CHAPTER 13
Logic

13.1 Introduction

In the last few chapters we looked at propositional nets as an alternative to the Game Description Language for encoding games. In this and the following chapters we return to GDL as the typical language in which the rules of games are communicated in general game playing.

We first look at the task of computing logical consequences from a logic program. This provides the background for implementing the basic functionality of a general game player to generate legal moves, compute state update and decide termination from a GDL game description (see Chapter 4).

The ability to draw inferences from a logic program is also needed for converting a GDL input into a propnet or any other data structure that can be more efficient to compute with. We will show how logic can be used to find structure in GDL games such as symmetries, which help improve the performance of a general game-playing system. We will look into how single-player games can be solved in logic and how logic can be used to aid decision making through the automatic construction of a goal-priented evaluation function.

The basic type of inference we are concerned with can be formulated as *queries* that ask whether a literal L, or a conjunction of literals L_1 & L_2 & … & L_n, follows from a set of clauses. Often a query contains variables, and then we are interested in obtaining values for these variables under which the query becomes true.

Let's suppose that we want to infer a legal move for a player, say white, in a particular state of a game. We can formulate this as the query `legal(white,L)`, which means to determine for which `L`, if any, this is a logical consequence of the given rules.

Let's use as an example the GDL description of a 2-player game known as Nim. Players take turns removing one or more objects from one of several heaps. The game rules 1-10 below define a legal move as reducing the size of a selected heap to a smaller value. The facts in lines 12-15 encode a randomly chosen current state.

```
1   legal(P,reduce(X,N)) :-
2      true(control(P)) & true(heap(X,M)) & smaller(N,M)
3
4   smaller(X,Y) :- succ(X,Y)
5   smaller(X,Y) :- succ(X,Z) & smaller(Z,Y)
6   succ(0,1)
7   succ(1,2)
8   succ(2,3)
9   succ(3,4)
10  succ(4,5)
11
12  true(heap(a,2))
13  true(heap(b,0))
14  true(heap(c,5))
15  true(control(white))
```

Figure 13.1 - A collection of rules from the game Nim along with facts encoding a given state.

13.2 Unification

The most basic step in computing a query to a logic program is called *unification*. It means the process of replacing variables so that two logical expressions become similar. This is needed to determine which rule from the program could provide an answer to an atomic query such as `legal(white,L)`.

Generally speaking, a program fact *A*, or a program rule *A* `:-` B_1 & … & B_m, can only provide an answer to a query atom *L* if it is possible to replace the variables occurring in *L* and *A* in such a way that the two atoms become similar.

Definition 13.1 *A* substitution *is a finite set of replacements* $\{x_1/t_1, \ldots, x_n/t_n\}$ *such that*

- $n \geq 0$;
- $x_1, \ldots, x_n$ *are pairwise distinct variables; and*
- $t_1, \ldots, t_n$ *are terms (which may or may not contain variables).*

The result SUBST(E,σ) *of applying a substitution* $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ *to an expression E is obtained by simultaneously replacing in E every* x_i *by its replacement* t_i.

Definition 13.2 *Two expressions* E_1 *and* E_2 *are* unifiable *if we can find a substitution* σ *such that* SUBST(E_1,σ) = SUBST(E_2,σ). *Any such* σ *is called a* unifier *of* E_1 *and* E_2.

Recalling the example from above, the atomic query `legal(white,L)` and the head of the first rule in Figure 13.1, `legal(P,reduce(X,N))`, are unifiable. Indeed there are many different unifiers for the two, including all of the following.

σ_1 = {`P/white,L/reduce(a,1),X/a,N/1`}:
  SUBST(`legal(white,L)`, σ_1) = `legal(white,reduce(a,1))`
  SUBST(`legal(P,reduce(X,N))`, σ_1) = `legal(white,reduce(a,1))`

σ_2 = {`P/white,L/reduce(X,X),N/X`}:
  SUBST(`legal(white,L)`, σ_2) = `legal(white,reduce(X,X))`
  SUBST(`legal(P,reduce(X,N))`, σ_2) = `legal(white,reduce(X,X))`

θ = {`P/white,L/reduce(X,N)`}:
  SUBST(`legal(white,L)`, θ) = `legal(white,reduce(X,N))`
  SUBST(`legal(P,reduce(X,N))`, θ) = `legal(white,reduce(X,N))`

Comparing the three substitutions, the last one, θ, appears to be the most general way of unifying our two atoms: While variable `L` is assigned a move of the form `reduce(X,N)` in all three cases, unifier σ_1 additionally fixes specific values for `X,N` whereas σ_2 binds the two variables together. But neither is necessary to unify `legal(white,L)` and `legal(P,reduce(X,N))`. This leads to Definition 13.3 below, which refers to the *composition* of two substitutions $\theta_1 \circ \theta_2$ as a substitution that, for any expression *E*, satisfies

$$\text{SUBST}(E,\theta_1 \circ \theta_2) = \text{SUBST}(\text{SUBST}(E,\theta_1), \theta_2)$$

Definition 13.3 *A unifier* θ *is* more general *than a unifier* σ *if there is a third substitution* τ *such that* $\theta \circ \tau = \sigma$.

A most general unifier *for expressions E_1 and E_2 is one that is more general than any other unifier of the two expressions.*

It follows that indeed our unifier θ from above is more general than both σ_1 and σ_2, as can be seen from

$$\sigma_1 = \theta \circ \{\texttt{X/a,N/1}\}$$
$$\sigma_2 = \theta \circ \{\texttt{N/X}\}$$

In fact, θ is a most general unifier for our example. Fortunately it is a known fact that whenever two atomic formulas are unifiable, then a most general unifier always exists and is unique up to variable permutation.

Moreover, most general unifiers can be easily computed. The recursive algorithm below takes as input two expressions and a partially computed unifier (which should be empty at the first function call). It is assumed that expressions as well as partial unifiers are encoded as lists. For example, the function call

```
unify([legal,white,L],[legal,P,[reduce,X,N]],nil)
```

returns the following most general unifier.

```
[[P,white],[L,[reduce,X,N]]]
```

The algorithm works by comparing the structure of the two expressions, argument by argument.

```
function unify (x,y,sigma)
 {if (x == y) {return sigma};
  if (is_variable(x)) {return unifyVariable(x,y,sigma)};
  if (is_variable(y)) {return unifyVariable(y,x,sigma)};
  if (! is_list(x))
     {if (is_list(y) || x != y) {return fail}
      else {return sigma}};
  if (! is_list(y)) {return fail};
  sigma = unify(head(x),head(y),sigma);
  if (sigma == fail) {return fail}
   else {return unify(tail(x),tail(y),sigma)}}

function unifyVariable(x,y,sigma)
 {if (var z = replacement_for(x,sigma)) {return unify(z,y,sigma)};
  if (var z = replacement_for(y,sigma)) {return unify(x,z,sigma)};
  if (x != y && occurs_in(x,y)) {return fail}
   else {return add_element([x,y],sigma)}}
```

Function `replacement_for(x,sigma)` is assumed to return the replacement for variable `x` according to unifier `sigma` if such a replacement exists; and `0` otherwise. Function `occurs_in(x,y)` should return true just in case variable `x` occurs anywhere inside the (possibly nested) list `y`.

13.3 Derivation Steps (Without Negation)

Backward-chaining is the standard technique for query answering in clausal logic. For this we work our way backwards from the query, first by finding a rule that applies to the leading query element and then proving that the conditions of the rule hold.

A clausee is applicable to an atom that can be unified with the head of the rule. There is a small subtlety though. Suppose we had formulated the search for legal moves of our player as `legal(white,X)` instead of `legal(white,L)`. We would then be unable to unify the query atom with the head of the clause shown below.

```
legal(P,reduce(X,N)) :- true(control(P)) & true(heap(X,M)) &  smaller(N,M)
```

The reason is that no substitution σ can possibly exist such that SUBST(`legal(white,X)`, σ) = SUBST(`legal(P,reduce(X,N))`, σ) because `X` cannot be equated with `reduce(X,N)`.

But the scope of a variable should never extend beyond the clause in which it appears. For this reason, each application of a rule should be preceded by generating a "fresh" copy in which the variables have been consistently replaced by new names that do not occur in the original query or any other step in the same derivation.

For a single derivation step we then proceed as follows. Let's first consider the case that neither the query nor any of the program clauses contains a negated literal.

Given:	Clauses G	(without negation)
	Query L_1 & L_2 & … & L_n	(without negation), $n \geq 1$
Let:	A :- B_1 & B_2 & … & B_m	"fresh" variant of a clause in G, $m \geq 0$
	σ	most general unifier of a L_1 and A
Then:	L_1 & L_2 & … & $L_n \Rightarrow$ SUBST(B_1 & B_2 & … & B_m & L_2 & … & L_n, σ)	

A derivation step thus replaces the leading element of the query by the body of a suitable clause and applies the necessary variable bindings to all of the new query.

Here is an example that uses the first clause of Figure 13.1.

```
   legal(white,L)                  rule 1 with {P/white,L/reduce(X,N)}
⇒ true(control(white)) &
   true(heap(X,M)) & smaller(N,M)
```

When the applied clause is a fact, i.e., a rule with empty body, then the respective query element is removed without replacement, as in the following derivation step.

```
   true(heap(X,M)) & smaller(N,M)  rule 12 with {X/a,M/2}
⇒ smaller(N,2)
```

13.4 Derivations

A complete derivation for a query is obtained by the repeated application of derivation steps until all subgoals have been resolved. If the original query includes variables, then the *computed answer* is determined by the variable bindings made along the way.

Definition 13.4 *A series of derivation steps is called a* derivation. *A* successful *derivation is one that ends with the empty query, denoted as* "□".

The answer *computed by a successful derivation is obtained by composing the unifiers* $\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_k$ *of each step and restricting the result to the variables in the original query.*

An example is shown below, where again we use the clauses of Figure 13.1.

```
   legal(white,L)                  rule 1 with {P/white,L/reduce(X,N)}
```

⇒ `true(control(white)) &`	rule 15 with {}
`true(heap(X,M)) & smaller(N,M)`	
⇒ `true(heap(X,M)) & smaller(N,M)`	rule 12 with {`X/a,M/2`}
⇒ `smaller(N,2)`	copy of rule 4 with {`X'/N,Y'/2`}
⇒ `succ(N,2)`	rule 6 with {`N/1`}
⇒ □	

Answer: {`L/reduce(a,1)`}

Here is another successful derivation for the same query.

`legal(white,L)`	rule 1 with {`P/white,L/reduce(X,N)`}
⇒ `true(control(white)) &`	rule 15 with {}
`true(heap(X,M)) & smaller(N,M)`	
⇒ `true(heap(X,M)) & smaller(N,M)`	rule 12 with {`X/a,M/2`}
⇒ `smaller(N,2)`	copy of rule 5 with {`X'/N,Y'/2`}
⇒ `succ(N,Z') & smaller(Z',2)`	rule 6 with {`N/0,Z'/1`}
⇒ `smaller(1,2)`	copy of rule 4 with {`X''/1,Y''/2`}
⇒ `succ(1,2)`	rule 7 with {}
⇒ □	

Answer: {`L/reduce(a,0)`}

Derivations aren't always successful.

Definition 13.5 *A derivation* fails *if it leads to a query whose first element does not unify with the head of any available clause*.

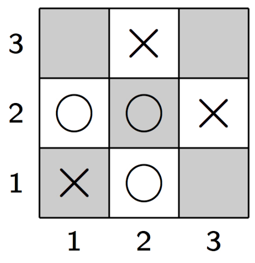
Here is an example of a derivation for the same query as above that leads to a dead-end.

`legal(white,L)`	rule 1 with {`P/white,L/reduce(X,N)`}
⇒ `true(control(white)) &`	rule 15 with {}
`true(heap(X,M)) & smaller(N,M)`	
⇒ `true(heap(X,M)) & smaller(N,M)`	rule 13 with {`X/b,M/0`}
⇒ `smaller(N,0)`	copy of rule 4 with {`X'/N,Y'/0`}
⇒ `succ(N,0)`	rule 6 with {`N/1`}
⇒ failure	

13.5 Derivation Tree Search

Some of the basic functions from Chapter 4 for a general game player require to consider all possible derivations of a query. For instance, `findlegals(`*role*`,`*state*`,`*description*`)` is expected to deliver every computable answer to `legal(`*role*`,M)` for a given state and game description. *Backtracking* provides a systematic way to compute these: After having found the first successful or failed derivation, one goes back to the most recent choice where a different clause could have been applied to the query, until all choices have been exhausted.

We can draw a tree with the original query as the root to illustrate the search space of backward-chaining. Each computed answer then corresponds to one branch that ends with the empty query. Switching to a different example, consider the problem of computing *all* legal moves for a player in a given Tic-Tac-Toe position. Let's use the clauses listed below in Figure 13.2. Specifically, the facts in lines 1-10 together encode the following game state, with white to move.



```
1  true(cell(1,1,x))
2  true(cell(1,2,o))
3  true(cell(1,3,b))
4  true(cell(2,1,o))
5  true(cell(2,2,o))
6  true(cell(2,3,x))
7  true(cell(3,1,b))
8  true(cell(3,2,x))
9  true(cell(3,3,b))
10 true(control(white))
11
12 legal(W,mark(X,Y)) :- true(cell(X,Y,b)) & true(control(W))
13 legal(white,noop) :- true(control(black))
14 legal(black,noop) :- true(control(white))
```

Figure 13.2 - The rules for legality in Tic-Tac-Toe along with facts encoding a given position.

The tree depicted in Figure 13.3 shows the 3 legal moves that can be computed for white in our example Tic-Tac-Toe position. The labels at the arcs indicate which clause has been selected. Backtracking can be implemented by a depth-first search through this tree.

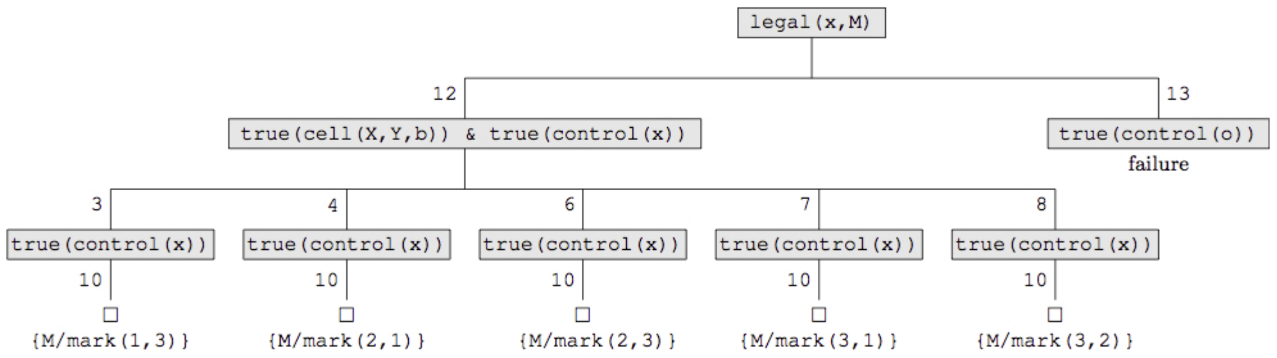


Figure 13.3 - Derivation tree to compute all legal moves for white from the clauses in Figure 13.2.

For black there is just one legal move in the same position, as the computation tree below illustrates.

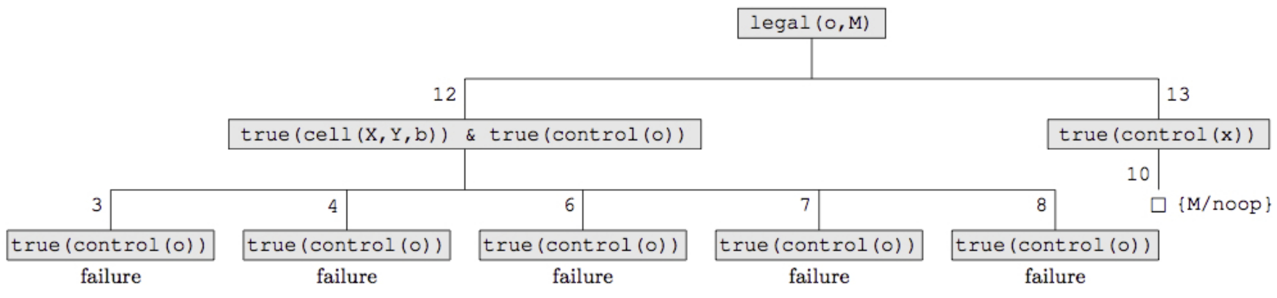


Figure 13.3 - Derivation tree to compute all legal moves for white from the clauses in Figure 13.2.

A correct implementation of a complete search through a derivation tree requires you to address the problem of potentially infinite derivations that may arise with some game descriptions. Even if a GDL describes a finite game, there is always the risk that a computation loops because of a recursive clause. Suppose, for example, the following rule were added to the game description of Nim in Figure 13.1.

```
16 smaller(X,Y) :- smaller(X,Y)
```

This clause is obviously redundant but perfectly correct, both logically and syntactically. A straightforward implementation to compute derivations with this rule will eventually enter a nonterminating loop.

```
   legal(white,L)                        rule 1 with {P/white,L/reduce(X,N)}
⇒ true(control(white)) &                rule 15 with {}
   true(heap(X,M)) & smaller(N,M)
⇒ true(heap(X,M)) & smaller(N,M)        rule 12 with {X/a,M/2}
⇒ smaller(N,2)                          copy of rule 16 with {X'/N,Y'/2}
⇒ smaller(N,2)                          ...
⇒ ...
```

Although it may be unlikely that you encounter such a tricky game description in practice, it can be a good idea to add a simple loop check to avoid nonterminating computations. Note that potential loops may be less obvious to detect from the GDL itself than in our example if they are spread over several clauses.

Recursive clauses may even lead to nonterminating derivations in which the query grows forever without repetition. To see why, consider the following alternative formalization of clause 5 from Figure 13.1.

```
5* smaller(X,Y) :- smaller(Z,Y) & succ(X,Y)
```

Logically, this means the very same thing, but it may give rise to a non-looping, infinite derivation as follows.

```
   legal(white,L)                                  rule 1 with {P/white,L/reduce(X,N)}
⇒ true(control(white)) &                          rule 15 with {}
   true(heap(X,M)) & smaller(N,M)
⇒ true(heap(X,M)) & smaller(N,M)                  rule 12 with {X/a,M/2}
⇒ smaller(N,2)                                    copy of rule 5* with {X'/N,Y'/2}
⇒ smaller(Z',2) & succ(N,Z')                      copy of rule 5* with {X''/Z',Y''/2}
⇒ smaller(Z'',2) & succ(Z',2) & succ(N,Z')        ...
⇒ ...
```

This problem can be avoided simply by reordering the atoms in a rule in such a way that recursive calls always come last.

13.6 Handling Negation

Let's now generalize our derivation procedure to queries and programs that include negative literals. A subgoal of the form *~A* can be treated according to a principle known as *negation-by-*

failure: If A itself cannot be derived, then $\sim A$ is true. This is justified by our use of the *minimal* model as the semantics of a set of GDL clauses.

For a single derivation step in the general case we thus proceed as follows.

Given: Clauses G
Query L_1 & L_2 & … & L_n $(n \geq 1)$

Case: $L_1 = A$ positive literal
• proceed as before

Case: $L_1 = \sim A$ negative literal without variables
• if all derivations for A fail: L_1 & L_2 & … & $L_n \Rightarrow L_2$ & … & L_n
• if a derivation for A succeeds: L_1 & L_2 & … & $L_n \Rightarrow$ failure

Recall, for example, one of the termination condition in Tic-Tac-Toe.

```
terminal :- ~open
open     :- true(cell(M,N,b))
```

As long as there is at least one blank cell in the current state, the sub-goal `true(cell(M,N,b))` has a successful derivation, and hence so has the atom `open`. Consequently, the negative sub-goal `~open` fails unless all cells have been marked. The query `terminal` therefore fails too, provided that no other terminating conditions hold that allow it to be derived.

But when a state is reached in which all cells have been marked, then `true(cell(M,N,b))` can no longer be derived, and hence `open` fails. Accordingly, `~open` now succeeds, and so does the query `terminal`.

The special, game-independent predicate `distinct` is computed very much like a negative subgoal.

Given: Query L_1 & L_2 & … & L_n $(n \geq 1)$

Case: $L_1 =$ `distinct`(t_1,t_2) without variables
• if t_1 and t_2 are syntactically different: L_1 & L_2 & … & $L_n \Rightarrow L_2$ & … & L_n
• if t_1 and t_2 are syntactically identical: L_1 & L_2 & … & $L_n \Rightarrow$ failure

It is crucial that only negative subgoals without variables can be subjected to the negation-by-failure computation. The same holds for the `distinct` relation. In some cases it may be necessary to reorder the subgoals to ensure that all variables have been replaced by non-variable terms before a negative literal gets selected. The GDL description of Multiple Buttons And Light (see Chapter 11) is a point in case. Consider the following encoding of a given position and move in that game along with one of the update rules.

```
true(p(1))
true(p(2))
true(p(3))
does(white,a(2))

next(p(X)) :- ~does(white,a(X)) & true(p(X))
```

Naturally we expect to be able to derive two answers to the query `next(p(Z))`, namely {`Z/1`} and {`Z/3`}, since the first and third light are not affected by the action. So let's consider the situation after the first step in a derivation.

```
   next(p(Z))                             rule 6 with {X/Z}
⇒  ~does(white,a(Z)) & true(p(Z))
```

But now if we were to ignore the restriction about negative literals with variables and chose the subgoals in the given order, then the query would fail. For the positive part of the negative literal, `does(white,a(Z))`, obviously has a successful derivation:

```
   does(white,a(Z))  rule 5 with {Z/2}
⇒  □
```

If, however, we reorder the subgoals so as to ensure that before we select a negated subgoal, all its variables have been substituted, then the two answers are obtained as expected.

```
   next(p(Z))                        rule 6 with {X/Z}
⇒  true(p(Z)) & ~does(white,a(Z))    rule 1 with {Z/1}
⇒  ~does(white,a(1))                 query does(white,a(1)) fails
⇒  □
```

```
   next(p(Z))                        rule 6 with {X/Z}
⇒  true(p(Z)) & ~does(white,a(Z))    rule 3 with {Z/3}
⇒  ~does(white,a(3))                 query does(white,a(3)) fails
⇒  □
```

Fortunately, the syntactic restrictions on valid GDLs guarantee that subgoals can always be reordered in this manner. Specifically, every variable in a rule must occur in a positive literal in the body, not counting the special game-independent predicate `distinct`. Hence you can always compute values for all variables in a clause before selecting a negative subgoal or an instance of `distinct`.

The negation-by-failure principle is recursively applied in case of nested negation, that is, when negative sub-goals arise during an attempted derivation for another negated literal. An illustrative example is given by the following excerpt from a fictitious game description.

```
1  role(red)
2  role(blue)
3  role(green)
4
5  true(free(blue))
6
7  trapped(R) :- role(R) & ~true(free(R))
8  goal(W,100) :- role(W) & ~trapped(W)
```

Only player `blue` can be shown to have achieved the goal in the current state, as per the nested derivation below.

```
   goal(blue,100)                 rule 8 with {W/blue}
⇒  role(blue) & ~trapped(blue)    rule 2
⇒  ~trapped(blue)                 sub-derivation †
⇒  □
```

†

`trapped(blue)`	rule 7 with {`R/blue`}
⇒ `role(blue) & ~true(free(blue))`	rule 2
⇒ `~true(free(blue))`	sub-derivation ‡
⇒ failure	

‡

`true(free(blue))`	rule 5
⇒ □	

The attempt to derive `goal(red,100)` fails.

`goal(red,100)`	rule 8 with {`W/red`}
⇒ `role(red) & ~trapped(red)`	rule 1
⇒ `~trapped(red)`	sub-derivation †
⇒ failure	

†

`trapped(red)`	rule 7 with {`R/red`}
⇒ `role(red) & ~true(free(red))`	rule 1
⇒ `~true(free(red))`	sub-derivation ‡
⇒ □	

‡

`true(free(red))`	
⇒ failure	

The same is true for `goal(green,100)`.

Exercises

1. The goal of this exercise is to unlock the mystery behind the single-player game shown below - and then to solve it.

```
role(you)

succ(1,2)
succ(2,3)
succ(3,4)
succ(4,5)
succ(5,6)
succ(6,7)
succ(7,8)

is_row(1)
is_row(Y) :-
  succ(X,Y)

base(step(1))
base(step(2))
base(step(3))
base(step(4))
base(step(5))
base(row(X,empty)) :-
```

```
  is_row(X)
base(row(X,one_coin)) :-
  is_row(X)
base(row(X,two_coins)) :-
  is_row(X)
input(you,jump(X,Y)) :-
  is_row(X) &
  is_row(Y)

init(step(1))
init(row(X,one_coin)) :-
  is_row(X)

babbage(X,Y) :-
  succ(X,Y)

babbage(X,Y) :-
  succ(X,Z) &
  true(row(Z,empty)) &
  babbage(Z,Y)

lovelace(X,Y) :-
  succ(X,Z) &
  true(row(Z,empty)) &
  lovelace(Z,Y)

lovelace(X,Y) :-
  succ(X,Z) &
  true(row(Z,one_coin)) &
  babbage(Z,Y)

turing(X,Y) :-
  succ(X,Z) &
  true(row(Z,empty)) &
  turing(Z,Y)

turing(X,Y) :-
  succ(X,Z) &
  true(row(Z,one_coin)) &
  lovelace(Z,Y)

turing(X,Y) :-
  succ(X,Z) &
  true(row(Z,two_coins)) &
  babbage(Z,Y)

legal(you,jump(X,Y)) :-
  true(row(X,one_coin)) &
  true(row(Y,one_coin)) &
  turing(X,Y)

legal(you,jump(X,Y)) :-
  true(row(X,one_coin)) &
  true(row(Y,one_coin)) &
  turing(Y,X)

next(row(X,empty)) :-
  does(you,jump(X,Y))

next(row(Y,two_coins)) :-
  does(you,jump(X,Y))

next(row(X,C)) :-
  true(row(X,C)) &
  does(you,jump(Y,Z)) &
  distinct(X,Y) &
  distinct(X,Z)

next(step(Y)) :-
```

```
  true(step(X)) &
  succ(X,Y)

terminal :-
  ~open

open :-
  legal(you,M)

goal(you,100) :-
  true(step(5))

goal(you,0) :-
  ~true(step(5))
```

a. **Answer Substitutions**

Compute all successful derivations and their answer substitutions to the query `init(F)`.

Hint: You should obtain 9 different answers, which together form the initial game state. Can you draw a simple diagram to visualize it?

b. **Derivations**

The key to unlocking the mystery is to understand the meaning of the three recursive relations, `babbage`, `lovelace`, and `turing`. To get the idea, suppose given some facts shown below.

```
true(row(1,one_coin))
true(row(2,one_coin))
true(row(3,empty))
true(row(4,empty))
true(row(5,one_coin))
true(row(6,one_coin))
true(row(7,two_coins))
true(row(8,one_coin))
```

Which of the following queries have a successful derivation?

- `babbage(2,5)`
- `babbage(2,6)`
- `lovelace(1,5)`
- `lovelace(1,6)`
- `turing(1,6)`
- `turing(1,7)`
- `turing(6,8)`

Can you now describe in words the meaning of `turing(X,Y)`?

c. **Derivations: Legal Moves**

Next, have a look at the definition for `legal(you,jump(X,Y))`. In words, what are the preconditions for jumping from row X to row Y? How many actions are possible in the initial game state?

d. **Derivations: State Update**

Now pick any one of the legal actions `jump(m,n)` in the initial state and compute the new state after `does(you,jump(m,n))`. In words, what is the effect of jumping from X to Y?

e. **Playing**

The definitions for `terminal` and `goal(you,100)` respectively imply that the game ends when you are stuck (i.e. there are no more legal moves) and that you win the game when you can make the maximum of 4 moves before getting stuck. Find a sequence of actions that solves this game!

Hint: There is more than one solution.

f. **Playing**

Bonus challenge: We humans are often better than computers at generalising a solution. How would you solve the game if you start with 998 coins in a row and the goal is to make the maximum of 499 moves without getting stuck?