# 5
# The Auxiliary Language

The Cathedral Theorem Proving Platform contains a compiler for an upper level language. This language is intended to support descriptions of logic deduction systems i.e. inference models. In this language, deduction systems for a logic are written as their deduction rules (or inference rules). The compiler's formal specification is in Appendix B.

Programs written in this language are theorem provers that can be run by the graph machine. Each program is made of rules that the machine tries to apply over a goal. The goal is to prove a formula. It is represented by the sub-graph reached from the vertex set as Goal Pointer. There are two types of rules: non-terminal rules and terminal rules.

Non-terminal rules are defined as graph transformations. They are written as a graph matching pattern; a sub-graph to be transformed into; and a set of commands to be invoked afterwards. The later may be "reset Goal Pointer register", or "load new Local Strategy"; for example. The machine tries to match the finding pattern with the sub-graph started by the Goal Pointer. If it does not match, the machine loads the next rule from Local Strategy register. If it matches, it transforms the graph in memory and executes the commands.

Terminal rules do not make graph transformation. They only try to match the sub-graph starting at Goal Pointer with its definition. In case of success, the machine executes commands and makes a function call to an auxiliary function called success. In case of failure, it calls failure auxiliary function. These mechanisms are defined as auxiliary functions to improve modularity. Success and failure mechanisms are described in 5.3 Strategy-based Programs.

## 5.1. Proof Building Example

In order to introduce the upper level language, the trace of a proof being built is presented. The theorem prover of the example is implemented in a particular way. The implementation uses a backtracking mechanism inspired in [22] cfr. section 9.3 and explained in detail in section 5.3.

The system chosen was a Sequential Calculus. The proof to be built is shown below:

$$\frac{\dfrac{b \vDash b}{x, b \vDash b}}{x, b \vDash a \vee b}$$

In order to enrich the example, the theorem prover's strategy is ingenuous and tries four times to build the proof, so it succeeds in the last time. The proofs are built in this order:

$$\frac{\dfrac{x \vDash a}{x, b \vDash a}}{x, b \vDash a \vee b}$$

$$\frac{\dfrac{b \vDash a}{x, b \vDash a}}{x, b \vDash a \vee b}$$

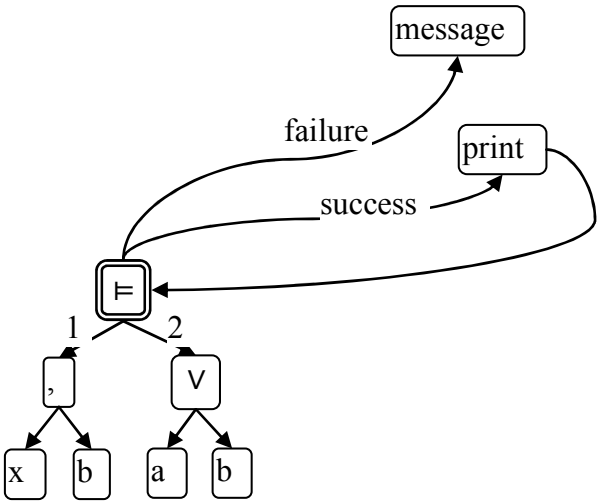
$$\frac{\dfrac{x \vDash b}{x, b \vDash b}}{x, b \vDash a \vee b}$$

$$\frac{\dfrac{b \vDash b}{x, b \vDash b}}{x, b \vDash a \vee b}$$

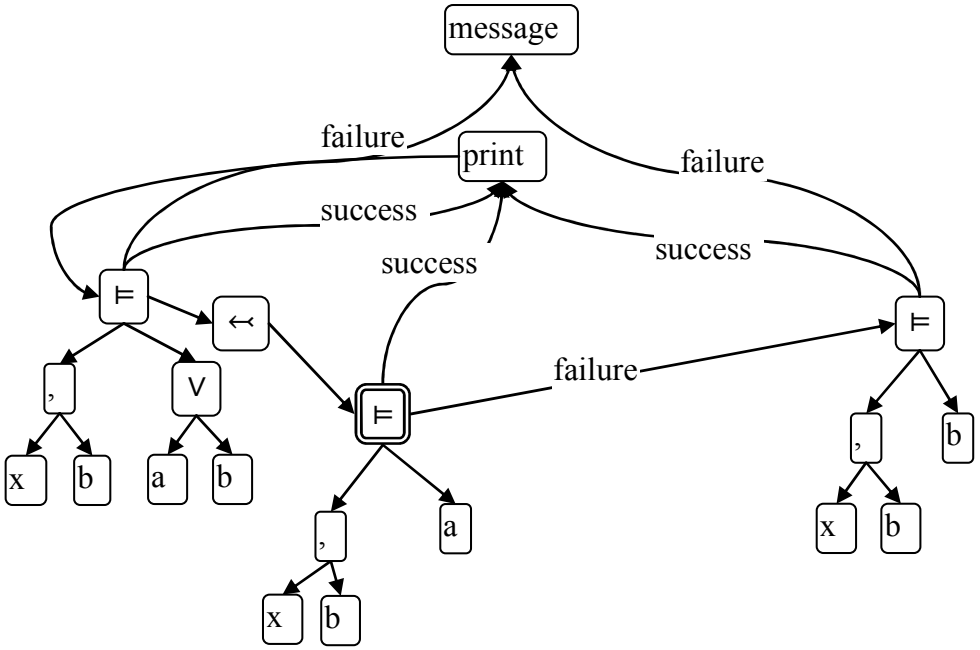
In this way, the used strategy consists of three attempts. The theorem prover first tries to apply the reduction rule for an OR on the consequent. If the application fails; the prover tries to apply a weakening rule on the left side of the sequent. When both attempts fail, the theorem prover inspects the formula trying to find the Sequent Calculus axiom. When the axiom is not found, the backtracking mechanism is used.

$x, b \vDash a \vee b$ is the primary goal to be proved. In the figure bellow, it is shown the graph representation of the sequent. The vertex with double border is the goal vertex. The formula is a subgraph reachable from it, trough unlabeled edges. The edges labeled success and failure and the vertices reached by them are

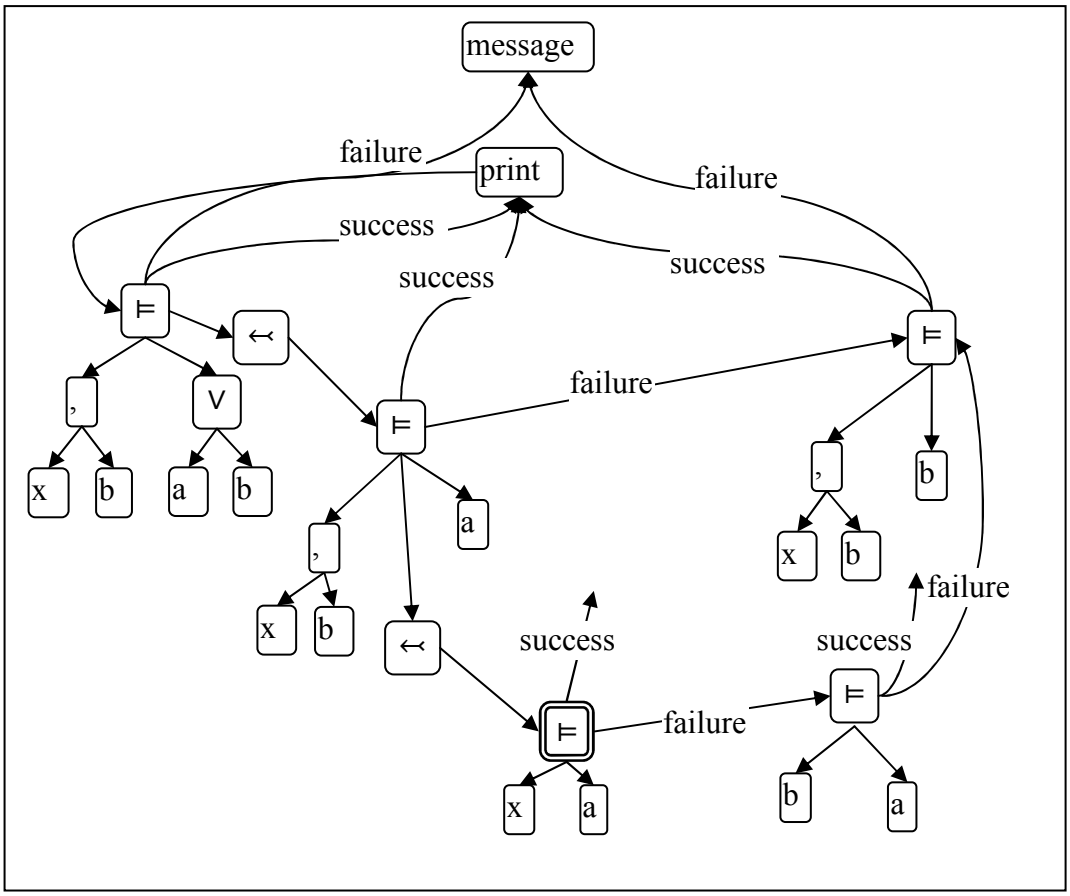
auxiliary metadata; commands. These commands are used at the end of the proof. If the proof succeeds, the prover prints the proof built; else, it prints an output message saying that the formulae cannot be proved.



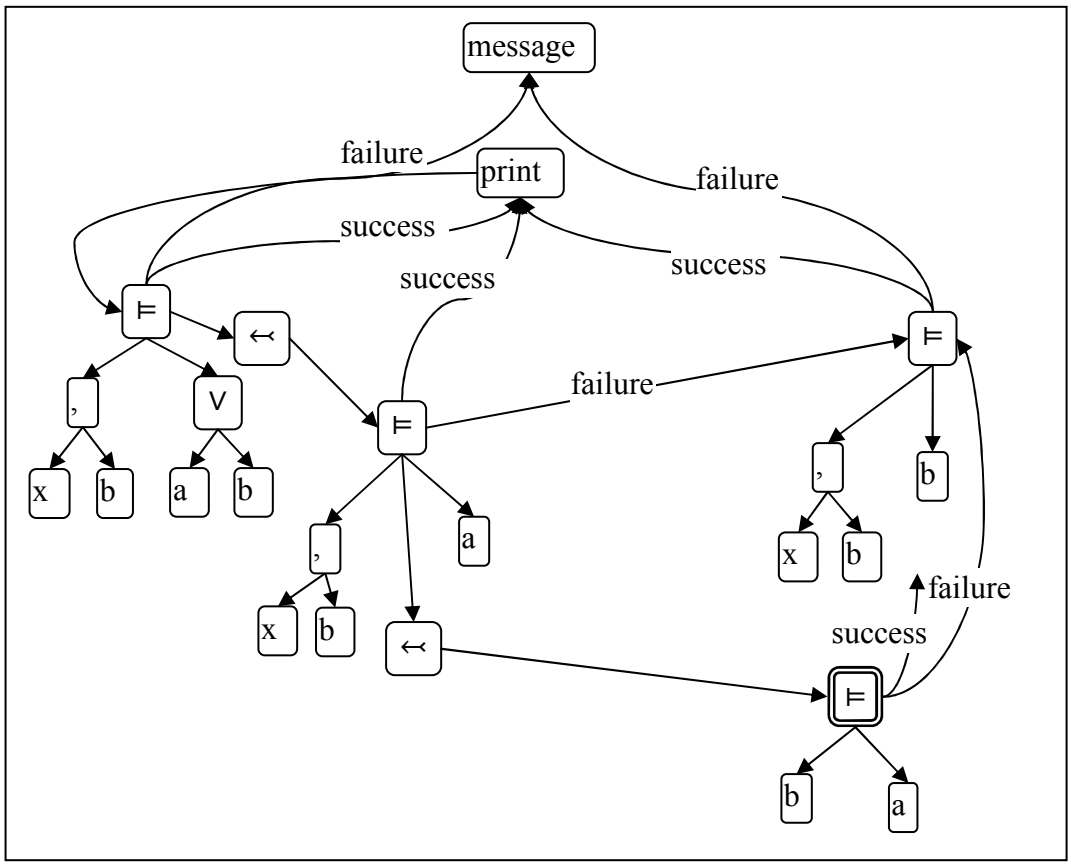

Following the strategy, the prover applies a reduction rule for the OR on the right side of the sequent. Two possibilities of proof are built. The proof snippets are linked by a failure edge. The goal vertex is now the first of the created formulae. The subgraph reachable from it trough unlabeled edges is the formula to be proved at this moment. It can be noticed, that the second formulae is reachable but not by an unlabeled edge; thus, it is not part of the formula to be proved now. The links with the commands for the end of the proof building are created for the new formulae also.



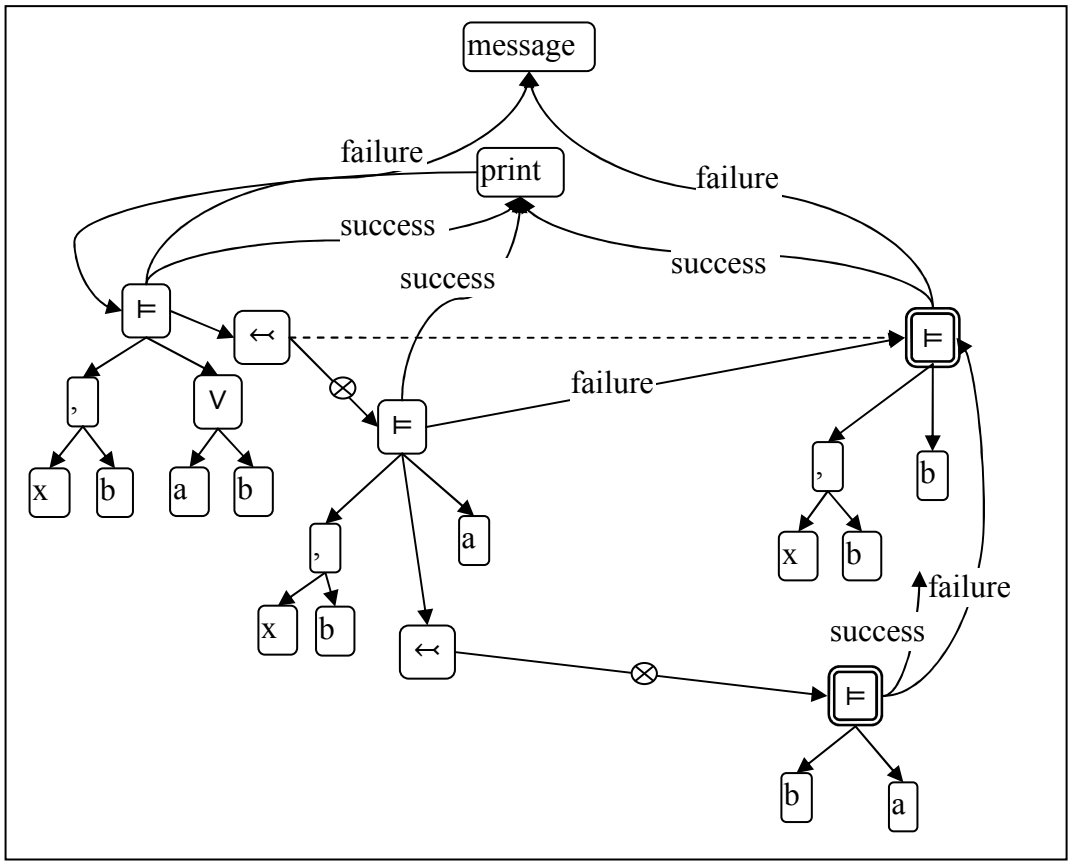

The proof building continues. The new goal is submitted to the theorem prover. According to its strategy, the prover searches an OR but it cannot find one. When the prover tries to apply a weakening rule on the left side of the sequent, it succeeds and builds the proof bellow.



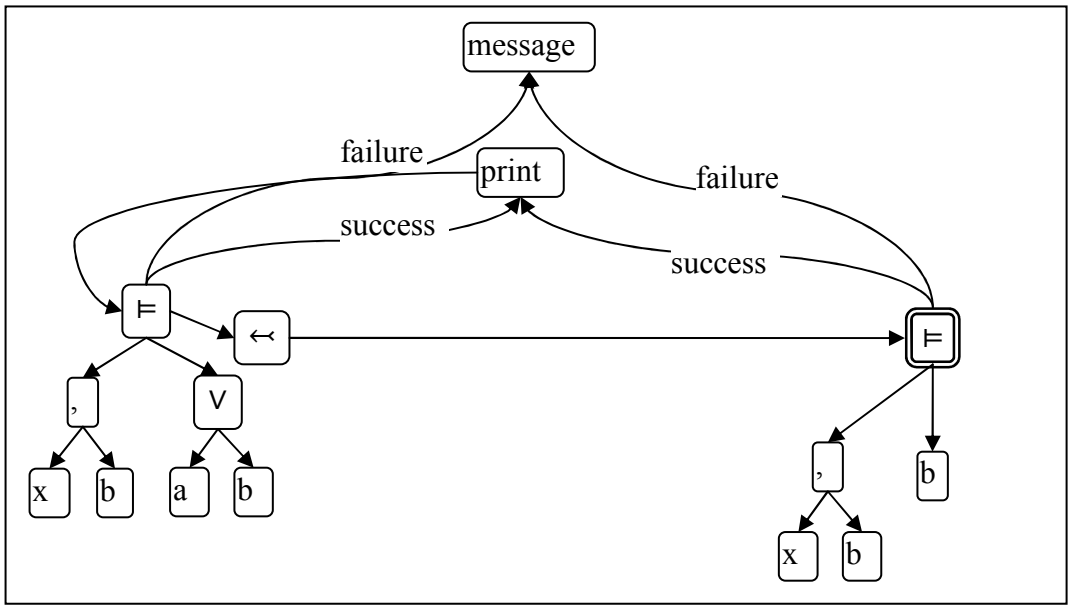
The prover tries to apply the OR reduction rule and the weakening rule on the new goal. The prover does not find the desired sub-formulae, and then it inspects the formula. It searches for the sequent calculus axiom. The goal formula is not an axiom. The built proof cannot be considered. The prover starts the backtracking mechanism that modifies the proof being built. It makes the ↤ bypass the goal. The vertex reachable by the failure edge is now the new goal vertex.



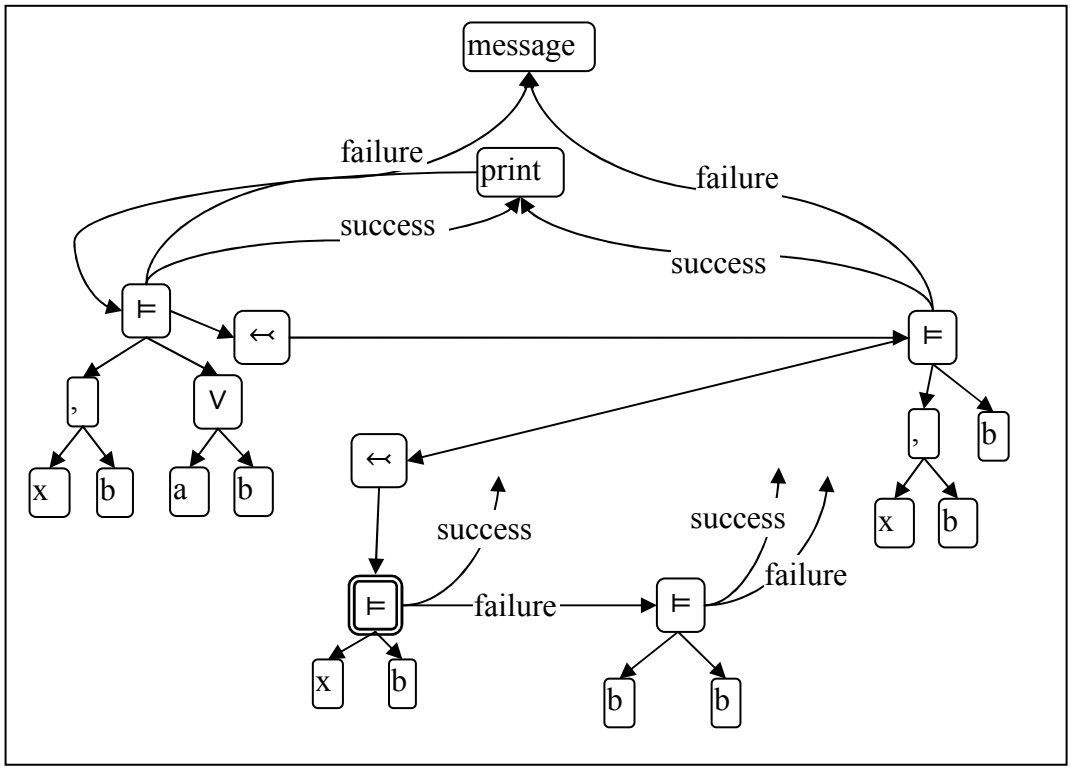

The current goal does not match any rule of the strategy nor the axiom. This time, the bypass mechanism has a thought task. The removed edges are shown bellow with a symbol over it; and the new edge is represented with a dotted line. The algorithm is shown in section 5.3.



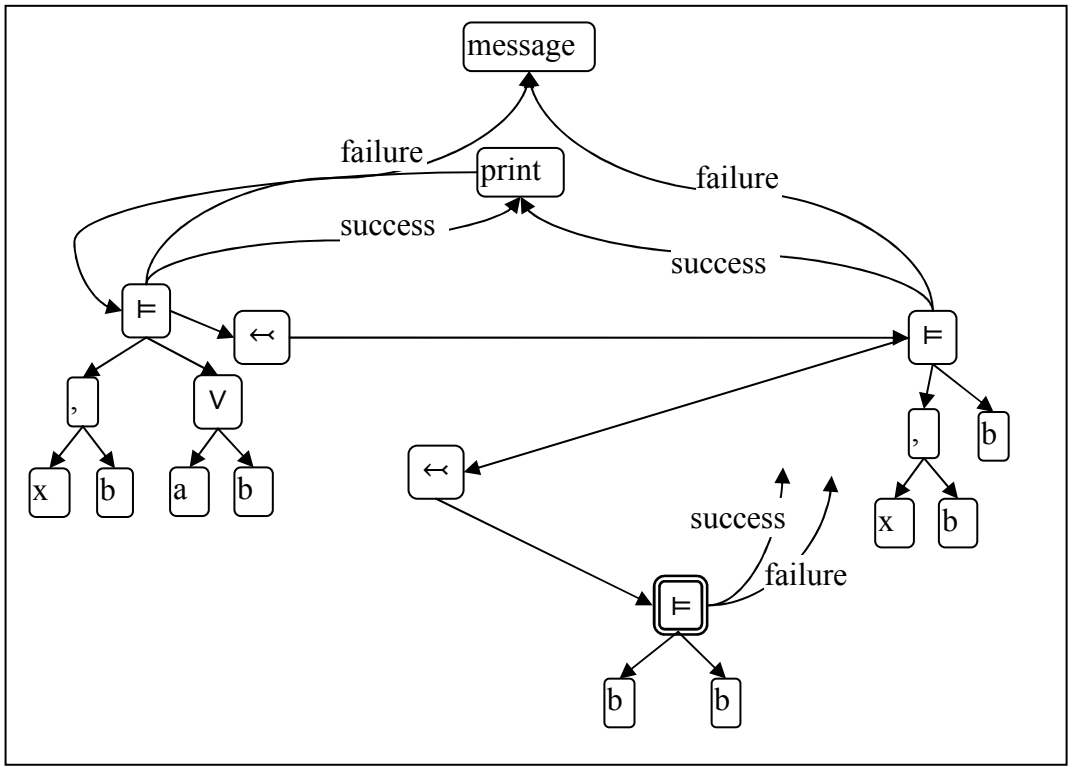

The proof building process now works on a smaller proof. The goal is represented by the doubled border.




The machine fails to apply the OR reduction, and it applies the weakening rule. Two new terminal formulae are generated. The first terminal formula does not accord to the axiom. The theorem prover repeats bypasses it.




When the theorem prover gets to prove the sequential calculus axiom; it completes the proof being built. The prover follows the success edge that leaves the goal vertex. It executes the command in the success vertex, that will print the subgraph reached from it trough unlabeled edges. The printed subgraph is the proof of the given tautology.



## 5.2. Compiler's Functionality

This section describes the compiler's functionality explaining the compilation process using examples. The compiler's formal specification is in Appendix B.

The non-terminal rule used is a definition of weakening rule for Sequential Calculus. It is a rule that compiles two possible weakening rules for the left side of the sequent (antecedent). An algebraic form of the rule can be seen bellow. It is also shown below the code in the upper level language and a graphical representation of the rule in Figure 5.

$$\frac{A \vdash C}{A, B \vdash C} weak_{L1} \quad \frac{B \vdash C}{A, B \vdash C} weak_{L2}$$

```
weak_L ::=
find
v01#"⊢"$ e01#1 $v02#"," ;
v01#"⊢"$ e02#2 $v03#C ;
v02#","$ e03#1 $v04#A ;
v02#","$ e04#2 $v05#B
transform-into
v01#"⊢"$ e05#-1 $v06#"↢" ;
v06#"↢"$ e06#-1 $v07#"⊢" ;
v07#"⊢"$ e07#1 $v08#A ;
v07#"⊢"$ e08#2 $v09#C ;
v07#"⊢"$ e09#-4,success $success(v01) ;
v07#"⊢"$ e10#-3,failure $v10#"⊢" ;
v10#"⊢"$ e11#1 $v11#B ;
v10#"⊢"$ e12#-4,success $success(v01) ;
v10#"⊢"$ e13#-3,failure $failure(v01) ;
v10#"⊢"$ e14#2 $v12#C
set_current_goal[v07]
```

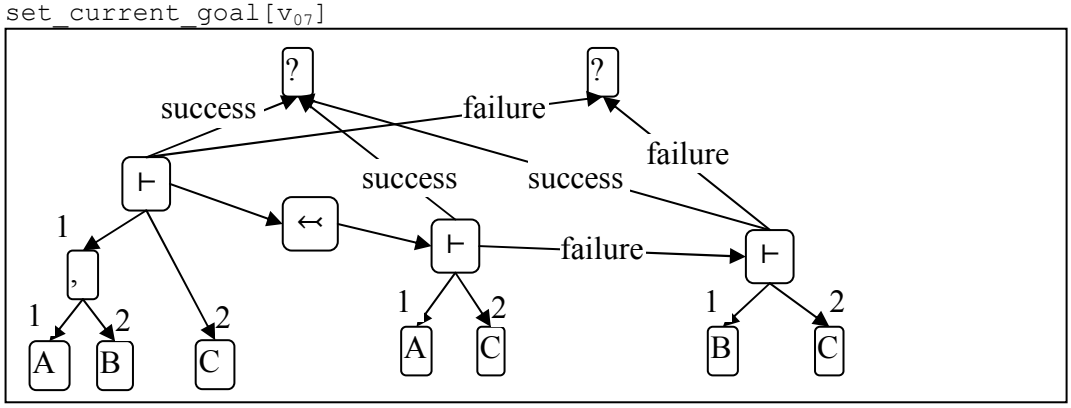



**Figure 5 Left weakening rule definition**

In the upper level language, non-terminal rules definitions are made of three parts: the matching graph, the graph to be transformed into and commands that are invoked after graph manipulation. The matching graph can be seen after the find imperative and before the transform-into. After transform-into, the transformation graph can be seen. The rule of the example, calls only one command, the command to set the Goal Pointer: set current goal.

In the example above, the matching graph is a graph representation of the sequential formula expected as rule conclusion ($A, B \vdash C$). Ordered edges guarantee antecedent and consequent identification. The edges linking the comma separator and its sub-formulae are also ordered. It is an implementation decision, not a conceptual requirement.

Specification shown in Appendix B, describes how the compile generates matching graph code. Each of the code's line represents a complete edge. Its content is the source vertex; the edge order and contents; and the target vertex. Vertices will be repeated every time they appear in an edge.

As the compiler parses a vertex, it identifies if it is a variable (a string) or an operator (a delimited string). Variables are included in the variable set when they are first found. When they already belong to the set, a code snippet comparing them is generated. Operators generate a code snippet comparing it to the vertex currently being worked. A pointer is created for every worked vertex in order to find vertices that were defined or input that was found.

In the example, the first vertex found ($v_{01}$) contains a sequent ("⊢"). This vertex is identifies as an operator and the following code is generated:

```
096#pcreate cv01,WV$098#-1$097#cmp WV,"⊢"
097#cmp WV,"⊢"$100#-1$099#jne
097#cmp WV,"⊢"$102#-1$101#je
099#jne$423#-1$418#lsload
101#je$105#-1$104#pcreate p103,WV
```

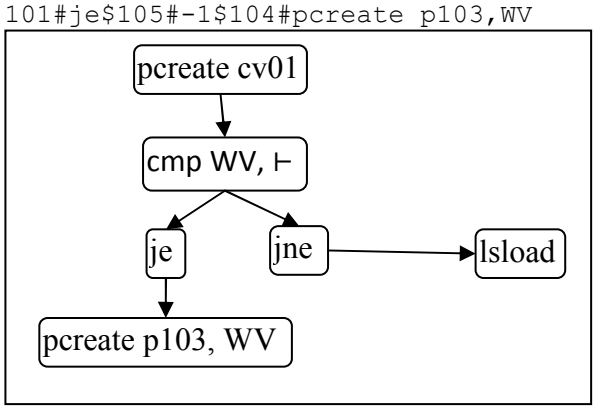



**Figure 6 Operator code snippet**

The compiler parses the edges informations: order and content. It creates a code snippet that gets the set of edges whose source is the identified working vertex. The code iterates over this set and compares edge order and content (if present), searching for the compatible edge. When the edge is found, its target is set as the Working Vertex register.

The transformation graph of the example above is illustrated in Figure 5. Transformation graphs compiling is simpler than finding graphs and they inherit the variable tables synthesized from the first graph.

Vertices which id has been used in the matching graph, or before in the transformation graph are identified and the pointer that was created is synthesized. New vertices generate a code snippet of its creation with pointer and the pointer is synthesized. There is also a way to reference vertices based on edges that are linked to it. They are found by the edge's content (success and failure in the example). In this case, a code snippet is created in order to find them and create a pointer to them. These vertices can be referenced nowhere in the rule, as in the example above.

Every line of the finding graph code is an edge that must be created. The compiler parses each of these lines. It creates edges referencing the vertices whose pointers ware synthesized. These edges receives order and content as specified in the code.

The commands at the end of the rule are function calls. The compiler generates the code for these function calls. Parameters are passed to it and its implementation has to be provided with the other auxiliary functions.

Deduction systems definitions are made of non-terminal rules and of terminal rules. The terminal rules are similar to the non-terminal rules as it can be seen in the example bellow (identity rule). The compilation works the same way it works for non-terminal rules. The terminal rule does not have a transformation graph to build and the generated code contains calls to success and failure mechanisms.

$$\frac{A \vdash A}{\ldots} id$$

```
id :=
v01#"⊢"$ e01#1 $v02#A ;
v01#"⊢"$ e02#2 $v03#A
```

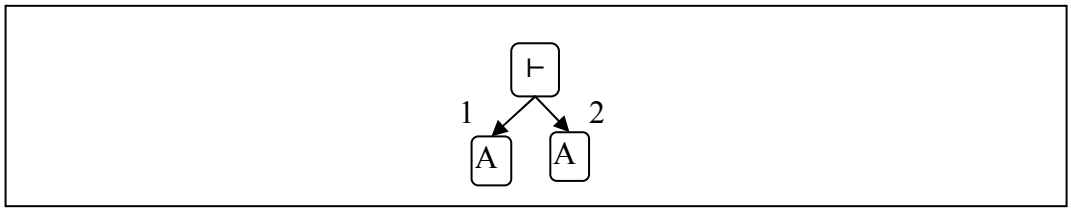



**Figure 7 Identity rule definition**

Whenever defining deduction systems or inference models, WFF formats are the first step of the definition. In the presented language, the approach is different. Proofs and formulae are represented as graphs in a graph machine's memory. It is possible to define a formula and a proof in a variety of different formats.

When the programmer is writing deduction rules, he defines more than WFF formats. The programmer defines a graph format that must be found by the theorem prover. Figure 8 shows different representations of formulae as graph.

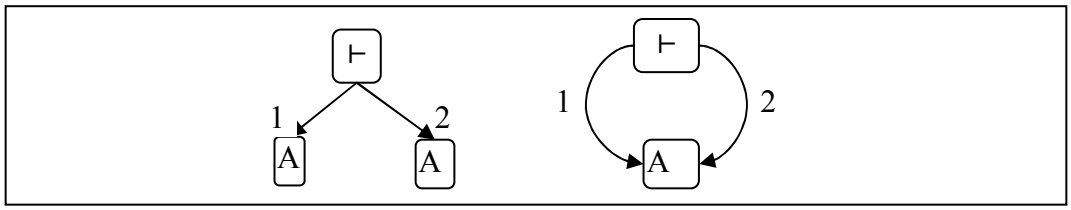



**Figure 8 Different representations of formulae as graph**

## 5.3. Strategy-based Programs

A particular way to develop strategy-based theorem provers (in Game Theory [19] sense) was implemented. It inspired in backtracking mechanisms [22] cfr. section 9.3. The formula to be proved (input or goal) is written as a directed graph. Each node of this graph is an operator, a predicate or a variable. The edges link operators and predicates with sub-formulae completing the structure.

In the goal graph, two extra edges and two extra vertices are added. They are the success and failure actions to be called at the end of the proof. The vertices contain a call to an auxiliary function and the edges link the initial vertex of the sub-graph (Goal Pointer) to them. Figure 9 shows three examples of graphs with success and failure actions. The print vertex is the success action; it prints the whole proof after successful application of the rules. The message vertex is the failure message shown at the end of the proof.

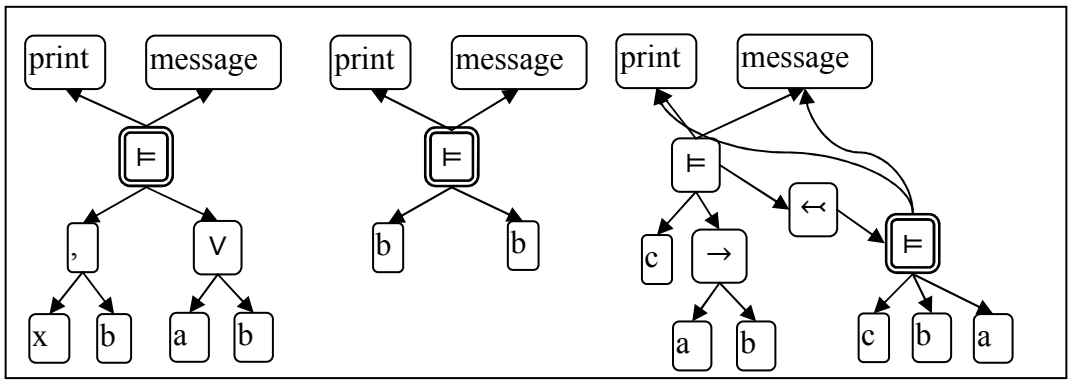



**Figure 9 Examples of proofs represented as graph**

In Figure 9, Goal Pointers are represented as a vertex with double lined border. The sub-graph that represents the formula to be proved starts at the Goal Pointer and contains every vertex that can be reached from it. Only directed edges are used. In the first and second examples, all the vertices belong to the goal sub-graph. The third example, however, illustrates a proof that is being built. Because of the directed edges, the Goal Pointer cannot reach every vertex. This is a simple way to build proofs.

When a terminal rule (5.2 Compiler's Functionality) is executed over a formula, after graph comparison, it calls the success mechanism or the failure mechanism. The implemented success function follows the Goal Pointer's success

edge and executes its target vertex. In the first two examples of Figure 9, the machine calls the function print, which prints the whole proof.

In the third example of Figure 9, a failure vertex that is executed is shown. The failure mechanism is more complicated than success mechanism. Figure 10 shows an example of use where the destroyed edges are crossed and the built edges are dashed. Failure edges are labeled. The function's pseudo-code is shown bellow with its subroutine called bypass.

```
Bypass( V_in )
      Iterate over edges whose source is V_in
            If edge is failure edge
                  V_t <- edge's target
                  Iterates over edges whose target is V_in
                        If edge is unordered
                              V_s <- edge's source vertex
                              Create edge V_s-V_t
     Iterate over edges whose target is V_in
            Destroy edge

Failure
      Iterates over edges whose source is Goal Pointer
            If edge is failure edge
                  V_1 <- edge's target
                  Iterates over edges whose target is V_1
                        If edge is failure edge and its source is
                        not Goal Pointer
                              Bypass( edges source )
      Bypass( Goal Pointer )
      Iterates over edges whose source is Goal Pointer
            If edge is failure edge
                  Add edge's target to Goal List
      Load new Goal Pointer from Goal List
```

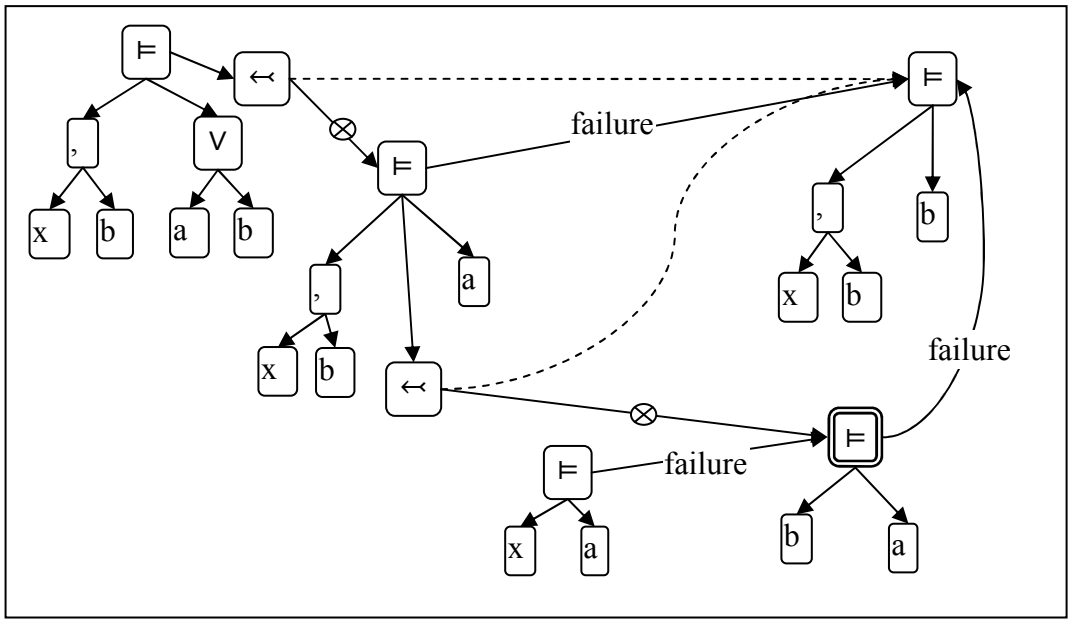



**Figure 10 Failure auxiliary function example**

Whenever a non-terminal rule is executed, it compares the subgraph under Goal Pointer to its definition. If the goal does not accord to the rule's definition, the machine gets the next rule to be tried from the register Local Strategy.

The register Local Strategy is an ordered list of rules to be executed. When this list is null, or when a non-terminal rule matches the goal graph and modifies it; the machine interrupts execution. The ChooserAgent (section 4.1) is advised of the interruption, because it implements choice listener interface. The agent is responsible for editing the Local Strategy register and the Goal List.

Global strategy is implemented in the ChooserAgent. The decisions are played as redefinition of the rules to be executed and of the goals in the proof.