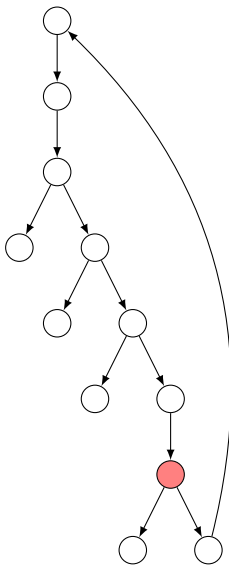
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times 1 \times$$

How to Interpret an FCG



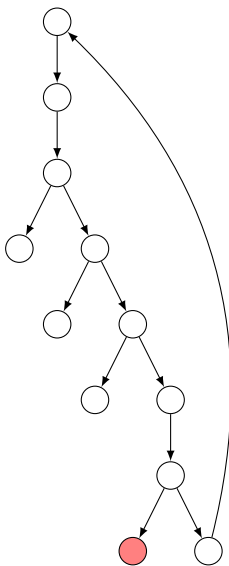
$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times 1 \times$$

How to Interpret an FCG



$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times 1 \times \dots \times$$

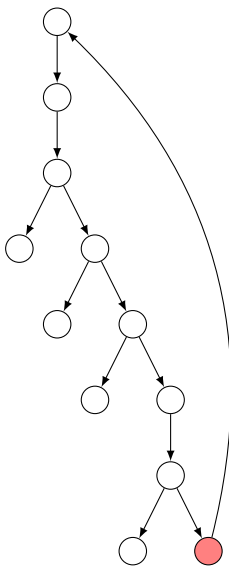
How to Interpret an FCG



$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times 1 \times [l < 2] \times$$

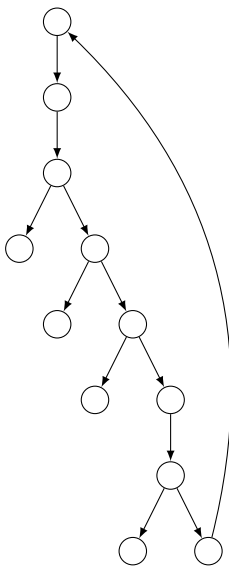
$$[\phi] = \begin{cases} 1 & \text{if } \phi \\ 0 & \text{if } \neg\phi \end{cases}$$

How to Interpret an FCG



$$f(m, n) = \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times 1 \times [l < 2] \times f(m - l, n - 1)$$

How to Interpret an FCG



$$\begin{aligned}
 f(m, n) &= \sum_{l=0}^m \binom{m}{l} 1 \times 1 \times 1 \times [l < 2] \times f(m-l, n-1) \\
 &= f(m, n-1) + mf(m-1, n-1)
 \end{aligned}$$

Compilation Rules

Definition

A (compilation) rule is a function that takes a formula and returns a set of (G, L) pairs, where

- G is a (potentially null) FCG,
- and L is a list of formulas.

Example Rule: Independence

Input formula:

$$(\forall x, y \in L. x = y) \wedge \quad (1)$$

$$(\forall x \in M. \forall y, z \in N. P(x, y) \wedge P(x, z) \Rightarrow y = z) \wedge \quad (2)$$

$$(\forall w, x \in M. \forall y \in N. P(w, y) \wedge P(x, y) \Rightarrow w = x) \quad (3)$$

Example Rule: Independence

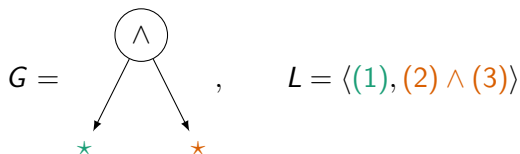
Input formula:

$$(\forall x, y \in L. x = y) \wedge \quad (1)$$

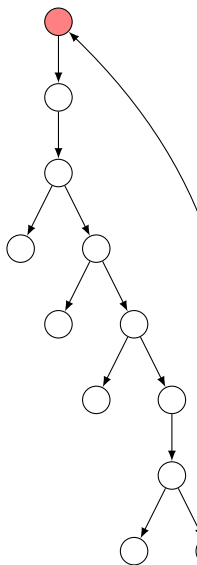
$$(\forall x \in M. \forall y, z \in N. P(x, y) \wedge P(x, z) \Rightarrow y = z) \wedge \quad (2)$$

$$(\forall w, x \in M. \forall y \in N. P(w, y) \wedge P(x, y) \Rightarrow w = x) \quad (3)$$

Only one (G, L) pair:



New Rule 1: (Generalised) Domain Recursion



Example

Input formula:

$$\forall x \in M. \forall y, z \in N. y \neq z \Rightarrow \neg P(x, y) \vee \neg P(x, z)$$

Output formula (with a new constant $c \in M$):

$$\forall y, z \in N. y \neq z \Rightarrow \neg P(c, y) \vee \neg P(c, z)$$

$$\forall x \in M. \forall y, z \in N. x \neq c \wedge y \neq z \Rightarrow \neg P(x, y) \vee \neg P(x, z)$$

New Rule 2: Constraint Removal

Example

Input formula (with a constant $c \in M$):

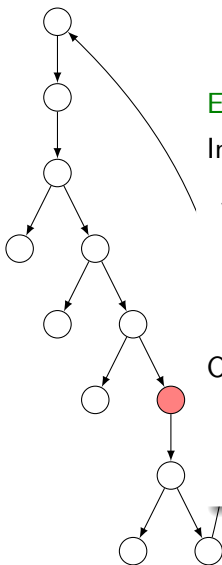
$$\forall x \in M. \forall y, z \in N. x \neq c \wedge y \neq z \Rightarrow \neg P(x, y) \vee \neg P(x, z)$$

$$\forall w, x \in M. \forall y \in N. w \neq c \wedge x \neq c \wedge w \neq x \Rightarrow \neg P(w, y) \vee \neg P(x, y)$$

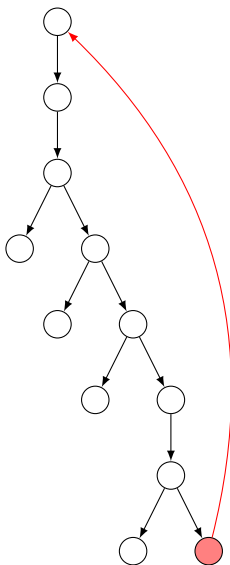
Output formula (with a new domain $M' := M \setminus \{c\}$):

$$\forall x \in M'. \forall y, z \in N. y \neq z \Rightarrow \neg P(x, y) \vee \neg P(x, z)$$

$$\forall w, x \in M'. \forall y \in N. w \neq x \Rightarrow \neg P(w, y) \vee \neg P(x, y)$$



New Rule 3: Identifying Possibilities for Recursion



Goal

Check if the input formula is isomorphic (up to domains) to a previously encountered formula.

Rough Outline

- 1 Consider pairs of 'similar' clauses.
- 2 Consider bijections between their sets of variables.
- 3 Extend each such bijection to a map between sets of domains.
- 4 If the bijection makes the clauses equal, and the domain map is compatible with previous domain maps, move on to another pair of clauses.

Summary & Future Work

Summary

The circuits hitherto used for FOMC become more powerful with:

- cycles,
- generalised domain recursion,
- and some more new compilation rules that support domain recursion.

Future Work

- Automate:
 - extracting and simplifying the definitions of functions,
 - finding all base cases.
- Open questions:
 - What kind of **sequences** are computable in this way?
 - Would using a **different logic** extend the capabilities of FOMC further?