

Finite Choice Logic Programming

CHRIS MARTENS
NORTHEASTERN UNIVERSITY

ROBERT J SIMMONS as DATALOG
OWL & CROW PRODUCTIONS

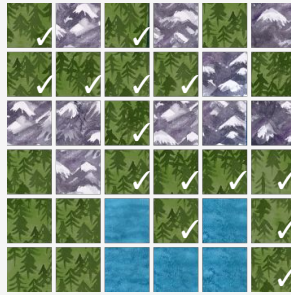
MICHAEL ARNTZENIUS as SAT
UC BERKELEY

POPL 2025
DENVER CO

DATALOG: Hi, I'm Datalog! I like representing things in logic.

SAT: Hi, I'm Boolean Satisfiability! I also like representing things in logic!

DATALOG: SAT, I bet you can help me with a problem I've been having.



- Each region contains mountains, forest, or ocean
- Oceans aren't next to mountains
- Calculate reachability-through-forest from home



DATALOG: I'm making a little adventure game that takes place on a tile grid, and I want to use logic to describe, and then generate, grids for the game.

Each tile has one of three types, and there's also a constraint that I don't want oceans to be right next to mountains. Finally, I want to calculate all the regions that are reachable from a home region through the forest.

SAT: Well, I know how to do *two* of those things.

$$\forall r \in R. \text{mountain}(r) \vee \text{forest}(r) \vee \text{ocean}(r)$$
$$\forall r \in R. (\neg \text{mountain}(r) \wedge \neg \text{forest}(r))$$
$$\vee (\neg \text{mountain}(r) \wedge \neg \text{ocean}(r))$$
$$\vee (\neg \text{forest}(r) \wedge \neg \text{ocean}(r))$$


SAT: So, for representing the possibility space, I can say every region is either a mountain, or a forest, or an ocean, and it isn't at least two of those things. So that means there's exactly one terrain type per region.

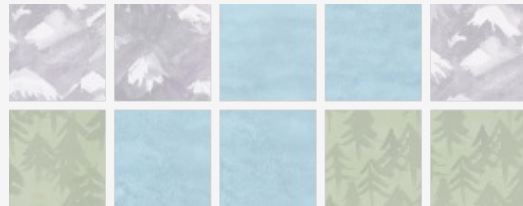
$$\forall r \in R. \text{mountain}(r) \vee \text{forest}(r) \vee \text{ocean}(r)$$

$$\forall r \in R. (\neg \text{mountain}(r) \wedge \neg \text{forest}(r))$$

$$\quad \vee (\neg \text{mountain}(r) \wedge \neg \text{ocean}(r))$$

$$\quad \vee (\neg \text{forest}(r) \wedge \neg \text{ocean}(r))$$

$$\forall r_1, r_2 \in R. (\text{ocean}(r_1) \wedge \text{next_to}(r_1, r_2))$$

$$\quad \supset (\text{forest}(r_2) \vee \text{ocean}(r_2))$$


SAT: And as for the constraint, I can say that if a region is next to an ocean then it's either a forest or an ocean, so it can't be mountain.

But I don't know how to calculate reachability.

DATALOG: No problem, I've actually got that part.

$reach(\text{🏠})$

$reach(r_2) :- reach(r_1), next_to(r_1, r_2), forest(r_2)$



DATALOG: I can describe reachability in Datalog with just two rules. The first rule forces the home region to be marked as reachable...

$reach(\text{house})$

$reach(r_2) :- reach(r_1), next_to(r_1, r_2), forest(r_2)$



DATALOG: ...and the second rule forces me to mark each transitively reachable region.

reach()

reach(r_2) : - reach(r_1), next_to(r_1, r_2), forest(r_2)



DATALOG: Once I can't derive any more facts, I conclude that every other region is *not* reachable.

$reach(\text{🏠})$

$reach(r_2) :- reach(r_1), next_to(r_1, r_2), forest(r_2)$



$reach(\text{🏠})$

$\forall r_1, r_2. reach(r_1) \wedge next_to(r_1, r_2) \wedge forest(r_2) \supset reach(r_2)$



DATALOG: I know rules in Datalog correspond to propositions in logic, so I should be able to replace comma with and, flip the implication around, and send this to a SAT solver, right?

SAT: You certainly can send it to a SAT solver, and your intended solution does satisfy that proposition. Unfortunately, so do many other things.

$reach(\text{🏠})$

$reach(r_2) :- reach(r_1), next_to(r_1, r_2), forest(r_2)$



$reach(\text{🏠})$

$\forall r_1, r_2. reach(r_1) \wedge next_to(r_1, r_2) \wedge forest(r_2) \supset reach(r_2)$



SAT: This if-then forces some reachability facts to be true, but it doesn't force any of them to be false, so as a SAT solver, I can freely choose to make any or all of them reachable.

$reach(\text{house})$

$reach(r_2) :- reach(r_1), next_to(r_1, r_2), forest(r_2)$



$\forall r_1, r_2. reach(r_2) \leftrightarrow$
 $(r_2 = \text{house}) \vee$
 $(reach(r_1) \wedge next_to(r_1, r_2) \wedge forest(r_2))$



SAT: You might think you could fix this by turning the if-then into an if-and-only-if: this is called the **Clark completion** of the program. Unfortunately, this still doesn't work.

It forces every reachable region to be next to a reachable forest, but if you look at this example on the right, you'll see there's two forests – which we don't want to be reachable – which are next to one another. Now if one of them is reachable, all of our if-and-only-if conditions for the other one are satisfied...

$reach(\text{house})$

$reach(r_2) :- reach(r_1), next_to(r_1, r_2), forest(r_2)$



$\forall r_1, r_2. reach(r_2) \leftrightarrow$

$(r_2 = \text{house}) \vee$

$(reach(r_1) \wedge next_to(r_1, r_2) \wedge forest(r_2))$



SAT: ...so we can assign both of them to be reachable and this still satisfies our proposition.

This is a kind of general problem with SAT. It's not really equipped to define inductive properties like reachability or transitive closure.

$$\forall r \in R. \text{mountain}(r) \vee \text{forest}(r) \vee \text{ocean}(r)$$

$$\forall r \in R. (\neg \text{mountain}(r) \wedge \neg \text{forest}(r))$$

$$\quad \vee (\neg \text{mountain}(r) \wedge \neg \text{ocean}(r))$$

$$\quad \vee (\neg \text{forest}(r) \wedge \neg \text{ocean}(r))$$

$$\forall r_1, r_2 \in R. (\text{ocean}(r_1) \wedge \text{next_to}(r_1, r_2))$$

$$\quad \supset (\text{forest}(r_2) \vee \text{ocean}(r_2))$$

reach(🏠)

reach(r_2) :- *reach*(r_1), *next_to*(r_1, r_2), *forest*(r_2)



DATALOG: So, this is a bummer. Like, I can represent half my problem in logic according to a SAT solver, and represent half of the problem in logic according to datalog, but I can't represent the whole problem with logic all at once.

[DATALOG shakes fists at the sky!!!1!!1one!!]

DATALOG: If only there were some hero who could help us!

INTRODUCING

CHRIS MARTENS

as FINITE CHOICE
LOGIC
PROGRAMMING

FCLP: Oh, hi! Hi Datalog! Hi Boolean Satisfiability! I'm Finite-Choice Logic Programming, and I think I might actually be able to resolve your tensions.

Datalog, something I've always liked about you is that you have a very high standard of evidence: you'll only accept that something is true if there's some kind of justification for it, right?

On the other hand, something I've always admired about you, Boolean Satisfiability, is that you've got a very open mind. You'll accept many possible explanations for the evidence you see, whereas Datalog only accepts one canonical version of the events.

So, let's see what we can do about this, right?

<i>proposition</i>	<i>truth value</i>	<i>attribute</i>	<i>value</i>
<i>mountain</i> (🏠)	✗ false	<i>terrain</i> (🏠)	🌳 forest
<i>forest</i> (🏠)	✓ true		
<i>ocean</i> (🏠)	✗ false		

FCLP: I want to start with a fresh slate, and we'll move away from this black and white, boolean mindset.

[SAT removes his black-and-white checkered hat as he considers this]

FCLP: Instead of having a predicate forest-of-home that can have the value true or false, we're going to introduce something called an **attribute**, terrain-of-home, which can have the **value** mountain, forest, or ocean.

<i>proposition</i>	<i>truth value</i>	<i>attribute</i>	<i>value</i>
<i>mountain</i> (🏠)	✗ false	<i>terrain</i> (🏠)	🌳 forest
<i>forest</i> (🏠)	✓ true		
<i>ocean</i> (🏠)	✗ false		

terrain(*r*) IS 🌳

 "attribute" "value"

FCLP: We'll write this as "the terrain of region is forest", where the "is" here is just syntax. Critically, what we want is that each attribute – so, in this example, the terrain of each region – can only ever have one value at a time.

<i>proposition</i>	<i>truth value</i>	<i>attribute</i>	<i>value</i>
<i>mountain</i> (🏠)	✗ false	<i>terrain</i> (🏠)	🌳 forest
<i>forest</i> (🏠)	✓ true		
<i>ocean</i> (🏠)	✗ false		

functional dependency

DATALOG: That's actually totally reasonable from a Datalog perspective. What you're saying that terrain is a two-place relation, with a functional dependency from the first argument, the region, to the second argument, the terrain type. It's actually pretty common in Datalogs to have functional dependencies.


- Calculate reachability-through-forest from home
- Each region contains mountains, forest, or ocean
- Oceans aren't next to mountains



FCLP: Cool. So, if we make this change we can return to our original spec and start writing this program.

$reach(\text{🏠})$


$reach(r_2) :- reach(r_1), next_to(r_1, r_2), terrain(r_2) \text{ IS } \text{🌳}$


- Calculate reachability-through-forest from home 
- Each region contains mountains, forest, or ocean
- Oceans aren't next to mountains





FCLP: So, for the reachability constraint, we can basically use the old datalog program, we just need to tweak the syntax as we described before. But for the next step, prepare yourself to *enter the unknown*.

[SAT and DATALOG steel themselves]

`reach()`


`reach(r_2) :- reach(r_1), next_to(r_1, r_2), terrain(r_2) IS `


`terrain(r) IS { , ,  } :- region(r)`

- Calculate reachability-through-forest from home 
- Each region contains mountains, forest, or ocean 
- Oceans aren't next to mountains






FCLP: In finite-choice logic programming we can represent the second constraint with a rule that looks like this third line of code here. So, for any region R, there's a **free choice** between the region having mountains, forests, or oceans as their terrain. This rule **forces** each region to have some terrain type, and requires that it's one of the three we expect.

reach()

reach(r_2) :- reach(r_1), next_to(r_1, r_2), terrain(r_2) IS 

terrain(r) IS { , ,  } :- region(r)

terrain(r_2) IS { ,  } :- terrain(r_1) IS , next_to(r_1, r_2)

- Calculate reachability-through-forest from home 
- Each region contains mountains, forest, or ocean 
- Oceans aren't next to mountains 


$\forall r_1, r_2 \in R. (\text{ocean}(r_1) \wedge \text{next_to}(r_1, r_2))$
 $\supset (\text{forest}(r_2) \vee \text{ocean}(r_2))$




FCLP: Finally, for the constraint that oceans aren't next to mountains, let's think about how we did it in SAT. We said that a region next to the ocean must be either forest or ocean, and we'll do something similar in our finite-choice logic program, where we can say, well, like every rule in a Datalog-like language, it forces the conclusion to apply if the premises apply. This will make it the case that ocean-adjacent regions can't have mountains.

DATALOG: Okay, I haven't seen functional dependencies used that way before.

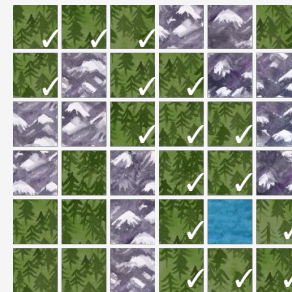
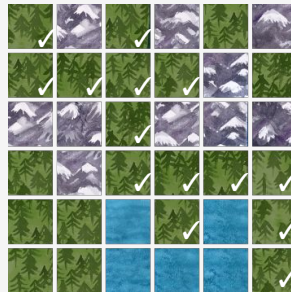
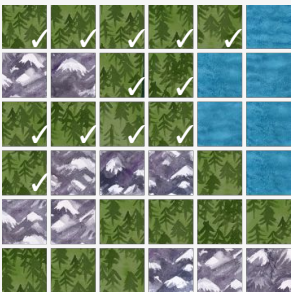
FCLP: Right.

reach()

reach(r_2) :- reach(r_1), next_to(r_1, r_2), terrain(r_2) IS 

terrain(r) IS { , ,  } :- region(r)

terrain(r_2) IS { ,  } :- terrain(r_1) IS , next_to(r_1, r_2)



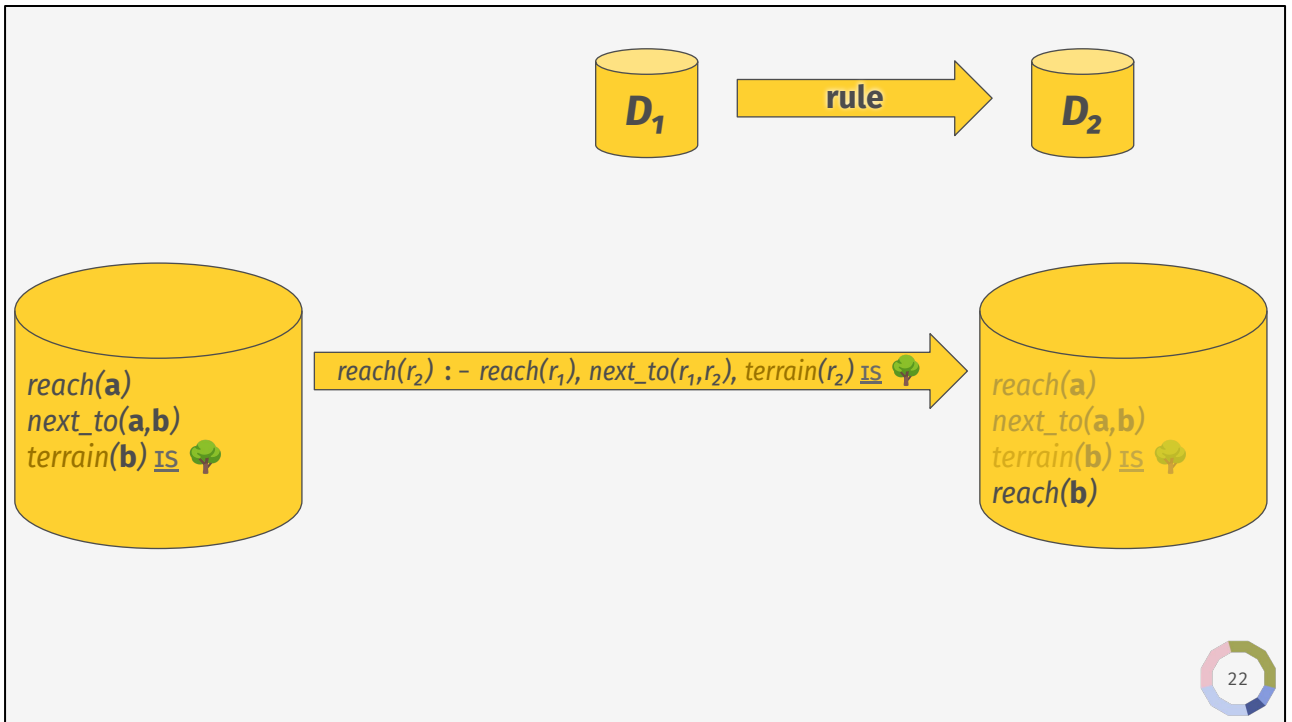
FCLP: Nonetheless, these four rules represent this possibility space we wanted of tile grids where oceans aren't next to mountains, and it correctly represents reachability from a starting location without the unjustified inferences.

SAT: That's pretty cool, but how do you precisely define the meaning of one of these finite-choice logic programs? I mean in SAT it's pretty simple: any assignment of "true" or "false" to propositions that makes the formula true is a valid meaning.

[SAT Turns to DATALOG]

SAT: For that matter, how does Datalog define the meaning of a program?

DATALOG: It's not *that* simple, but I'd still say it's pretty straightforward.



DATALOG: In Datalog, everything that we're doing is in terms of monotonic functions that grow our **database**, the set of facts that we're gathering.

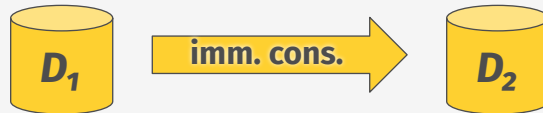
Every rule is a function that can grow the database: if the premises are satisfied, then we can add the conclusion. For example, the transitive reachability rule is a function taking a database with a couple of facts – two regions are next to each other, one's reachable, the other's forest – and outputting a database that also contains the fact that the other one is reachable.

Informally, the meaning of a Datalog program is "run the rules until you can't run the rules anymore," but formally, there's a pretty well-understood recipe.

1. Interpret each rule as a function



2. Join the output of ALL rules



3. Show imm. cons. is monotonic

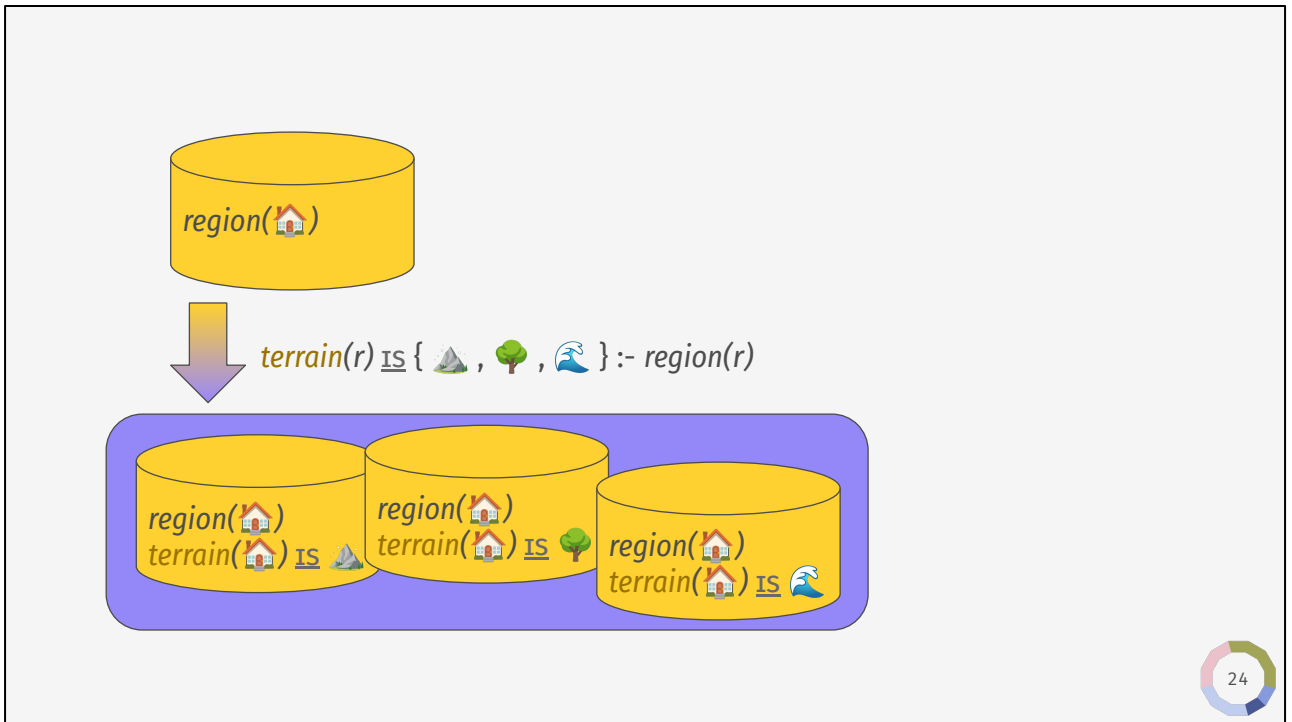
4. Observe we have a complete lattice

∴ The least fixed point of **imm. cons.** is uniquely defined (Knaster-Tarski)



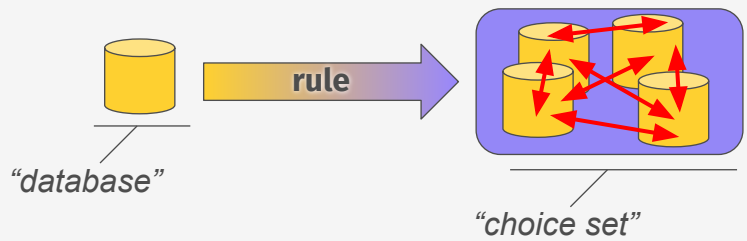
DATALOG: Once you explain what one rule does, you join the result of *all* of the rules together: this is a function from databases to databases called **immediate consequence**.

Then, you need to show that the immediate consequence function is monotonic under set inclusion. Because sets ordered in this way are a complete lattice, the Knaster-Tarski theorem gives us a least fixed point, and that least fixed point is the meaning of the Datalog program.



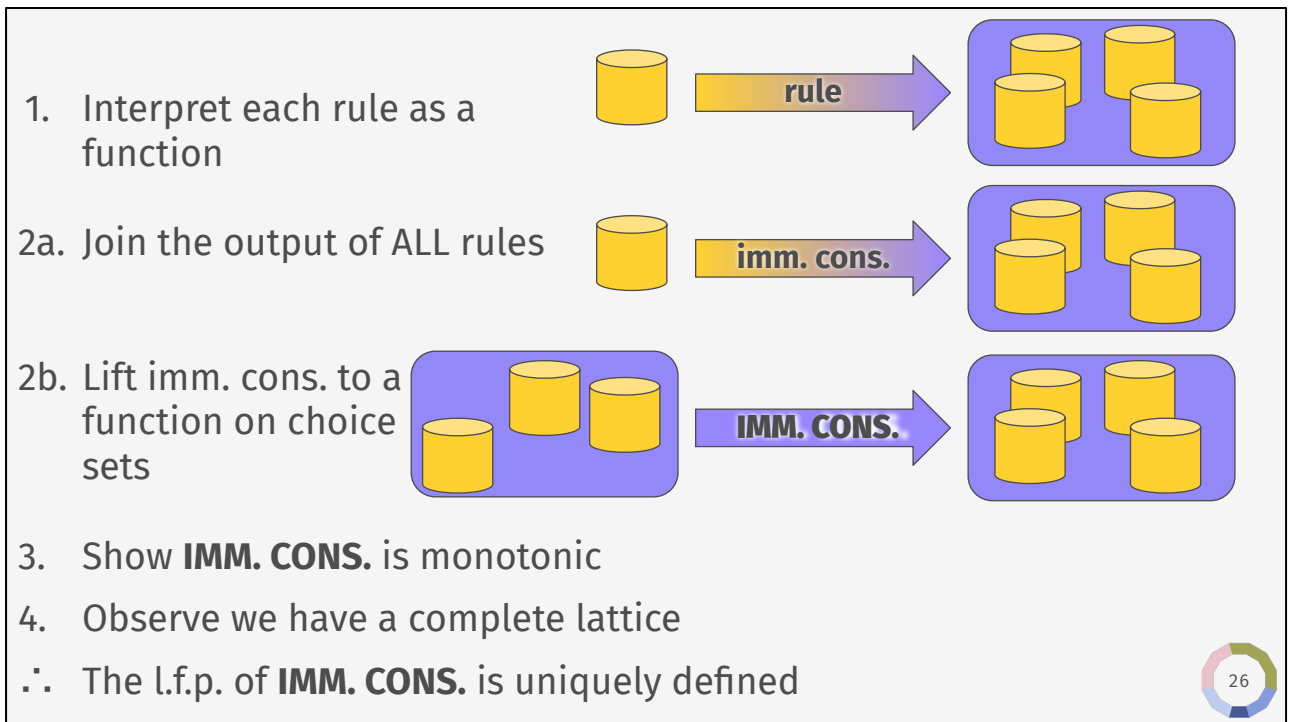
DATALOG: But in this case a rule with choices seems like it creates a whole set of possible successor databases, so I don't know what it means to iterate this a fixed point. The domain and codomain aren't even the same thing!

FCLP: Right. Yeah, so this is where things get fun.



FCLP: You're on the right track with observing that applying a rule can be thought of as function from one database to a set of databases. And in fact, they're actually sets of *mutually incompatible* databases, which means that any two databases will always disagree on at least one of their attributes.

This structure is something important for our definition and our semantics: we call it a **choice set**. And, equipped with this mathematical definition, we can actually follow more or less the same recipe Datalog uses. We'll just need one extra step.



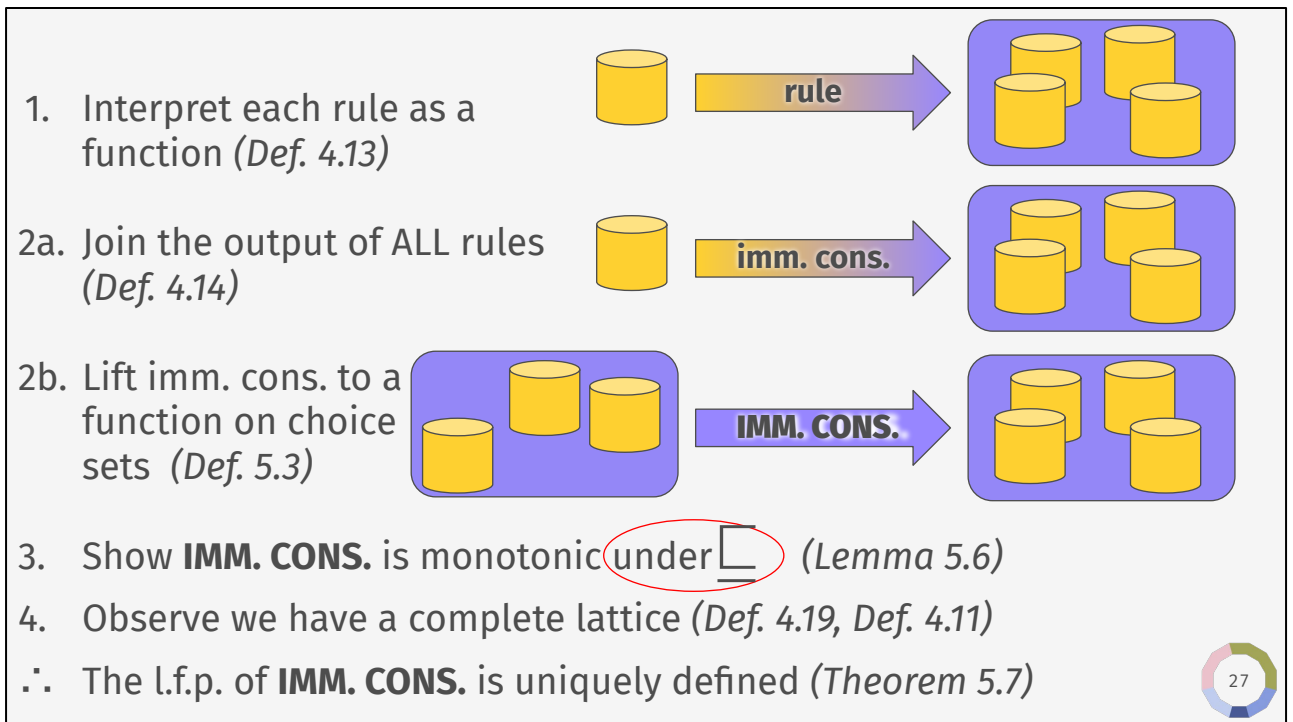
FCLP: Okay. So, adapting this recipe.

First, we're going to interpret each rule in a finite-choice logic program as a function from a database to a set of databases: a choice set.

Then, we'll join the output of all rules using an appropriate definition of "join," and that will produce our immediate consequence operator from a database to a choice set.

And then, the one extra step we need is just to lift that immediate consequence operator to be a function on choice sets so that we have the domain and codomain as the same.

And once we do that, we need to show that this immediate consequence operator is monotonic, observe we have a complete lattice, and the least fixed point of this immediate consequence function is the meaning of a finite-choice logic program.



[DATALOG boggles]

DATALOG: Okay. So. I recognize the pattern here... but... I have questions.

FCLP: That's understandable. All the gory details, of course, are in the paper.

But maybe just to give you a flavor of it, I'll explain a key piece here: the ordering that makes choice sets into a complete lattice.

DATALOG: Okay. So in datalog, we gain information and we grow our database by adding new facts. How do you going to gain information and grow a choice set?

Three ways to grow a choice set

1. Learn more information

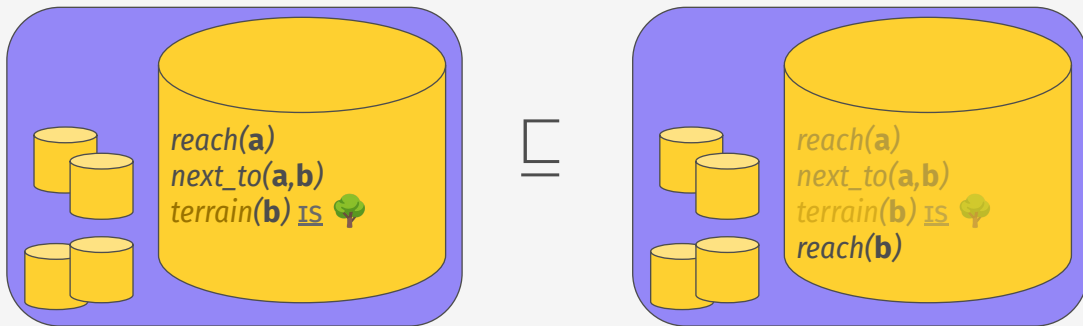


FCLP: Great question. So there's three ways.

There's the Datalog way in which it's possible for a database to have more information than another database by virtue of being a superset.

Three ways to grow a choice set

1. Learn more information

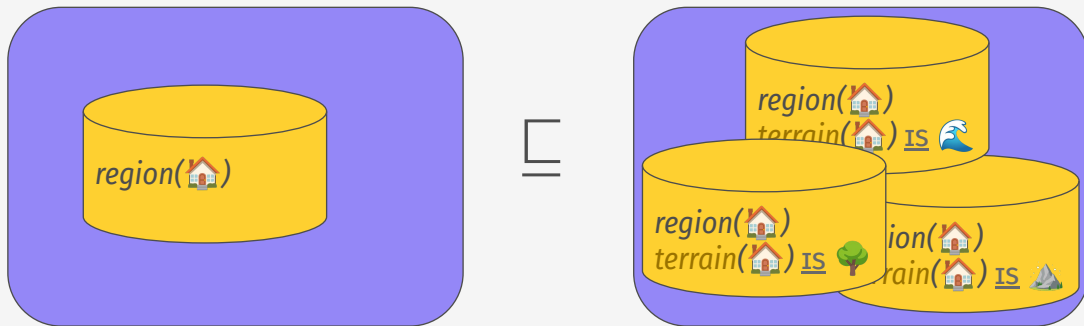


FCLP: If we embed those databases in otherwise-similar choice sets, then we can say we're also adding information to the choice set.

Another way to think of it is that we replace one database in the smaller choice set with a single successor database in the larger choice set.

Three ways to grow a choice set

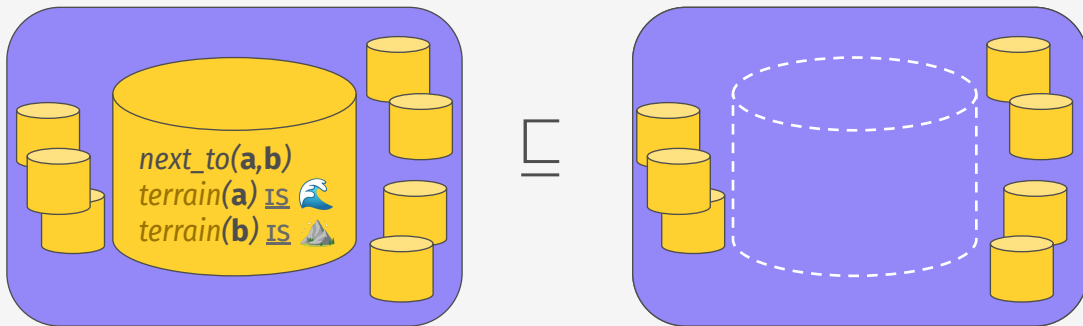
1. Learn more information
2. Split into multiple possibilities



FCLP: The second way we can learn information is by splitting into multiple possibilities. When we freely choose a terrain type for a region, we are adding information by replacing a single database with multiple, mutually incompatible, successors.

Three ways to grow a choice set

1. Learn more information
2. Split into multiple possibilities
3. Eliminate possibilities



FCLP: And then finally, we can actually gain information by removing a database.

So if it violates a constraint – say, by putting mountains next to oceans – then you can learn more information by removing that database. In other words, you can replace this database with zero successors.

Three ways to grow a choice set

1. Learn more information
2. Split into multiple possibilities
3. Eliminate possibilities

Definition 4.9, part 1:

$$C_1 \sqsubseteq C_2$$

if and only if

for all $D_2 \in C_2$, there exists a $D_1 \in C_1$ such that $D_1 \subseteq D_2$



FCLP: If we put all these together into a formal definition, we get something with kind of a contravariant flavor to it. So, a choice set C_1 is less than the choice set C_2 if, for every database in the greater choice set, there is some smaller database in the lesser choice set C_1 .

Three ways to grow a choice set

1. Learn more information
2. Split into multiple possibilities
3. Eliminate possibilities

Definition 4.9, part 1 (Smyth ordering on choice sets):

$$C_1 \sqsubseteq C_2$$

if and only if

for all $D_2 \in C_2$, there exists a $D_1 \in C_1$ such that $D_1 \subseteq D_2$



FCLP: If any of you are fans and familiars of domain theory, this might look familiar to you as the “Smyth” ordering on powerdomains, which is a common model of nondeterministic programming!

Finite-Choice Logic Programming

- Like SAT, allows mutually-exclusive choices
- Like Datalog, makes only justified inferences
- Like Datalog, has a least-fixed-point semantics



FCLP: So. In summary, the model of finite-choice logic programming combines the mutually-exclusive choices of SAT with the justified inferences and least-fixed-point semantics of datalog.

DATALOG: Great! And so all this, like, exists?!!!

Finite-Choice Logic Programming

- Like SAT, allows mutually-exclusive choices
- Like Datalog, makes only justified inferences
- Like Datalog, has a least-fixed-point semantics



<https://dusa.rocks/>

The screenshot shows the Dusa web interface. On the left, a code editor displays a Prolog program for map generation:

```
1 # Map generation
2
3 reach 🏠.
4 reach R2 :-
5     reach R1,
6     next_to R1 R2,
7     terrain R2 is "🌲".
8 terrain R is {"🏠", "🌲", "🌊"} :-
9     re R.
10 terrain R2 is {"🌲", "🌊"} :-
11     terrain R1 is "🌊",
12     next_to R1 R2.
13
```

On the right, the execution results are shown, including a message "program changed! reload?" and a list of deduced facts:

- reach 🏠
- reach (coord 1 1)
- reach (coord 1 2)
- reach (coord 2 1)
- reach (coord 3 1)
- reach (coord 3 2)
- reach (coord 4 2)
- reach (coord 4 3)
- reach (coord 4 4)
- reach (coord 5 2)
- terrain 🏠 is "🌊"
- terrain (coord 1 1) is "🌲"

At the bottom, the status bar indicates: "Pause running, 704172 deductions, 270805 choices, 55077 dead ends". A circular progress indicator in the bottom right corner shows the number 35.

FCLP: Yeah, actually! We have an implementation called Dusa which you can try out at dusa-dot-rocks. It's pretty good!

[DATALOG is jubilant!!!!1!!!one!!!]

DATALOG: This is amazing! Surely nothing will ever go wrong!

INTRODUCING
RONALD GARCIA
as ANSWER SET
PROGRAMMING

36

ASP: PARDON ME. I HAVE MORE OF A COMMENT THAN A QUESTION.

You see, Answer Set Programming handles this combination of forced deduction and free choice that you've been describing. What's the connection?

FCLP: So, this is a great observation. Everyone, I'd like to introduce you to Answer Set Programming, an old friend of mine who I've actually learned a lot from. In fact, one of my original motivations was understanding ASP better.

Finite-Choice Logic Programming and ASP

- With open rules for default reasoning, finite-choice logic programming can interpret ASP



FCLP: Let me give you at least a kind of first answer here.

We have another kind of rule in the paper that we haven't talked about that permitting a kind of "default reasoning." With that addition, we can actually translate all answer set programs to finite-choice logic programs.

ASP: Okay. No offense, but why not just translate your finite-choice logic programs into answer set programming? Are you giving us anything new?

Finite-Choice Logic Programming and ASP

- With open rules for default reasoning, finite-choice logic programming can interpret ASP
- Finite-choice logic programming can work with *infinite* possibility spaces without a "grounding" step



FCLP: Okay, fair question. So, actually, yes!

One of the big limitations of the standard **stable model semantics** for answer set programming is that it only gives meaning to "ground" programs – those without any logic variables in them.

In practice, this actually means that conventional ASP solvers require a separate "grounding phase" before they can even start solving, so they'll actually only be able represent finite domains, and they can't handle possibility spaces with infinitely many solutions.

$$\begin{aligned} \text{run}(0) &\underline{\text{IS}} \{ \text{stop}, \text{run} \} \\ \text{run}(n+1) &\underline{\text{IS}} \{ \text{stop}, \text{run} \} \text{ :- } \text{run}(n) \underline{\text{IS}} \text{run} \end{aligned}$$

Finite-Choice Logic Programming and ASP

- With open rules for default reasoning, finite-choice logic programming can interpret ASP
- Finite-choice logic programming can work with *infinite* possibility spaces without a "grounding" step

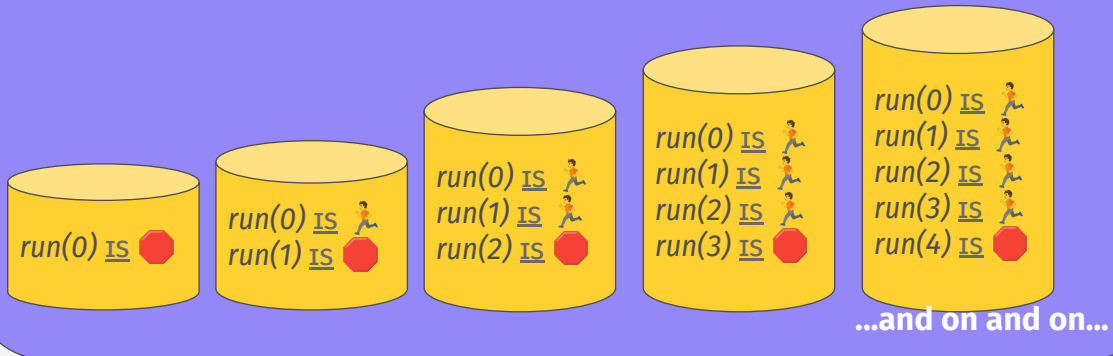


FCLP: To illustrate this, let me give you the simplest possible example I can think of.

At index 0 of this program, we will make a free choice between "stop" and "continue."

And then, if we've made the choice to continue at index n, then at index n-plus-one, we'll make that choice again.

$$\text{run}(0) \text{ IS } \{ \text{stop}, \text{run} \}$$

$$\text{run}(n+1) \text{ IS } \{ \text{stop}, \text{run} \} \text{ :- } \text{run}(n) \text{ IS } \text{run}$$


FCLP: The meaning according the semantics that we've defined contains infinitely many possibilities, one for each (co)natural number, and our Dusa implementation will actually enumerate these solutions productively.

But, given a similar answer set program, a conventional ASP solver would have to count to infinity before would return solutions, so of course, it won't.

$$\begin{aligned} \text{run}(0) &\underline{\text{IS}} \{ \text{stop sign}, \text{runner} \} \\ \text{run}(n+1) &\underline{\text{IS}} \{ \text{stop sign}, \text{runner} \} \text{ :- } \text{run}(n) \underline{\text{IS}} \text{runner} \end{aligned}$$

Finite-Choice Logic Programming and ASP

- With open rules for default reasoning, finite-choice logic programming can interpret ASP
- Finite-choice logic programming can work with *infinite* possibility spaces without a "grounding" step (like lazy-grounding ASP)



FCLP: There has been work on something called "lazy answer set programming" that can handle these situations. But, it's, like, treated as an additional complication on top of answer set programming; whereas, we found – kind of excitingly – when we started with this more a direct denotational semantics, the meaning of infinite possibility spaces emerges naturally.

$$\begin{aligned} \text{run}(0) &\underline{\text{IS}} \{ \text{stop sign}, \text{runner} \} \\ \text{run}(n+1) &\underline{\text{IS}} \{ \text{stop sign}, \text{runner} \} \text{ :- } \text{run}(n) \underline{\text{IS}} \text{runner} \end{aligned}$$

Finite-Choice Logic Programming and ASP

- With open rules for default reasoning, finite-choice logic programming can interpret ASP
- Finite-choice logic programming can work with *infinite* possibility spaces without a "grounding" step (like lazy-grounding ASP)
 - **Spaces of inductively defined datatypes are infinite possibility spaces**



FCLP: This stop-and-go example is, you know, the simplest thing I could think of, but I'm excited about it because I think it can actually scale to the task of, say, enumerating algebraic datatypes, which is something you can't easily do in ASP.

So, I hope that answers some of your concerns, perhaps.

[Thumbs up from ASP]

Finite-Choice Logic Programming

- Like SAT, allows mutually-exclusive choices & multiple solutions
- Like Datalog, makes only justified inferences
- Least-fixed-point semantics in terms of choice sets
- Implementation at <https://dusa.rocks>

Also in the paper

- Many examples
- Nondeterministic algorithm & correctness proof
- McAllester-style prefix-firing cost semantics
- Performance comparison with multiple ASP implementations (graphs!)
- Open rules and ASP-to-finite-choice-logic-programming translation



FCLP: Okay. So now, for real, in conclusion: finite-choice logic programming unifies the strengths of Datalog and Boolean Satisfiability, generalizes answer set programming, and enjoys a least-fixed point semantics in terms of these "choice set" structures. We're excited about this approach to representing and exploring "possibility spaces", or sets of possible worlds defined by a shared set of rules that produce them.

I hope we gave you a few reasons why finite-choice logic programming might be worth learning more about. And if you like more math, examples, experiment data, and so on, you can find it in [the paper](#). If you prefer hand-drawn pictures and examples and puzzles, [ask](#) for a [zine](#).

And that's it! Thanks very much.

[QR code links to dusa.rocks/docs/pop1]