2024-05-31-Relational Programming Simple Use-Case

Type Checking vs. Declarative (Relational) Programming

When you declare something in Haskell, you are making a statement about consistency criteria in your code. You still have to write the code to perform the actions. The type checker doesn't do any coding for you, it only checks to see that what you wrote is self-consistent.

When you declare rules in Prolog, you are asking the Prolog engine to find certain relationships in the user's data. You don't write code. The engine performs the actions. The engine does the searching - you don't need to write code telling it how to search.

Prolog declarations are “code” sent to the run-time Prolog engine. The engine searches run-time (end-user) data based on the declarations. The programmer only declares the relationships that must be satisfied, but, does not specify how the matching must proceed[1]. The engine executes the search - also called “inferencing” - at runtime on runtime data. The Prolog engine performs backtracking and can (on request) determine every possible set of matches for a set of relations.

Haskell, on the other hand, does no backtracking at runtime[2]. Declarations relate only to programmers’ code, not to the end-users’ data. The type-checker only checks - without backtracking - whether all of the gears mesh properly, as specified by the programmer. If a type error is encountered, the programmer must rewrite / repair the code, the type checker does not retry or find other possible combinations. Programmers’ code is converted to low-level assembler operations which are interpreted by CPU hardware to mimic mathematical functional notation, instead of controlling a more elaborate engine. CPU hardware uses non-reentrant *subroutines.* Haskell uses only a subset of assembler operations - plus some extra code for implementing reentrancy - to fake out mathematical notation.

It is possible to write relations and pattern-matching in Haskell, but, the programmer must resort to explicitly specifying the operational details using

[1] The *cut* operator was added to Prolog, though, to direct the actions of the engine and to cut off searches which are known to be dead-ends, i.e. the *cut* operator is a way to insert search optimizations into the declarative code.

[2] and, little, if any, backtracking at compile time

loops, recursion, data structures, continuation-passing, etc. The type checker only checks whether the operational details mesh together in a consistent manner, but, does not provide an engine that does the work of looping, recurring, control flow, etc. In essence, programmers must write all of the code to implement a relational engine using Haskell, while Prolog provides the engine[3].

[3] You can think of the relational engine as a runtime library that is scripted / commanded by Prolog code ("declarations"). Analogously, "C" compilers use a runtime library - `crt0`, for performing initialization at runtime (like zeroing out parts of memory, writing values into initiazlized variables, etc).

Minimal Example

The following is a minimal example Prolog code[4]...

```
        president(washington).
        president(lincoln).
        eq(X,Y) :- X = Y.
        neq(X,Y) :- eq(X,Y),!,fail.
        neq(_,_).

        goals(A,B) :-
            president(A),
            president(B),
            neq(A,B).

        goals_no_neq(A,B):-
            president(A),
            president(B).
```

In the above, the first two lines create a factbase (database) of 2 facts, hard-coding the names of 2 presidents.

The next 3 lines create Prolog *rules* for searching the factbase (`eq`, `neg` (*version 1*), and, `neg` (*version 2*)). The way that the rules are written, when the first version of `neg` fails, Prolog tries the second version of `neg`. The second `neg` is written to always succeed. Also, when you ask Prolog to retry (backtrack), Prolog tries every version of every rule.

On the other hand, when the first `neg` finds that X is equal to Y, Prolog proceeds and makes a *cut* (!) which tells the backtracker that this branch is final and to not backtrack over it. Then, Prolog proceeds and executes `fail`. Because of the *cut*, the second `neg` is not tried and the `neg` rule fails completely.

Prolog programmers would actually write the `eq` rule as

```
        eq(X,X).
```

stating that rule `eq` succeeds if the first argument is the same as the second argument.

[4] You can run this code in Prolog at the command line. Suggested: save this code in a file called "neg.pl". Then run `swipl` as shown below (you need to ensure that `swipl` is installed).

The rest of the lines of code create queries (‘`:-`’). The query names are `goals`, and, `goals_no_neg`.

The only difference between `goals` and `goals_no_neg` is the 3rd line, `neg(A,B)`, in `goals`.

When Prolog executes the `goals` rule, it proceeds sequentially, picking some president as A, then picking some president as B - maybe the same as A, maybe different from A. When it finally hits the last line `neg(A,B)`, Prolog calls the neg rule (above). If A and B are the same, the `neg` rule fails which makes the `goals` rule fail. If A and B are not the same, the 2nd version of the `neg` rule is invoked. It always succeeds, hence, the `goals` rule succeeds.

Running this code at the command line[5], we see:

```
$ swipl neg.pl
...
?- goals(P,Q).
P = washington,
Q = lincoln ;
P = lincoln,
Q = washington ;
false.

?- goals_no_neq(P,Q).
P = Q, Q = washington ;
P = washington,
Q = lincoln ;
P = lincoln,
Q = washington ;
P = Q, Q = lincoln.

?-
```

Note that *;* tells Prolog to back up and try again.

[5] This is how to run `swipl` on the above code, saved in `neg.pl`.

Note that the `goals_no_neg` rule prints out a successful match where, both, P and Q are `washington`, and, where, both P and Q are `lincoln`. The 3rd line (`neg(A,B)`) in `goals` weeds these matches out of the results from the `goals` rule.

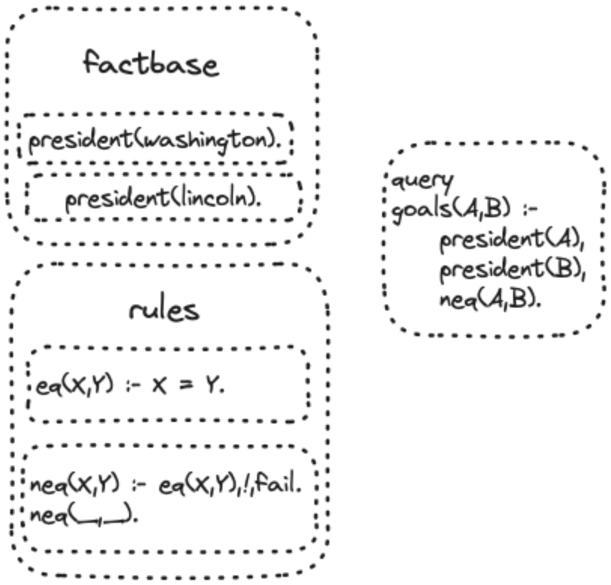
Note, also, that the backtracker is very thorough and produces what looks like 2 similar results `washington` and `lincoln`, and, `lincoln` and `washington`. The results are, technically, different. If you want to weed out this kind of apparent duplication, you need to add more rules[6].

Note, also, that case is significant. Any identifier that begins with an upper case letter is deemed to be a *logic variable*. Everything else must begin with a lower case letter. Logic variables begin as empty slots which the Prolog engine is free to fill and mutate during searching and backtracking. Two logic variables with the same name, say `P`, represent the same slot. The Prolog engine must search for combinations of patterns where all values of the same logic variables are consistent.

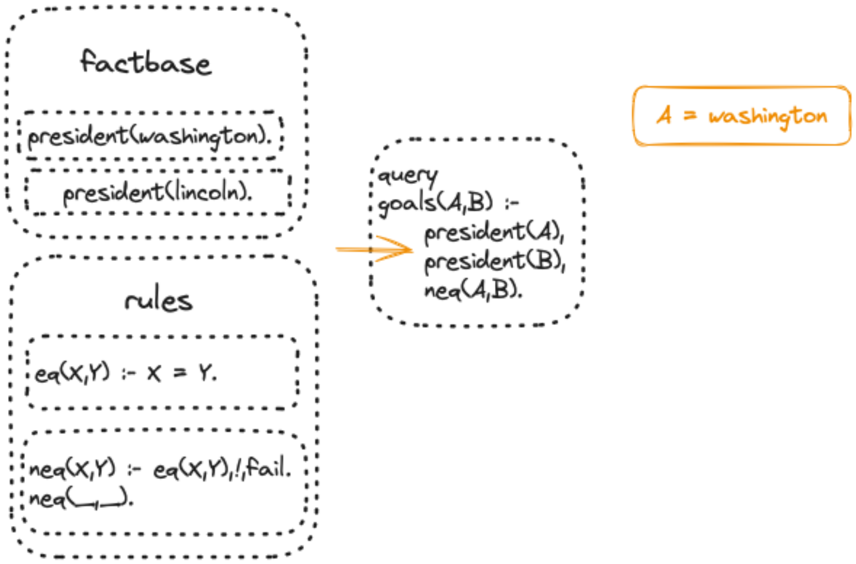
[6] One way is to keep a keep a current list of matches and to check against the list. Another way is to use *cut* judiciously.

Sketch of Backtracking

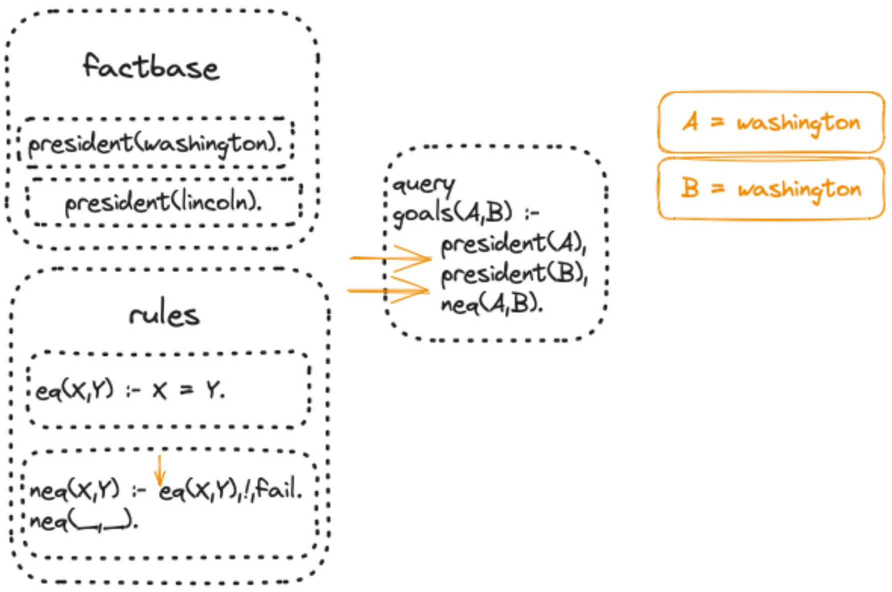
Factbase Initially



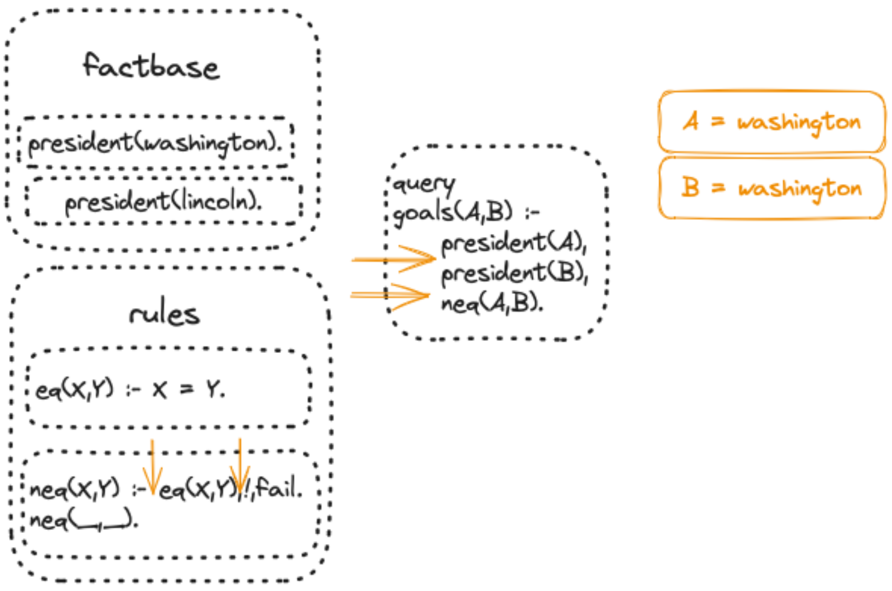
First Match



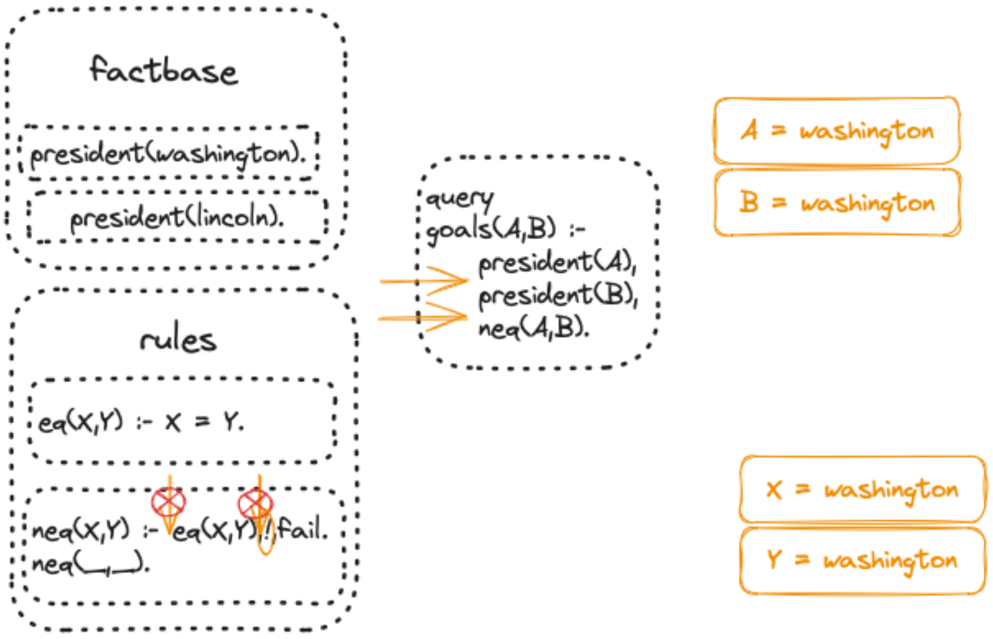
Second Match, And, Begin Rule Neq



Advance Rule Neq

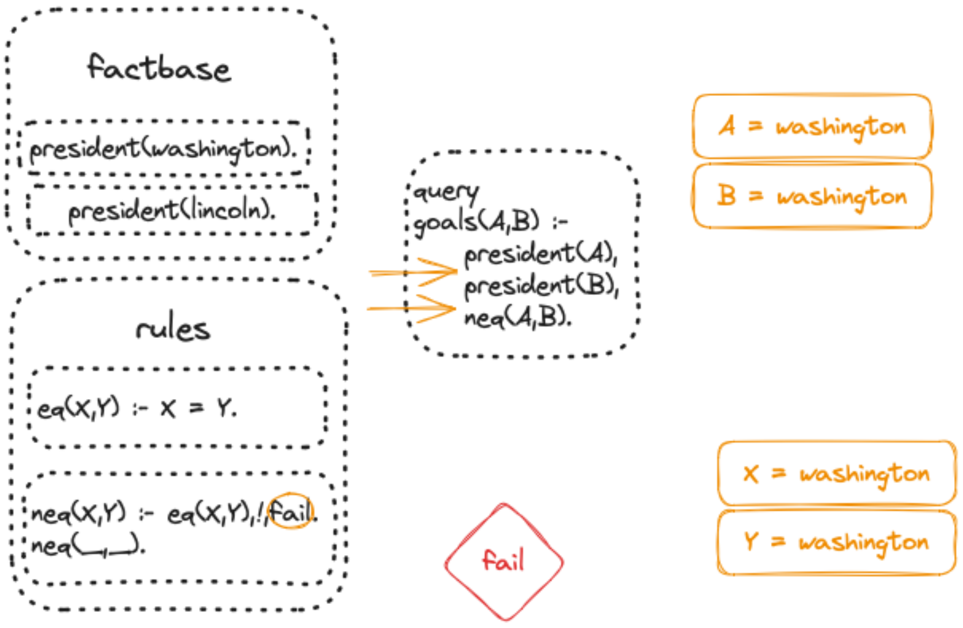


Advance Past Cut



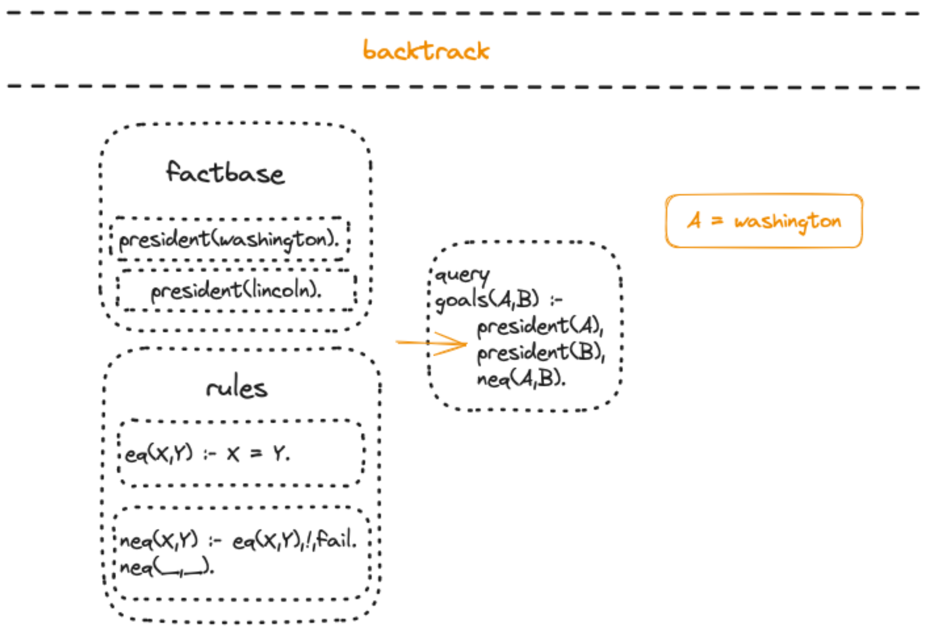
Fail in Rule Neq

No backtracking occurs within rule `neq` due to previous *cut*. Rule `neq` fails.



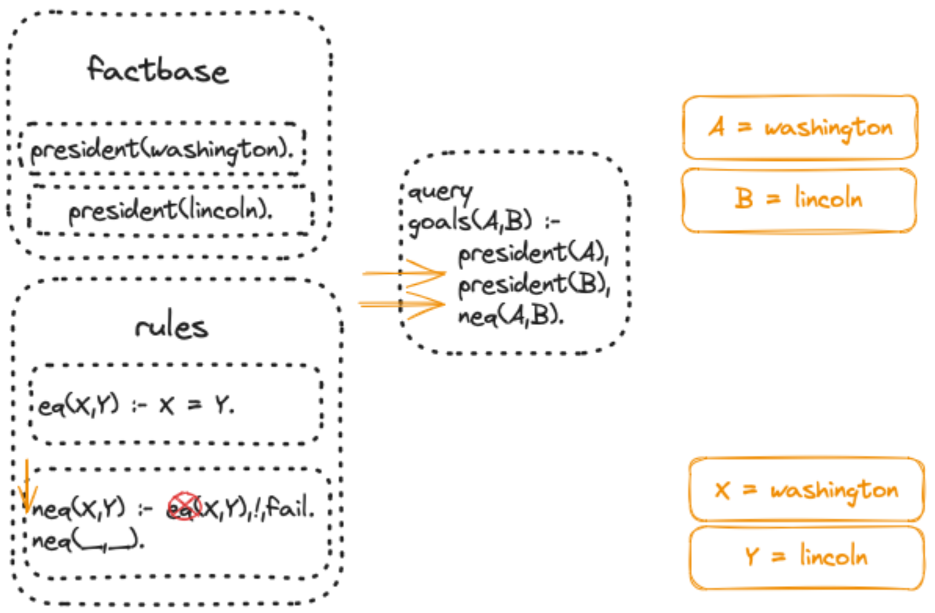
Backtrack

Backtrack to main query and try again.

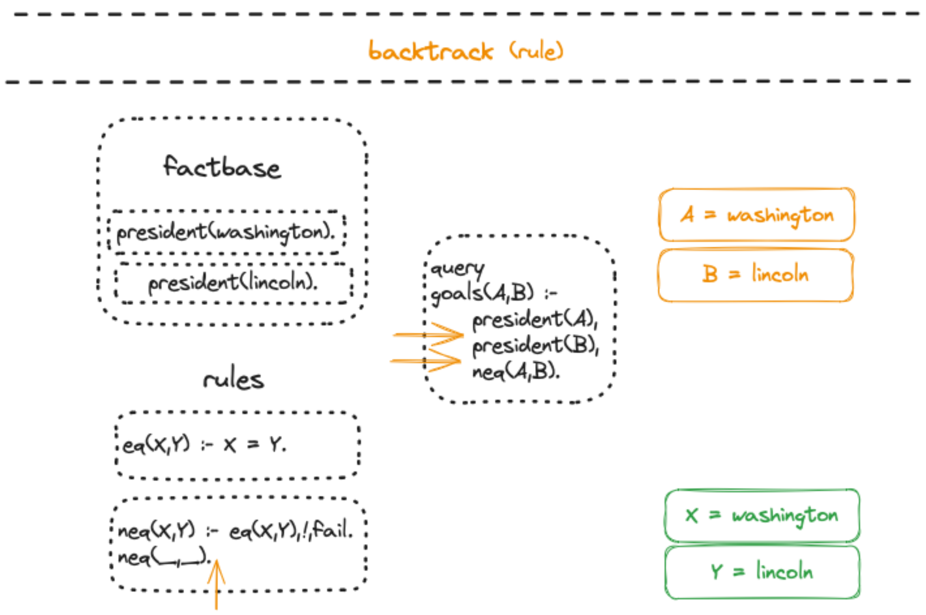


Choose Another Value for B, First Neq Fails Before Cut

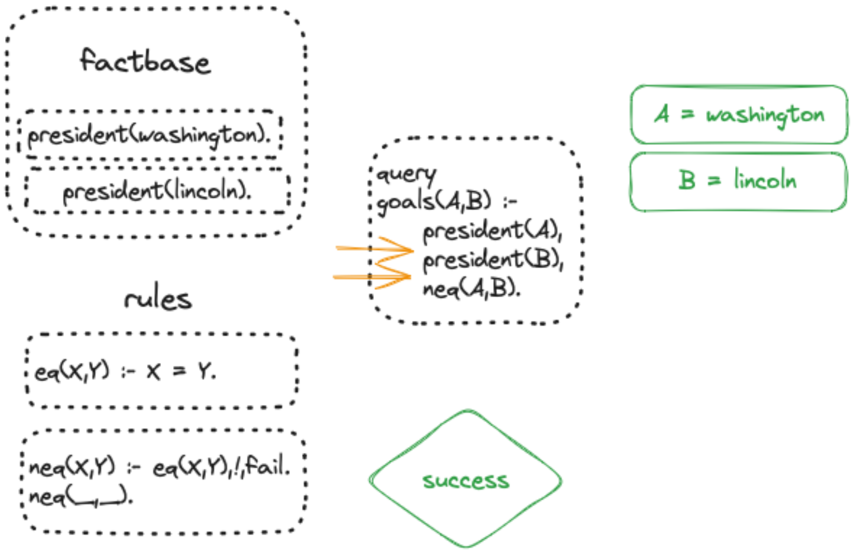
A and B are not equal, so the first version of rule `neq` fails...



Try the Second Version of Rule `neq`

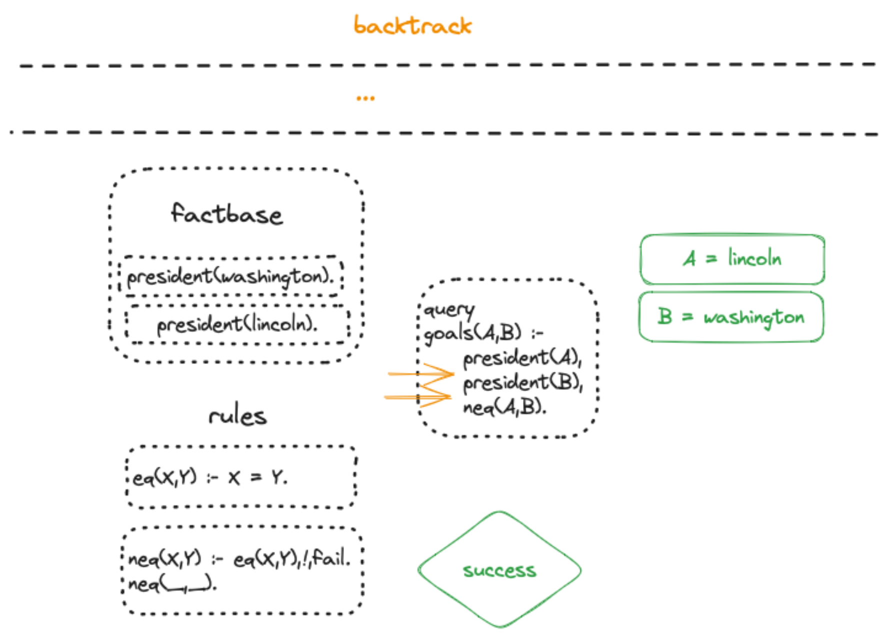


Top Query, `Goals`*, Succeeds*

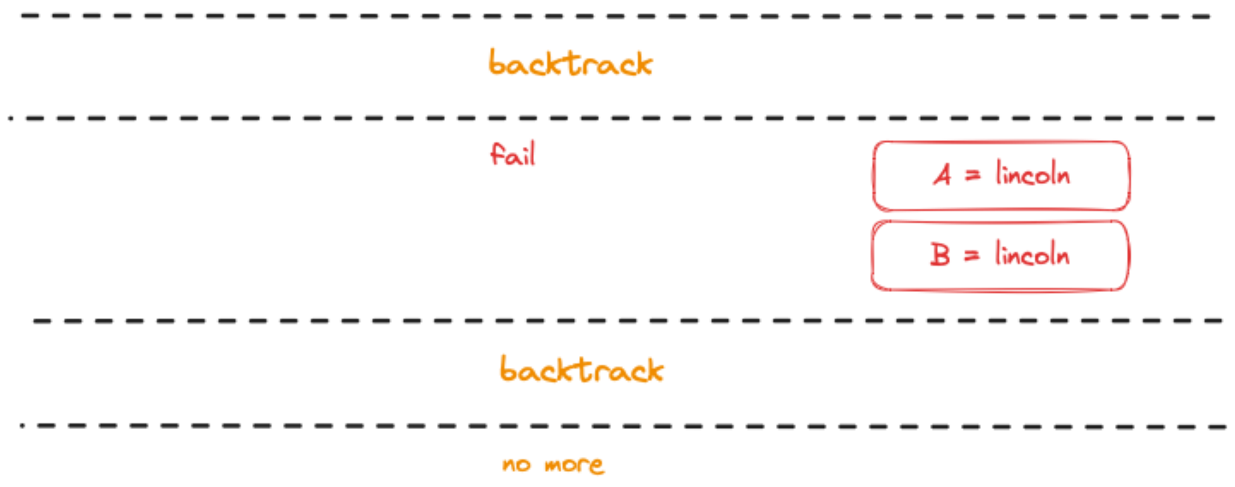


Another Success

Using A=`lincoln` and B=`washington` succeeds. This is similar to the first success, but, the rules as written, do not weed out this similarity.



More Retries Until There Are No More Possibilities



Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*