Report Number: WUCS-89-24

1989-06-01

A Justification Finder

Guillermo R. Simari

The technical report presents a succinct description of the Justification Finder (if). The system if is a practical implementation of the theoretical ideas introduced elsewhere (see the technical report "On the Logic of Defeasible Reasoning", G Simari, WUCS-89-12). It is used to explore and validate those ideas. The system provides support for defeasible reasoning in a Prolog environment. The complete Prolog language is available and only a few new predicates are introduced extending the reserved words of the language. We will present the theoretical underpinnings of the system in a very terse manner. The reader is referred to [1]... **Read complete abstract on page 2.**

A Justification Finder

Guillermo R. Simari

Complete Abstract:

The technical report presents a succinct description of the Justification Finder (if). The system if is a practical implementation of the theoretical ideas introduced elsewhere (see the technical report "On the Logic of Defeasible Reasoning", G Simari, WUCS-89-12). It is used to explore and validate those ideas. The system provides support for defeasible reasoning in a Prolog environment. The complete Prolog language is available and only a few new predicates are introduced extending the reserved words of the language. We will present the theoretical underpinnings of the system in a very terse manner. The reader is referred to [1] for a more complete account of defeasible reasoning.

A JUSTIFICATION FINDER

Guillermo R. Simari

WUCS-89-24

June 1989

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

ABSTRACT

This technical report presents a succinct description of the Justification Finder (jf). The system jf is a practical implementation of the theoretical ideas introduced elsewhere (see the technical report "On the Logic of Defeasible Reasoning", G. Simari, WUCS-89-12). It is used to explore and validate those ideas. The system provides support for defeasible reasoning in a Prolog environment. The complete Prolog language is available and only a few new predicates are introduced extending the reserved words of the language. We will present the theoretical underpinnings of the system in a very terse manner. The reader is referred to [1] for a more complete account of defeasible reasoning.

Contents

1 Introduction

This technical report presents a succinct description of the Justification Finder (**jf**). The system **jf** is a practical implementation of the theoretical ideas introduced elsewhere (See the technical report "On the Logic of Defeasible Reasoning", G. Simari, WUCS-89-12)[1]. It is used to explore and validate those ideas.

Like Don Nute's d-Prolog [2, 3], **jf** provides support for defeasible reasoning in a Prolog environment. The complete Prolog language is available and only a few new predicates are introduced extending the reserved words of the language. Unlike d-Prolog, **jf** does not need to be supplied with "defeater rules". The system will find the defeaters among the theories it is able to construct, using specificity constrains.

We will give, as an appendix, the theoretical underpinnings of the system. That appendix will introduce, in a very terse manner, the indispensable definitions. The reader is referred to above mentioned technical report for a more complete, general, account of defeasible reasoning.

2 The Representational Language of jf

Given an agent **a** we will represent his knowledge using the Horn clause subset, denoted **neg**-$Horn$, of first order logic in which we distinguish a negation relation (**neg**), and a set of defeasible rules constructed in a **neg**-$Horn$ clause-like manner, *i.e.* they have the same form as those clauses but the symbol "<==" is replaced by the symbol "--<". Therefore the context $\mathcal{K}$ will be a subset of **neg**-$Horn$, and Δ will be composed of defeasible rules constructed as described.

The system **jf** provides a very simple language to represent the context $\mathcal{K}$ and the set of defeasible rules Δ. We will use the standard terminology in logic programming extending the meaning of some terms to include the similar entities described here. We will follow the conventions for Prolog regarding the representations of predicates and constants (lower case) and variables (upper case). For an introduction to the programming language Prolog see [4, 5].

We will refer to the knowledge represented by the context $\mathcal{K}$ and the defeasible rules in Δ as the *knowledge base*. Next, we will describe the language used to represent a knowledge base.

2.1 The Context $\mathcal{K}$

The context $\mathcal{K}$ is divided in the set $\mathcal{N}$ of necessary facts and the set $\mathcal{C}$ of contingent facts. In the appendix, members of $\mathcal{N}$ that had been represented in the form $A \supset B$ now have the general form:

```
B <== A1,A2,...,An.
```

where Ai and B are Prolog *compound terms* with arity 0 or bigger, possible affected by the neg predicate (see below). The *head* B of the rule is an atomic goal. The *body* of the rule, A1,An,...,An, is understood as the conjunction of the compound terms A1, A2, ..., An. The period at the end of the rule is syntactically necessary as terminator. Variables in the rule, if they exist, are assumed universally quantified, and their scope is the whole rule. In the case of the contingent facts, previously represented simply as B, there is only one term Ai in the body of the rule and it has the special form true. For example,

```
bird(X) <== penguin(X).
```
penguins are birds, is a necessary fact.

```
penguin(opus) <== true.
```
opus is a penguin, is a contingent fact.

The change in notation from the appendix is motivated by the wish of keeping a common syntactic "flavor" with Prolog, while having a separated representation at the same time.

2.2 The Set Δ of Defeasible Rules

The defeasible rules represented as $A \succ\!\!- B$ in the appendix will have the form:

```
B --< A1,A2,...,An.
```

where Ai and B are Prolog *compound terms* with arity 0 or bigger, possible affected by the neg predicate (see below). The *head* B of the rule is an atomic goal. The *body* of the rule, A1,An,...,An, is understood as the conjunction of the compound terms A1, A2, ..., An. The period at the end of the rule is syntactically necessary as terminator. Variables in the rule are assumed universally quantified, and their scope is the whole rule. For example:

```
fly(X) --< bird(X).
```
birds usually fly

```
neg fly(X) --< penguin(X).
```
penguins usually do not fly

are defeasible rules. Again, the change in representation is motivated by the desire of maintaining a Prolog-like syntax.

2.3 Negation (neg)

Negation is handled in the same way as proposed by Nute [2, 3]. We introduce the relation neg, as a prefix operator, with the only property that

```
neg neg A = A
```

The goal `neg A` will be assumed as a consequence of a subset R of neg-$Horn$ if and only if `neg A` is deducible from R via a finite number of applications of modus ponens. The goal `neg neg A` will be assumed as a consequence of a set R of rules in neg-$Horn$ if and only if `neg neg A`, or `A`, is deducible from R via a finite number of applications of modus ponens.

This form of negation is in no way related to "negation as failure" [6] as usually defined in Prolog. Negation, the prefix operator *not*, in Prolog is defined by saying that the goal *not A* succeeds if and only if the goal *A* is "finitely failed". In that way, the operator *not* is only a partial form of negation in first order logic. Several other forms of negation, with increasing power, can be implemented (see Naish [7]), but it would require to change the resolution method. We would like to keep Prolog as our base language and take advantage of the efficient compilers available.

Our neg operator can appear in the head of the rules, defeasible and otherwise. For instance,

```
neg A <== true.
neg A <== neg B, C.
neg A --< B, C, neg D.
```

are legal rules. Notice that the first rule is asserting the fact `neg A`.

3 The Command Language of jf

Assuming that **jf** is available, the system is invoked by entering `jf` at the Unix prompt, represented here as "%", obtaining the following answer:

```
% jf

      Justification Finder
          Version  1.0
          June 1, 1989
yes
| ?-
```

At this point the system is loaded, and waiting for a request to be made. **jf** offers the user an interactive environment with tools for incrementally building a knowledge base.

The text representing the rules in $\mathcal{K}$ and $\mathit{\Delta}$ of a knowledge base is normally created in two files using a text editor. These two files contain the context $\mathcal{K}$ and the defeasible rules in $\mathit{\Delta}$. They must have a common prefix name with the extensions "`context`" and "`delta`" respectively. For instance, if we choose to give our file the prefix name "`kb`", the

file for $\mathcal{K}$ must be called "`kb.context`" and the file for Δ must be called "`kb.delta`". The prefix is arbitrary, the extensions are mandatory. **jf** can then be instructed to read-in the rules that form the knowledge base from these files. To change the knowledge base loaded in memory, it is possible to add rules to, or delete rules from, it. The current state of the knowledge base can be shown, and the system provides a command to save the knowledge base as it is in memory at that stage of the execution. The capability to delete an entire knowledge base from memory is also available.

Once a knowledge base is loaded and modified to the user satisfaction, **jf** can be used to explore what is being supported by the rules. Commands for producing an entire listing of supported facts and explore how a particular fact has become supported (or unsupported) are also provided. We will now present a comprehensive account of the commands available.

3.1 Notation

The commands and system responses will be shown in `teletype` font. When the command is introduced we will use *emphasized* text to represent a generic name, *i.e.* in the command "`load_kb(`*+knowledge_base*`).`", the `load_kb` word, the parenthesis and the period are required as shown. Meanwhile, the word "*knowledge_base*" is generic and can be replaced by a real name chosen by the user. Uppercase letters will represent variables and lowercase letters will represent constants. The plus sign "*+*" in front of "*knowledge_base*" is a *mode annotation* affecting the interpretation of a given parameter. They have the following meaning:

"*+*":	The argument is an input to the command.
"*-*" :	The argument is returned by the command, is an output.
"*?*" :	The argument is an input or an output of the command.

This conventions follow similar ones used in Quintus Prolog. These characters *must not be entered with the commands*, they are only a documentation aid.

The sequence of characters "`| ?- `" shown before the text of a command is the Prolog prompt. The word "`yes`", or "`no`", that sometimes precedes the Prolog prompt after a query from the user is the Prolog response to that query. In our case, that word is the response to a command and indicates that it was successful, in case of "`yes`", or failed, in case of "`no`".

All the **jf** commands are terminated by a period ".", which is the query terminator for Prolog. After the period, and not shown in the text, the user must enter a "return" by pressing the appropriate key in the keyboard.

3.2 Maintenance of the Knowledge Base

As mentioned above, the contents of $\mathcal{K}$ and Δ are put in two separate files, using any text editor. The files for $\mathcal{K}$ and Δ can include comments using the standard syntax for Prolog comments. If we choose to name our database with the prefix "`kb`", then the file for $\mathcal{K}$ will be "`kb.context`" and the file for Δ will be "`kb.delta`".

We will now introduce the commands used in the maintenance of the knowledge base after its initial creation by the user. The actions related to the maintenance are: loading a knowledge base to the system, showing the current content of the knowledge base, adding and removing single rules from it, saving the current knowledge base to files, removing all the rules from the system.

3.2.1 Loading the Knowledge Base

The form of the command is:

`load_kb(`*+knowledge_base*`).`

where *knowledge_base* is the prefix name chosen for the knowledge base. Suppose we are loading the knowledge base "`kb`", the system will automatically default to loading "`kb.context`" and "`kb.delta`", as shown below:

```
| ?- load_kb(kb).
[consulting kb.delta...]
[kb.delta consulted 0.117 sec 856 bytes]
[consulting kb.context...]
[kb.context consulted 0.100 sec 880 bytes]

yes
| ?-
```

3.2.2 Displaying the Knowledge Base

The form of the command is:

`print_kb.`

with no parameter. The command will print the contents of the knowledge base in memory as shown below. Notice that the variables are printed in lower case. The system will use the variable names `x, x1, x2, ...` to name the different variables in a rule for printing purposes. Remember that the scope of a variable is just the rule that contains it. The reason to use these names is purely cosmetic. Prolog names the variables internally using

the underscore character and a number, *i.e.* _73, and if used the printout of the rules would be unclear.

Our knowledge base contains a number of context and defeasible rules representing a traditional problem in nonmotonic reasoning, the "canonical example". Basically, it says that birds usually fly, penguins usually do not fly, all penguins are birds, all canaries are birds and that "opus" is a penguin and "tweety" is a canary.

```
| ?- print_kb.
========================================================================

   Defeasible Rules (Delta)

            flies(x) --< bird(x).

            neg flies(x) --< penguin(x).

------------------------------------------------------------------------

   Context Knowledge (K)

        Necessary Facts

            bird(x) <== penguin(x).

            bird(x) <== canary(x).

        Contingent Facts

            penguin(opus) <== true.

            canary(tweety) <== true.

========================================================================

yes
| ?-
```

3.2.3 Adding and Removing Rules from the Knowledge Base

Once the knowledge base is in memory, or as another way of creating a knowledge base from scratch, the user can add or remove rules to it. The syntax of the commands is:

`add_rule((+`*newrule*`)).`

for adding a rule "*newrule*", and

`remove_rule((+`*oldrule*`)).`

for removing a rule "*oldrule*".

Notice the double parenthesis set around the rule. That is syntactically necessary to prevent Prolog from doing anything before jf compiles the rule to an internal representation. For instance, we can add a rule saying that penguins in planes fly and that "opus" and "tweety" are in a plane,

```
| ?- add_rule((flies(X) --< is_in_a_plane(X), penguin(X))).

X = _42

| ?- add_rule((is_in_a_plane(opus) <== true)).

yes
| ?- add_rule((is_in_a_plane(tweety) <== true)).

yes
| ?-
```

The response "`X = _42`" after the execution of the first addition is Prolog showing the binding of the variable "`X`", bound to the internal variable "`_42`" in this case.

Now we notice that "tweety" is not in the plane anymore, he has flown away, and we want to remove the rule saying that he is in the plane. The following command will have that effect,

```
| ?- remove_rule((is_in_a_plane(tweety) <== true)).

yes
| ?-
```

3.2.4 Saving the Current Knowledge Base

After interacting with the knowledge base, *i.e.* adding rules to, and removing rules from it, the user may want to save the current status of the knowledge base to a file. That can be accomplished using the command:

`save_kb((+`*knowledge_base*`)).`

where "*knowledge_base*" is the prefix name of the files for $\mathcal{K}$ and $\boldsymbol{\Delta}$, *i.e.* the current status of the knowledge base will be saved to "*knowledge_base*`.context`" and "*knowledge_base*`.delta`". The prefix name could be the same as before, in which case the old files will get destroyed, or a new prefix name for creating a new knowledge base. For instance, the command:

```
| ?- save_kb(kb2).

yes
| ?-
```

has saved the current content of the knowledge base to the files "`kb2.context`" and "`kb2.delta`".

3.2.5 Removing the Current Knowledge Base

After working with a knowledge base and before loading a new one it is convenient to remove all the rules corresponding to the old knowledge base. That can be accomplished using the command,

```
remove_kb.
```

For instance, the command

```
| ?- remove_kb.

yes
| ?-
```

will remove all the rules corresponding to the context and all the defeasible rules in the system.

3.3 Interaction with the Knowledge Base

Several possibilities are open at this point. We will describe them in increasing order of complexity. For all the interactions with the knowledge base we will assume the following status for it:

```
| ?- print_kb.
========================================================================

   Defeasible Rules (Delta)

            tree_dweller(x) --< flies(x).

            flies(x) --< bird(x).

            neg flies(x) --< penguin(x).

            flies(x) --< is_in_a_plane(x), penguin(x).

------------------------------------------------------------------------

   Context Knowledge (K)

        Necessary Facts

            bird(x) <== penguin(x).

            bird(x) <== canary(x).

        Contingent Facts

            penguin(opus) <== true.

            canary(tweety) <== true.

            is_in_a_plane(opus) <== true.

========================================================================

yes
| ?-
```

3.3.1 Notation for Theories

Theories will be returned to the user as a list of rules in the same notation as they are input and shown by the `print_kb` command. For instance, the following is a theory for "`flies(opus)`" as returned by `jf`:

```
[(flies(opus)--<bird(opus)),(bird(opus)<==penguin(opus)),(penguin(opus)<==true)]
```

Notice the parenthesis around every rule. The order in which the rules are shown is the order in which they are found.

3.3.2 Finding and Displaying Theories for Facts

A very primitive operation over the knowledge base is the finding of theories for a fact that *could* be supported by it. The following command performs that action:

`get_theory(`*+fact, −Theory*`)`.

where *fact* is a supposedly supported fact and *Theory* should be a variable that would be instantiated by **jf** in case it succeeds in finding a theory that could support the fact. For instance, for the fact `flies(opus)` the system will return:

```
| ?- get_theory(flies(opus),T).

T = [(flies(opus)--<bird(opus)),
     (bird(opus)<==penguin(opus)),(penguin(opus)<==true)]

| ?-
```

The line "`T = [(flies(opus) ...`", is the instantiation of "T" returned by **jf**. We have changed the format of the line for typesetting reasons. Prolog prints that text in only one line. The only change introduced was a carriage-return after the first rule. The system will wait at the end of that line for a request from the user to backtrack or just a carriage-return. In this case we enter the latter. In the following example we request the backtracking using the character "**;**" at the end of the line. We obtain another theory and when we ask for another the system answer "no" signaling that there are no more theories for the fact "`flies(opus)`":

```
| ?- get_theory(flies(opus),T).

T = [(flies(opus)--<bird(opus)),
     (bird(opus)<==penguin(opus)),(penguin(opus)<==true)] ;

T = [(flies(opus)--<is_in_a_plane(opus), penguin(opus)),
     (penguin(opus)<==true),(is_in_a_plane(opus)<==true)] ;

no
| ?-
```

Again we change the format of the instantiation line in the way above described.

The theories returned by **jf** can be pretty-printed using the command:

`print_theory(+`*theory*`).`

where *theory* is a list of rules in the same format as returned by the command "`get_theory`". For instance, we first obtain a theory for "`tree_dweller(opus)`" using "`get_theory`" and then we print it using "`print_theory`":

```
| ?- get_theory(tree_dweller(opus),T), print_theory(T).

  Theory for "tree_dweller(opus)"

        tree_dweller(opus) --< flies(opus).

        flies(opus) --< bird(opus).

  Context used

        bird(opus) <== penguin(opus).

        penguin(opus) <== true.

T = [(tree_dweller(opus)--<flies(opus)),
     (flies(opus)--<bird(opus)),
     (bird(opus)<==penguin(opus)),(penguin(opus)<==true)]

| ?-
```

is the result of printing the theory for "`tree_dweller(opus)`" obtained by using "`get_theory`". The backtracking is still available. Theories can be printed also as trees using the command:

`print_theory_as_tree(+`*theory*`).`

where *theory* is a list of rules in the same format as returned by the command "`get_theory`". For instance, we first obtain a theory for "`tree_dweller(opus)`" using "`get_theory`" and then we print it using "`print_theory_as_tree`":

```
| ?- get_theory(tree_dweller(opus),T), print_theory_as_tree(T).


  Theory for "tree_dweller(opus)"


    tree_dweller(opus) --<
              flies(opus) --<
                       bird(opus) <==
                                penguin(opus) <== true


T = [(tree_dweller(opus)--<flies(opus)),
     (flies(opus)--<bird(opus)),
     (bird(opus)<==penguin(opus)),(penguin(opus)<==true)]

| ?-
```

3.3.3 Finding and Displaying Counterarguments and Defeaters

The reader is referred to the appendix for the basic definitions. Throughout this section we will assume the status shown in page 9 for the knowledge base. Commands for finding and displaying counterarguments and defeaters are introduced together in this section because of the similar syntax.

The commands for finding counterarguments and defeaters expect a theory as input, so they should be used in conjunction with the "`get_theory`" command. The command for counterarguments has the following form:

`get_theory_counterarguments(`*+theory, −List*`)`.

where *theory* is a theory expressed as a list of rules (see page 9). The output parameter *List* will contain a list of ordered pairs [*fact, Theory*] where *fact* is a counterargument point and *Theory* is the theory which supports it. The command for defeaters has the following form:

`get_theory_defeaters(`*+theory, −List*`)`.

where *theory* is a theory expressed as for counterarguments (see page 9). The output parameter *List* will contain a list of ordered pairs [*fact, Theory*], *fact* is the point of counterargument for a defeater counterargument, *Theory* is the theory which supports it. Examples of both commands follow. Observe that before executing the commands we call "`get_theory`". Example for "`get_theory_counterarguments`":

```
| ?- get_theory(flies(opus),T), get_theory_counterarguments(T,C).

T = [(flies(opus)--<bird(opus)),
     (bird(opus)<==penguin(opus)),(penguin(opus)<==true)],

C = [[neg flies(opus),[(neg flies(opus)--<penguin(opus)),(penguin(opus)<==true)]]]

| ?-
```

and an example for "get_theory_defeaters":

```
| ?- get_theory(neg flies(opus),T), get_theory_defeaters(T,D).

T = [(neg flies(opus)--<penguin(opus)),(penguin(opus)<==true)],

D = [[flies(opus),[(flies(opus)--<is_in_a_plane(opus),penguin(opus)),
      (penguin(opus)<==true),(is_in_a_plane(opus)<==true)]]]

| ?-
```

In both examples the instantiation of the variables is shown by Prolog. There is a better way of displaying the counterarguments and defeaters by using the commands "print_theory_counterarguments" and "print_theory_defeaters". The command format for counterarguments is:

print_theory_counterarguments(+*list*).

where *list* will contain a list of ordered pairs [*fact, theory*], where *fact* is a counterargument point, and *theory* is the theory which supports it. The command syntax for defeaters is:

print_theory_defeaters(+*list*).

where *list* will contain a list of ordered pairs [*fact, theory*], where *fact* is a defeater point, and *theory* is the theory which supports it. Examples of both commands follow. Observe that before executing the commands we call "get_theory", "get_theory_counterargument", and "get_theory_defeater". Example for counterarguments:

```
| ?- get_theory(flies(opus),T), get_theory_counterarguments(T,C),
     print_theory_counterarguments(C).

---(Counterargument)-------------------------------------------------

  Theory for "neg flies(opus)"

        neg flies(opus) --< penguin(opus).


  Context used

        penguin(opus) <== true.

----------------------------------------------------------------------

T = [(flies(opus)--<bird(opus)),
     (bird(opus)<==penguin(opus)),(penguin(opus)<==true)],

C = [[neg flies(opus),[(neg flies(opus)--<penguin(opus)),(penguin(opus)<==true)]]]

| ?-
```

Example for defeaters:

```
| ?- get_theory(neg flies(opus),T), get_theory_defeaters(T,D),
     print_theory_defeaters(D).

---(Defeater)--------------------------------------------------------

  Theory for "flies(opus)"

        flies(opus) --< is_in_a_plane(opus), penguin(opus).


  Context used

        penguin(opus) <== true.

        is_in_a_plane(opus) <== true.

----------------------------------------------------------------------

T = [(neg flies(opus)--<penguin(opus)),(penguin(opus)<==true)],

D = [[flies(opus),[(flies(opus)--<is_in_a_plane(opus),penguin(opus)),
      (penguin(opus)<==true),(is_in_a_plane(opus)<==true)]]]

| ?-
```

3.3.4 Finding Justifications for Facts

The system **jf** provides two commands for finding justifications for facts. The first one is very general an inspects the whole knowledge base, and the second works on a particular fact showing the interplay between defeaters and counterarguments in the *making* of a justification. The format for the inspection of the entire knowledge base is:

```
show_justified.
```

An example of the output produced when the knowledge base is in the status shown on page 9 follows.

```
| ?- show_justified.

------------------------------------------------------------------------

[1] Justification for: is_in_a_plane(opus)

    is_in_a_plane(opus) <== true

------------------------------------------------------------------------

[2] Justification for: canary(tweety)

    canary(tweety) <== true

------------------------------------------------------------------------

[3] Justification for: penguin(opus)

    penguin(opus) <== true

------------------------------------------------------------------------

[4] Justification for: bird(opus)

    bird(opus) <==
            penguin(opus) <== true

------------------------------------------------------------------------

[5] Justification for: bird(tweety)

    bird(tweety) <==
            canary(tweety) <== true

------------------------------------------------------------------------
```

```
[6] Justification for: flies(opus)

    flies(opus) --<
            is_in_a_plane(opus) <== true
            penguin(opus) <== true

------------------------------------------------------------------------

[7] Justification for: flies(opus)

    flies(opus) --<
            bird(opus) <==
                    penguin(opus) <== true

------------------------------------------------------------------------

[8] Justification for: flies(tweety)

    flies(tweety) --<
            bird(tweety) <==
                    canary(tweety) <== true

------------------------------------------------------------------------

[9] Justification for: tree_dweller(opus)

    tree_dweller(opus) --<
            flies(opus) --<
                    is_in_a_plane(opus) <== true
                    penguin(opus) <== true

------------------------------------------------------------------------

[10] Justification for: tree_dweller(opus)

    tree_dweller(opus) --<
            flies(opus) --<
                    bird(opus) <==
                            penguin(opus) <== true

------------------------------------------------------------------------
```

```
[11] Justification for: tree_dweller(tweety)

    tree_dweller(tweety) --<
            flies(tweety) --<
                    bird(tweety) <==
                            canary(tweety) <== true

------------------------------------------------------------------------

rm: remove jf-tmp-file?

yes
| ?-
```

The message "`rm: remove jf-tmp-file?`" can be ignored by entering a carriage return. The system produces a file containing the whole listing named "`jf-tmp-file`" which pipes through the Unix utility "`more`". The system is asking if the temporary file can be removed, a "y" answer will remove the file and a carriage return, "n", will save it. Saving the file can be useful for printing out its content.

The second command introduced in this section can be used to see the process for which a fact becomes justified or fails to obtain a justification. The syntax is:

`analyze(+`*fact*`)`.

where *fact* is the fact for which the justification is sought. Several examples follow.

```
| ?- analyze(flies(opus)).

========================================================================

Candidate(s) to justify "flies(opus)":

<T1,flies(opus)>

    flies(opus) --<
            is_in_a_plane(opus) <== true
            penguin(opus) <== true

<T2,flies(opus)>

    flies(opus) --<
            bird(opus) <==
                    penguin(opus) <== true

========================================================================

Analyzing: <T1,flies(opus)>
```

```
    flies(opus) --<
            is_in_a_plane(opus) <== true
            penguin(opus) <== true

It has counterargument(s):

    Theory for "neg flies(opus)":

    neg flies(opus) --<
            penguin(opus) <== true

No counterargument found, or all counterarguments were defeated.

The theory being analyzed has no defeater.
------------------------------------------------------------------------
Analyzing: <T2,flies(opus)>

    flies(opus) --<
            bird(opus) <==
                    penguin(opus) <== true

It has counterargument(s):

    Theory for "neg flies(opus)":

    neg flies(opus) --<
            penguin(opus) <== true

No counterargument found, or all counterarguments were defeated.

The theory being analyzed has no defeater.
------------------------------------------------------------------------


rm: remove jf-tmp-file?

yes
| ?-
```

```
| ?- analyze(neg flies(opus)).

========================================================================

Candidate(s) to justify "neg flies(opus)":

<T1,neg flies(opus)>

    neg flies(opus) --<
            penguin(opus) <== true

========================================================================

Analyzing: <T1,neg flies(opus)>

    neg flies(opus) --<
            penguin(opus) <== true

It has counterargument(s):

    Theory for "flies(opus)":

    flies(opus) --<
            is_in_a_plane(opus) <== true
            penguin(opus) <== true

    Theory for "flies(opus)":

    flies(opus) --<
            bird(opus) <==
                    penguin(opus) <== true

Counterargument at "flies(opus)" has no defeater.

The theory being analyzed has defeater(s):

    Theory for "flies(opus)":

    flies(opus) --<
            is_in_a_plane(opus) <== true
            penguin(opus) <== true

------------------------------------------------------------------------


rm: remove jf-tmp-file?

yes
| ?-
```

```
| ?- analyze(bird(tweety)).

========================================================================

Candidate(s) to justify "bird(tweety)":

<T1,bird(tweety)>  ( T1 = empty_set )

    bird(tweety) <==
            canary(tweety) <== true

========================================================================

The fact "bird(tweety)" is logically derivable from K

No counterargument found, or all counterarguments were defeated.

The theory being analyzed has no defeater.
------------------------------------------------------------------------


rm: remove jf-tmp-file?

yes
| ?-

| ?- analyze(neg flies(tweety)).

Fact "neg flies(tweety)" has no supporting theory


rm: remove jf-tmp-file?

yes
| ?-
```

4 Appendix: Basic Definitions

This appendix is a summary of the relevant definitions contained in [1]. The reader is encouraged to refer to that source for a more detailed exposition.

The language introduced here will be appropriate to represent the knowledge of a given agent **a** about the world and perform his defeasible reasoning. In that construction we follow the characterizations given by Poole [8] for specificity and by Pollock [9] for justifications.

Theories and Specificity

The knowledge of **a** is represented by a subset $\mathcal{K}$ of the language of *predicate logic*. We recognize some structure in $\mathcal{K}$ distinguishing between *necessary facts* $\mathcal{N}$ and *contingent facts* $\mathcal{C}$, *i.e.* $\mathcal{K} = \mathcal{N} \cup \mathcal{C}$ and we require that $\mathcal{K} \nvdash \bot$, *i.e.* that $\mathcal{K}$ be consistent. The set of necessary facts represents the general knowledge applicable to every situation; meanwhile the set of contingent facts represents the particular knowledge regarding the situation at hand, the "observables". Sometimes we will refer to $\mathcal{K}$ as the *context*.

Defeasible rules have the form $A \succ\!\!- B$, where the relation "$\succ\!\!-$" is understood as saying that "reasons to believe in the antecedent provide reasons to believe in the consequent". These rules represent tentative inferences to be used in conjunction with $\mathcal{K}$.

DEFINITION **4.1 (Theory)** *Given a context* $\mathcal{K} = \mathcal{N} \cup \mathcal{C}$ *and a set* $\boldsymbol{\Delta}$ *of defeasible rules we say that* $T \subseteq \boldsymbol{\Delta}$ *is a theory for* h *in the context* $\mathcal{K}$*, denoted by* $\langle T, h \rangle_{\mathcal{K}}$*, if and only if*

$$\mathcal{K} \cup T \mid\!\sim h$$

$$\mathcal{K} \cup T \not\mid\!\sim \bot$$

The pair $\langle T, h \rangle_{\mathcal{K}}$ *is called a* theory structure.

REMARKS: We use $\Gamma \mid\!\sim \alpha$ as an abbreviation for "α is a defeasible consequence of Γ". This *defeasible consequence* relation is defined in terms of "$\vdash$" augmented with the *modus ponens* for "$\succ\!\!-$". Whenever possible we will drop the reference to the context and we will write $\langle T, h \rangle$ meaning $\langle T, h \rangle_{\mathcal{K}}$. We will refer to the collection of all possible theory structures as $\mathbf{TSTRUC}_{\boldsymbol{\Delta}}$ or just **TSTRUC**. There is a distinguished theory, for any context $\mathcal{K}$, $\langle \emptyset, \mathcal{K}^{\vdash} \rangle$, *i.e.* no theory is necessary to support the conjunction of the atoms of the deductive closure ($\mathcal{K}^{\vdash}$) of the knowledge in $\mathcal{K}$. Finally, for $\langle T, h \rangle$ we will assume that the subset T of $\boldsymbol{\Delta}$ is minimal in the sense of not containing any rule that is unnecessary in the inference of h.

DEFINITION **4.2 (Subtheories)** *Let $\langle T, h\rangle$ be a theory structure for h, and $\langle S, j\rangle$ a theory structure for j such that $S \subseteq T$. We will say that $\langle S, j\rangle$ is a subtheory of $\langle T, h\rangle$ and use the notation $\langle S, j\rangle \subseteq \langle T, h\rangle$.*

Having defined these objects we would like to establish certain binary relations on $\mathbf{TSTRUC}_{\Delta}$ in such a way that it would help us to choose the "better" theory structure that supports a conclusion. The following definitions will characterize this relation.

Let $\mathcal{S}_C$ be the set of sentences formed with the elements in C, the contingent part of our knowledge. The following is essentially Poole's definition of specificity [8].

DEFINITION **4.3 (Strict-Specificity)** *Given two theory structures $\langle T_1, h_1\rangle$, $\langle T_2, h_2\rangle$ in* **TSTRUC**, *we say that T_1 for h_1 is strictly more specific than T_2 for h_2 denoted by $\langle T_1, h_1\rangle \succ_{\mathbf{spec}} \langle T_2, h_2\rangle$, if and only if*

1. $\forall e \in \mathcal{S}_C$ *such that* $\mathcal{N} \cup \{e\} \cup T_1 \mid\sim h_1$ *also* $\mathcal{N} \cup \{e\} \cup T_2 \mid\sim h_2$, *and*
2. $\exists e \in \mathcal{S}_C$ *such that:*

$$\mathcal{N} \cup \{e\} \cup T_2 \mid\sim h_2 \quad \text{(Activates } T_2\text{)},$$

$$\mathcal{N} \cup \{e\} \cup T_1 \not\mid\sim h_1 \quad \text{(Does not activate } T_1\text{)}$$

$$\mathcal{N} \cup \{e\} \not\vdash h_2 \quad \text{(Non-triviality condition)}$$

DEFINITION **4.4 (Equi-specificity)** *Two theory structures T_1 for h_1 and T_2 for h_2 are* equi-specific, *denoted by $\langle T_1, h_1\rangle \approx_{\mathbf{spec}} \langle T_2, h_2\rangle$, when the following condition holds, $\forall e \in \mathcal{S}_C$, $\mathcal{N} \cup \{e\} \cup T_1 \mid\sim h_1$ if and only if $\mathcal{N} \cup \{e\} \cup T_2 \mid\sim h_2$.*

DEFINITION **4.5 (Specificity)** *We say that a theory structure T_1 for h_1 is at least as specific as a theory structure T_2 for h_2 denoted by $\langle T_1, h_1\rangle \succeq_{\mathbf{spec}} \langle T_2, h_2\rangle$, if and only if $\langle T_2, h_2\rangle \approx_{\mathbf{spec}} \langle T_1, h_1\rangle$ or $\langle T_1, h_1\rangle \succ_{\mathbf{spec}} \langle T_2, h_2\rangle$.*

Theories are objects that represent "pieces" of the reasoning process. They relate to each other in different ways. Some interesting "relationships" follow.

DEFINITION **4.6 (Disagreement)** *We say that two theory structures T_1 for h_1 and T_2 for h_2* disagree, *denoted by*

$$\langle T_1, h_1\rangle \bowtie_{\mathcal{K}} \langle T_2, h_2\rangle$$

if and only if $\mathcal{K} \cup \{h_1, h_2\} \vdash \bot$

DEFINITION **4.7 (Counterargument)** *We say that a theory structure T_1 for h_1 counterargues a theory structure T_2 for h_2 at h, denoted by*

$$\langle T_1, h_1\rangle \otimes\!\!\!\!\!\!\rightarrow^{h} \langle T_2, h_2\rangle$$

if and only if there exists a subtheory $\langle T, h\rangle$ of $\langle T_2, h_2\rangle$ such that $\langle T_1, h_1\rangle \bowtie_{\mathcal{K}} \langle T, h\rangle$ i.e. $\langle T, h\rangle \subset \langle T_2, h_2\rangle$ and $\mathcal{K} \cup \{h_1, h\} \vdash \perp$.

DEFINITION **4.8 (Defeat)** *We say that a theory structure T_1 for h_1 defeats a theory structure T_2 for h_2 denoted by*

$$\langle T_1, h_1\rangle \gg_{\mathbf{def}} \langle T_2, h_2\rangle$$

if and only if there exists a subtheory structure $\langle T, h\rangle$ of $\langle T_2, h_2\rangle$ such that:

$\langle T_1, h_1\rangle \otimes\!\!\!\!\!\!\rightarrow^{h} \langle T_2, h_2\rangle$ *i.e. T_1 for h_1 counterargues T_2 for h_2 at h, and*

$\langle T_1, h_1\rangle \succ_{\mathbf{spec}} \langle T, h\rangle$ *i.e. T_1 for h_1 is more specific than T for h.*

DEFINITION **4.9 (Levels)** *Theories are active at various levels as supporting or interfering theories.*

1. *All theories are level-0 S-theories (supporting theories) and I-theories (interfering theories).*
2. *A theory $\langle T_1, h_1\rangle$ is a level $n+1$ S-theory if and only if there is no level n I-theory $\langle T_2, h_2\rangle$ such that for some h, $\langle T_2, h_2\rangle$ counterargues $\langle T_1, h_1\rangle$ at h, i.e. $\nexists \langle T_2, h_2\rangle \in \mathbf{TSTRUC}$ such that, for some h, $\langle T_2, h_2\rangle \otimes\!\!\!\!\!\!\rightarrow^{h} \langle T_1, h_1\rangle$.*
3. *A theory $\langle T_1, h_1\rangle$ is a level $n+1$ I-theory if and only if there is no level n I-theory $\langle T_2, h_2\rangle$ such that $\langle T_2, h_2\rangle$ defeats $\langle T_1, h_1\rangle$.*

REMARK: A level n S-theory will be denoted by S^n-theory and a level n I-theory will be denoted by I^n-theory.

DEFINITION **4.10 (Justification)** *We say that a theory $\langle T, h\rangle$* justifies *p if and only if there exists a subtheory $\langle R, p\rangle$ of $\langle T, h\rangle$ for which there exists m such that for all $n \geq m$ $\langle R, p\rangle$ is an S^n-theory for p. We say that p is* justified in $\Omega \subseteq \mathbf{TSTRUC}$ *if there is a $\langle T, h\rangle \in \Omega$ that justifies p.*

References

[1] Guillermo R. Simari. On the logic of defeasible reasoning. Technical Report WUCS-89-12, Washington University, Department of Computer Science, April 1989.

[2] Donald Nute and Michael Lewis. d-Prolog: A Users Manual. Technical Report ACMC 01-0017, University of Georgia, Athens, 1986.

[3] Donald Nute. Defeasible reasoning. In James H. Fetzer, editor, *Aspects of Artificial Intelligence*, pages 251–288. Kluwer Academic Publishers, Norwell, MA, 1988.

[4] W.F.Clocksin and C.S.Mellish. *Programming in Prolog.* Speinger-Verlag, third edition, 1988.

[5] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[6] Keith L. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[7] Lee Naish. *Negation and Control in Prolog (LNCS-238).* Springer-Verlag, 1986.

[8] David L. Poole. On the comparison of theories: Preferring the most specific explanation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 144–147. IJCAI, 1985.

[9] John L. Pollock. Defeasible reasoning. *Cognitive Science*, 11:481–518, 1987.