Bill J. Ellis
March 18, 2002

SPADE Rules - Logic, Tricks and NuSPADE

1 Introduction

A key requirement for NuSPADE is to show by example that everything it proves can be repeated inside SPADE. In practise this means that NuSPADE will generate command logs that can be executed inside SPADE to prove a current problem. Thus a fundamental problem in building NuSPADE is determining how to correctly make use of the rules available in SPADE. This can be split into two separate issues:

- **Logic -** What the SPADE rules logically mean.
- **Limits -** What limits SPADE imposes on using rules, even though such use is logically sound.

This note tries to capture understanding of these two problems.

2 SPADE Logic

2.1 Rule Location and Context

Rules are loaded into SPADE through a *.RLS, *.RUL or *.RLB file. In all of these cases the rules take the same form. Rules are often supplemented with some Functional Description Language (FDL) held in a *.FDL file. These add type information to the identifiers of the rules. This type information is used inside SPADE to constrain the application of rules.

2.2 Declaring Rules

The rules themselves are classified as either substitution rules or inference rules. The following two tables list the different rule forms and observations about their use:

Substitution	**Observation**
X may_be_replaced_by Y.	X and Y may contain meta-variables. Every meta-variable in Y must be in X. Any hypothesis or conclusion that has a sub-term that matches with X may be replaced by Y, instantiating meta-variables accordingly.
X may_be_replaced_by Y if [CONDITIONS].	Same as X may_be_replaced_by Y. Except now the sub-term replacement is only allowed where the list of conditions holds true. Each condition may make reference to meta-variables.
X & Y are_interchangeable.	Has the same effect as if X may_be_replaced_by Y and Y may_be_replaced_by X were both available. As before every meta variable in Y must be in X and, due to the bidirectional nature, every meta variable in X must be in Y. Thus X and Y must share the same meta variables.
X & Y are_interchangeable if [CONDITIONS].	Same as X & Y are_interchangeable. Except now it must also be shown that the list of conditions hold true.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the Clam-Spark project (EPSRC GR/R24081).

Inference	Observation
X may_be_deduced.	X may contain meta-variables. A new hypothesis X can be inserted into the current VC, instantiating all of the meta-variables.
X may_be_deduced if [CONDITIONS].	Same as X may_be_deduced. Except now the new hypothesis X can only be inserted where the list of conditions hold true. Each condition may make reference to meta-variables.
X may_be_deduced_from [Y].	X and Y may contain meta-variables. Allows new hypothesis X to be inserted, instantiating all of its meta-variables, where a list of hypotheses Y are available.
X may_be_deduced_from [Y] if [CONDITIONS].	Same as X may_be_deduced_from [Y]. Except now it must also be shown that the list of conditions hold true.

2.3 Logic Behind Rules

Each rule will ultimately be based on a formula. This formula with either an equality ($X = Y$), Boolean equivalence ($X \leftrightarrow Y$) or implication ($X \rightarrow Y$). In [1] section 6.4, a method is outlined to convert SPADE rules into VCs, such that the rules themselves can be proved inside SPADE. This is important, as it shows the appropriate logical formula that maps to each of the different kinds of rules in SPADE. The mappings are summarised in the two tables below:

Substitution	Prove As Formula
X may_be_replaced_by Y.	$X = Y$
X may_be_replaced_by Y if [CONDITIONS].	$CONDITIONS' \rightarrow (X = Y)$
X & Y are_interchangeable.	$X = Y$
X & Y are_interchangeable if [CONDITIONS].	$CONDITIONS' \rightarrow (X = Y)$

Inference	Prove As Formula	
X may_be_deduced.	X	
X may_be_deduced if [CONDITIONS].	$CONDITIONS' \rightarrow (X)$	$\star$
X may_be_deduced_from [Y].	$Y' \rightarrow X$	
X may_be_deduced_from [Y] if [CONDITIONS].	$CONDITIONS' \rightarrow (Y' \rightarrow X)$	$\star$

Where A' means the list of formulas A is presented as a single formula of conjunctions. For example, if $A = [I, J, K]$ then $A' = I$ and J and K. The rows marked $\star$ are not explicitly mentioned in [1]. Reasonable proposals are made for the formula of these rows.

2.4 Expected Rewrite Rules From Formulas

The expected rewrite rules that will be made available from each formula are shown below:

Formula	Rewriting Subterm Of	Rewrite Rule(s)
$X = Y$	Hypothesis	$X \Rightarrow Y$, $Y \Rightarrow X$
	Conclusion	$X \Rightarrow Y$, $Y \Rightarrow X$
$X \rightarrow Y$	Hypothesis	$X \Rightarrow Y$
	Conclusion	$Y \Rightarrow X$

All other formulas can be expressed in terms of those above. However, to help elaborate on the SPADE rules and their formula equivalents, some more formula are considered below:

Formula	Rewriting Subterm Of	Rewrite Rule(s)
$(X \leftrightarrow Y) = ((X \rightarrow Y)$ and $(Y \rightarrow X))$	Hypothesis	$X \Rightarrow Y$, $Y \Rightarrow X$
	Conclusion	$X \Rightarrow Y$, $Y \Rightarrow X$
$X = (X \leftrightarrow \mathit{true}) = ((X \rightarrow \mathit{true})$ and $(\mathit{true} \rightarrow X))$	Hypothesis	$X \Rightarrow \mathit{true}$, $\mathit{true} \Rightarrow X$
	Conclusion	$X \Rightarrow \mathit{true}$, $\mathit{true} \Rightarrow X$

Note that to use the same rewrite rule $X \Rightarrow Y$ on both hypothesis and conclusions requires either an equality or equivalence supporting formula.

3 SPADE Restrictive Mode To Support NuSPADE

NuSPADE will require some mechanism to use its proof planning steps inside SPADE. It is likely that this mechanism will be implemented using a number of SPADE commands. Some of these commands will have additional side effects to their intended purpose. For example, as default, simplification is performed after an `infer` command. Unfortunately, these efforts can interfere with the natural proof plans envisaged for NuSPADE.

In many cases these side effects can be switched off by setting flags accordingly. This effectively puts SPADE in a restrictive mode, where it only ever does what it is explicitly told to do. The existence of such a mode is of great value to NuSPADE. However, from an interactive viewpoint, this restrictive mode will require many extra keystrokes from the user.

Thus it is expected that all NuSPADE generated commands will be executed inside SPADE during restrictive mode. Interaction with the user will be performed inside SPADE as normal. Switching in and out of these two modes can be achieved by setting the appropriate flags. This will be done as part of the generated command logs. The initial flags identified for entering SPADE restrictive mode are:

```
set auto_done to off.
set simplify_in_infer to off.
set simplify_during_load to off.
```

There are some instances where flags cannot be set to ensure SPADE only does what it is told. These are considered below.

3.1 SPADE During Infer Command

The SPADE `infer` command is of the form:

```
infer FORMULA
      using RULES
      from HYPOTHESIS_LIST.
```

During execution of this command SPADE tries to infer `FORMULA` using the constrained information provided. Where it is unsuccessful it presents the user with the option of subgoaling. Unfortunately, `infer` does not strictly operate within the user constrained search. It has several additional rules and facilities built-in. These have been traced to the section of code show in Figure 1. The predicate `see_if_can_infer/1` makes use of many predicates `do_infer/1` that implement directly a number of logic rules.

There are a number of points where `infer/1` is used in performing the `infer` command. However, these can all be traced to the root predicate `fulfil/2` shown in Figure 2. The predicates `try_to_satisfy/2` and `match_up/2` both lead to use of the `infer/1`.

```
infer(A) :-
        (
           simplify_in_infer(on),
           simplify(A, B),
           !,
           (
              see_if_can_infer(B)
           ;
              A \= B,
              see_if_can_infer(A)
           ;
              standardise_in_infer(on),
              norm_typed_expr(A, boolean, C),
              see_if_can_infer(C)
           )
        ;
           see_if_can_infer(A)
        ;
           standardise_in_infer(on),
           norm_typed_expr(A, boolean, B),
           A \= B,
           see_if_can_infer(B)
        ), !.
```

Figure 1: inferenc2.pl: starting line 40

```
fulfil([], []) :- !.
fulfil(GOALS, UNSATISFIED_GOALS) :-
    split(GOALS, PRIMARIES,SECONDARIES, REST),
    try_to_satisfy(PRIMARIES, UNSATISFIED_PRIMARIES),
    (
       UNSATISFIED_PRIMARIES=[],
       match_up(SECONDARIES, UNSATISFIED_SECONDARIES)
    ;
       UNSATISFIED_PRIMARIES\=[],
       UNSATISFIED_SECONDARIES=SECONDARIES
    ),
    append(UNSATISFIED_PRIMARIES, UNSATISFIED_SECONDARIES, UNSATISFIED_FRONT),
    (
       REST=[],
       UNSATISFIED_GOALS=UNSATISFIED_FRONT
    ;
       REST\=[],
       do_direct_goals(REST, REMAINDER),
       fulfil(REMAINDER, UNSATISFIED_REMAINDER),
       append(UNSATISFIED_FRONT, UNSATISFIED_REMAINDER, UNSATISFIED_GOALS)
    ).
```

Figure 2: infer2.pl: starting line 464

3.2 SPADE After Infer Command

After every SPADE `infer` command the section of code in Figure 3 is executed. While the flag `simplify_in_infer/1` can be switched off to avoid simplification, there is no setting to avoid calling `flatten/2`. This predicate performs VC wide simplification, removing conjuncts in favour of having shorter, more numerous, hypothesis and conclusions.

```
layout(MS, L, F) :-
    (
       simplify_in_infer(on),
       simplify(F,FF)
    ;
       FF = F
    ),
    !,
    flatten(FF,F1),
    !,
    store_vc(MS, L, 1, F1).
```

Figure 3: loadvc5.pl: starting line 1722

3.3 Determining Side Effects Requires Defining Command

The two key SPADE commands `replace` and `infer` have clear definitions. Only through this clear definition is it possible to identify the additional behaviour that can be classed as side effects. Less well understood commands, such as `done`, present a problem. The help text for `done`, presented when the command `help done` is issued inside SPADE, is presented below:

> The "done" command checks to see if any of the current conclusions can be inferred reasonably directly from the current hypotheses and, if so, these conclusions are taken to be true (under current conditions, i.e. taking account cases, etc.).

The phrase *reasonably directly* allows `done` a great deal of flexibility in its behaviour. It is arguably impossible for done to have a side effect as any action it takes could be seen as reasonable. Fortunately, it is likely that NuSPADE will only need to use these kinds of commands to discharge the final steps of a proof. In fact CLAM, the key proof planning component of NuSPADE, originally generated tactics that made use of the somewhat woolly OYSTER tactic `simplify`.

Thus, so long as NuSPADE can relay on some tested subset of the less well defined commands, any extra behaviour available does not present a problem.

3.4 Add New Restrictive Mode Setting

It will be necessary for NuSPADE to have some mechanism to deal with these unrequested side effects. A collection of possible solutions are presented below:

- **Use another command -** Where SPADE commands have undesirable side effects, a different command might be used instead. Unfortunately, as each SPADE command performs a discrete function, there is little opportunity to use another command in its place.
- **Undo the side effects -** Where the troublesome SPADE commands must be used, their undesirable side effects might be undone by a further sequence of commands. Implementing this will require knowing in detail the exact side effects that SPADE will perform, so that their inverse can be automatically inserted. It will also be necessary to implement the undo sequence of commands using SPADE commands that do not have side effects. This may not always be possible.
- **Accept the side effects -** While converting the NuSPADE proof plan into SPADE the side effects are taken into account. This will require similar knowledge as would be required for inserting a sequence of undo commands. Only now the side effects are accepted and incorporated into the generated NuSPADE command proof. This is always possible to implement. However, it will require much more effort in presenting the NuSPADE proof plan to SPADE. For example, after each implication rewrite it may be necessary to radically restructure the current

problem. This could affect the structure of expressions and the numbering of hypotheses and conclusions. Most significantly these changes will need to be related to the current point in the NuSPADE proof plan.

- **Remove the side effects -** In CS-note 6 some core principles were established concerning the development of NuSPADE. In particular there was to be no changes to the copy of SPADE used as part of NuSPADE. These principles, however, do not restrict insights gained from developing NuSPADE to be suggested to the developers of SPADE as potential modifications.

  A new setting, or settings, could be introduced to switch off all problematic side effects in SPADE. The only areas that would need changes are those where settings do not already exist. However, it may be desirable to have a single system wide setting that turns off all side effects. This would override some exiting settings and be added where settings do not exist.

The advantage of the latter suggestion is it only requires the addition of a single setting, makes it quite clear in the SPADE source where side effects are taking place and makes it straight forward for NuSPADE to flip in and out of restrictive mode without changing the users setting preferences. An example section of code to illustrate the idea of this single setting is shown in Figure 4.

```
layout(MS, L, F) :-
    (
       simplify_in_infer(on),
       restrictive_mode(off),   %<==May override user setting to enforce the
                                %   restrictive mode.
       simplify(F,FF)
    ;
       FF = F
    ),
    !,
    (
       restrictive_mode(off),   %<==May override the default behaviour to
                                %   enforce restrictive mode.
       flatten(FF,F1)
    ;
       F1=FF
    ),
    !,
    store_vc(MS, L, 1, F1).
```

Figure 4: Restrictive mode version of loadvc5.pl: starting line 1722

4 Rewriting Based On Equality

All of the SPADE substitution rules are based on an equality formula of the form $CONDITIONS' \rightarrow (X = Y)$, where $CONDITIONS'$ may be true. Therefore it should be possible, where the conditions are satisfied, to use rewrite rules of the form $X \Rightarrow Y$ and $Y \Rightarrow X$ on both hypothesis and conclusions. Fortunately, this is directly supported in SPADE through the `replace` command. This takes the general form:

```
replace [HYPOTHESIS|CONCLUSION]#NUMBER:
        SUB_TERM by NEW_SUBTERM
using SUBSTITUTION_RULE.
```

The only issue for equality rewriting is SPADE limits placed on substitutions that are declared for one-way rewriting, even though their supporting formula logically supports two-way rewriting. A suitable trick is shown below to deal with this case.

4.1 SPADE reversing "X may_be_replaced_by Y" Proof Trick

Inside SPADE the rule X may_be_replaced_by Y only allows one-way rewriting from X to Y. However, the corresponding formula $X = Y$ suggests that the rule is powerful enough to allow Y may_be_replaced_by X. This is remarked upon in [1] section 5.2, which proposes a method for using a X may_be_replaced_by Y rule to perform a Y may_be_replaced_by X operation. This is best illustrated by example:

Suppose the following rule is available:

```
defsucc(1): A+1 may_be_replaced_by succ(A).
```

And the VC being tackled is:

```
H1:     true.
         ->
C1:     succ(x)-1=x.
```

The conclusion `C1` can not be rewritten using `defsucc(1)` as there is no occurrence of pattern `A+1`. Were the rule applicable in reverse, the pattern `succ(A)` would match with `succ(x)` allowing this to be replaced with `x+1`, leading to the numerically true `C1`, `x+1-1=x`. Recognising that the reverse of this rule would be beneficial, a new hypothesis is inferred that equates the expected instanciated replaced sub-term with itself:

```
infer x+1=x+1 using nuspaderules(1).
```

Surprisingly, a rule of the form X=X may_be_deduced is not included as part of the SPADE base rules. Thus a suitable extra rule `nuspaderule(1)` is made available.

```
rule_family nuspaderules:
    X = Y requires [ X:any, Y:any ].

nuspaderules(1): X=X may_be_deduced.
```

Note however, that the SPADE `done` command makes use of built in reasoning that includes the logic of this rule. The actual code for this is in inferenc2.pl line 82:

```
do_infer(A=A) :- !.
```

An alternative method to declaring extra rules is thus:

```
infer x+1=x+1 using inference(1). y. y.
done.
```

This latter method may be preferred as it builds directly on SPADE. Whichever style of proof is used for showing `A=A`, the new hypothesis will become available in the VC:

```
H1:     true.
H2:     x+1=x+1.
         ->
C1:     succ(x)-1=x.
```

The `defsucc(1)` rule may be used on either side of the new hypothesis `H2`. Applying it to the left-most side:

```
replace h#2: x+1 by succ(x) using defsucc(1). 1. y. n.
```

Gives the modified VC:

```
H1:     true.
H2:     succ(x)=x+1.
         ->
C1:     succ(x)-1=x.
```

The SPADE built in rule `eq(1)` can then be instantiated with `H2` to allow the reverse rewrite that is known to lead to a proof:

```
eq(1):    X may_be_replaced_by Y if [ X=Y, goal(X\=Y) ].
```

The required rewrite is performed:

```
replace c#1: succ(x) by x+1 using eq(1). y. n.
```

Giving the final VC:

```
H1:    true.
H2:    succ(x)=x+1.
        ->
C1:    x+1-1=x.
```

The existence of this trick shows how one-way rewriting can be reversed. Thus the one-way rules exist not to constrain rewriting in the other direction, rather to promote rewriting in this direction in any SPADE proof searches. This is summarised in [1] page 82 as:

> ... one-way rules do not strictly affect the actual power of a proof rule, but merely suggest a particular common mode of use.

4.2 Hypothesis Rewriting

Assuming that SPADE is operating in restrictive mode, the plan shown in Figure 5 may be used to equality rewrite a hypothesis, in both directions, irrespective as to how the rule is declared in SPADE. Note that this will, and should, introduce a new hypothesis rather than replace the original. This is desired to support cases where the same starting hypothesis has more than one rewrite rule applied to it.

4.2.1 Example

A concrete example of this rewrite plan is given below. Suppose the following rule is available:

```
rule(1): a may_be_replaced_by b.
```

And the VC being tackled is:

```
H1:  b or c
-->
C1:  c or d
```

It should be possible to rewrite `b or c` into `a or c` using `rule(1)` as a hypothesis equality rewrite rule. The first step is to newgoal on the required new hypothesis, in this case `a or c`:

```
infer a or c using inference(1). y. y.
```

This gives the new goal VC1:

```
H1:  b or c
-->
C1:  a or c
```

The next step is to introduce the substitution that is justifying the rewrite:

```
infer a = b using inference(1). y. y.
```

This gives the new goal VC2:

```
H1:  b or c
-->
C1:  a = b
```

This can be proved through being rewritten into an obvious equality using `rule(1)`:

```
replace c#1: a by b using rule(1). y. n.
infer c#1 using nuspaderules(1).
```

Or if preferred the `done` command can be used instead of the extra inserted rule `nuspaderules(1)`:

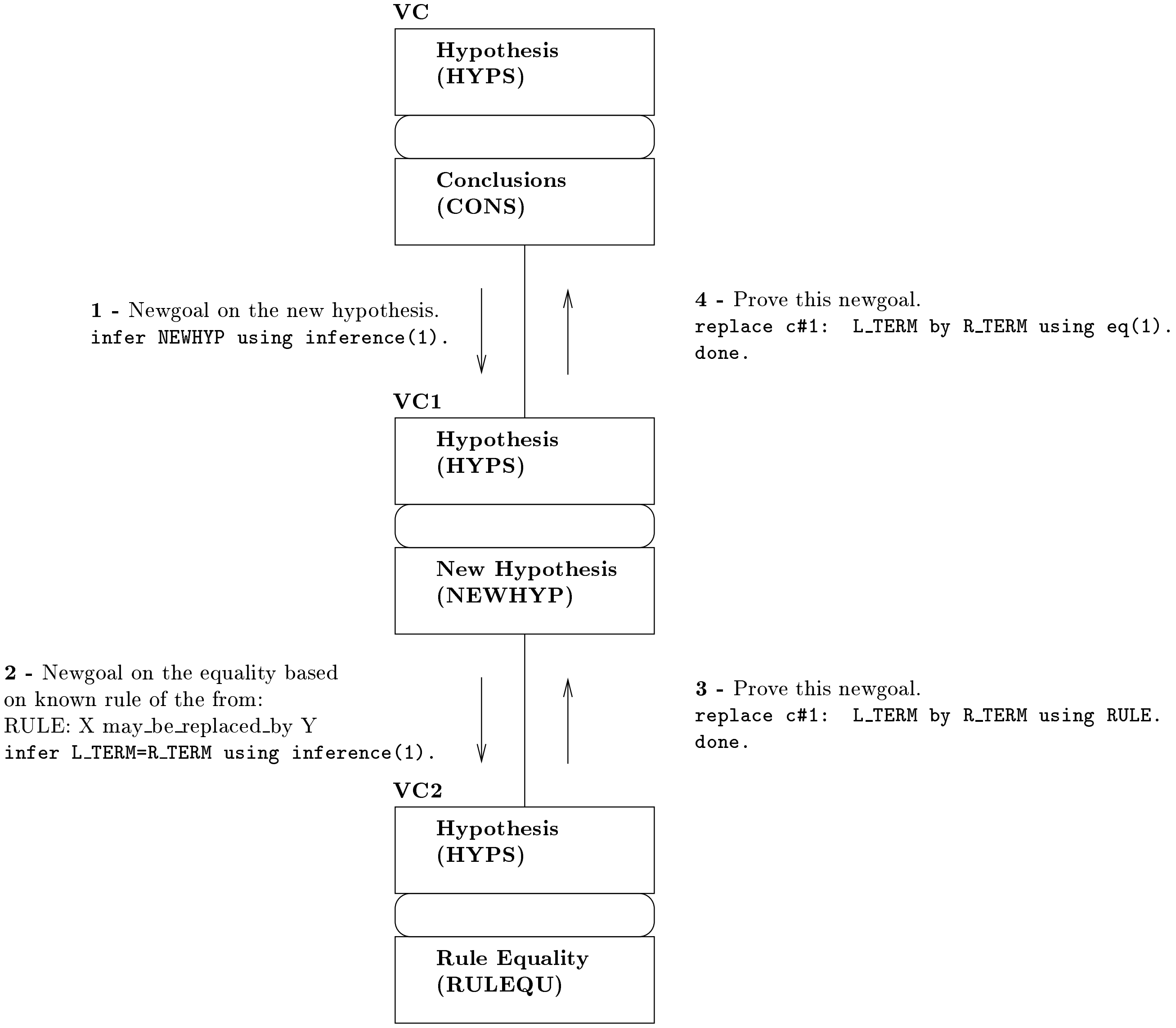

Figure 5: Hypothesis equality rewrite plan

```
replace c#1: a by b using rule(1). y. n.
done.
```

Either of these returns to goal VC1, with the equality available as a new hypothesis:

```
H1:  a or b
H2:  a = b
-->
C1:  a or c
```

The SPADE built in rule `eq(1)` can then be instantiated with `H2` to allow rewriting `C1` to `a or b`. This then matches hypothesis `H1`:

```
replace c#1: a by b using eq(1). y. n.
done.
```

Giving the final VC:

```
H1:  b or c
H2:  a or c
-->
C1:  c or d
```

4.3 Conclusion Rewriting

Assuming that SPADE is operating in restrictive mode, the plan shown in Figure 6 may be used to equality rewrite a conclusion. Note that this will replace the existing conclusion rather than introduce a new conclusion. This behaviour is essential, otherwise the initial conclusion would still need to be solved.

4.3.1 Example

A concrete example of this rewrite plan is given below. Suppose the following rule is available:

```
rule(2): a may_be_replaced_by b.
```

And the VC being tackled is:

```
H1:  true
-->
C1:  a or b
```

It should be possible to rewrite `a or b` into `a or a` using `rule(2)` as a conclusion equality rewrite rule. The first step is to newgoal on the required new conclusion, in this case `a or a`:

```
infer a or a using inference(1). y. y.
```

This gives the new goal VC1:

```
H1:  true
-->
C1:  a or a
```

Assuming by some means this is solved, the VC will become:

```
H1:  true
H2:  a or a
-->
C1:  a or b
```

It is then necessary to justify that the extra information gained from proving VC1 is sufficient to prove the original VC. The next step is to introduce the equality that justified the rewrite:

```
infer a = b using inference(1). y. y.
```

This gives the new goal VC2:

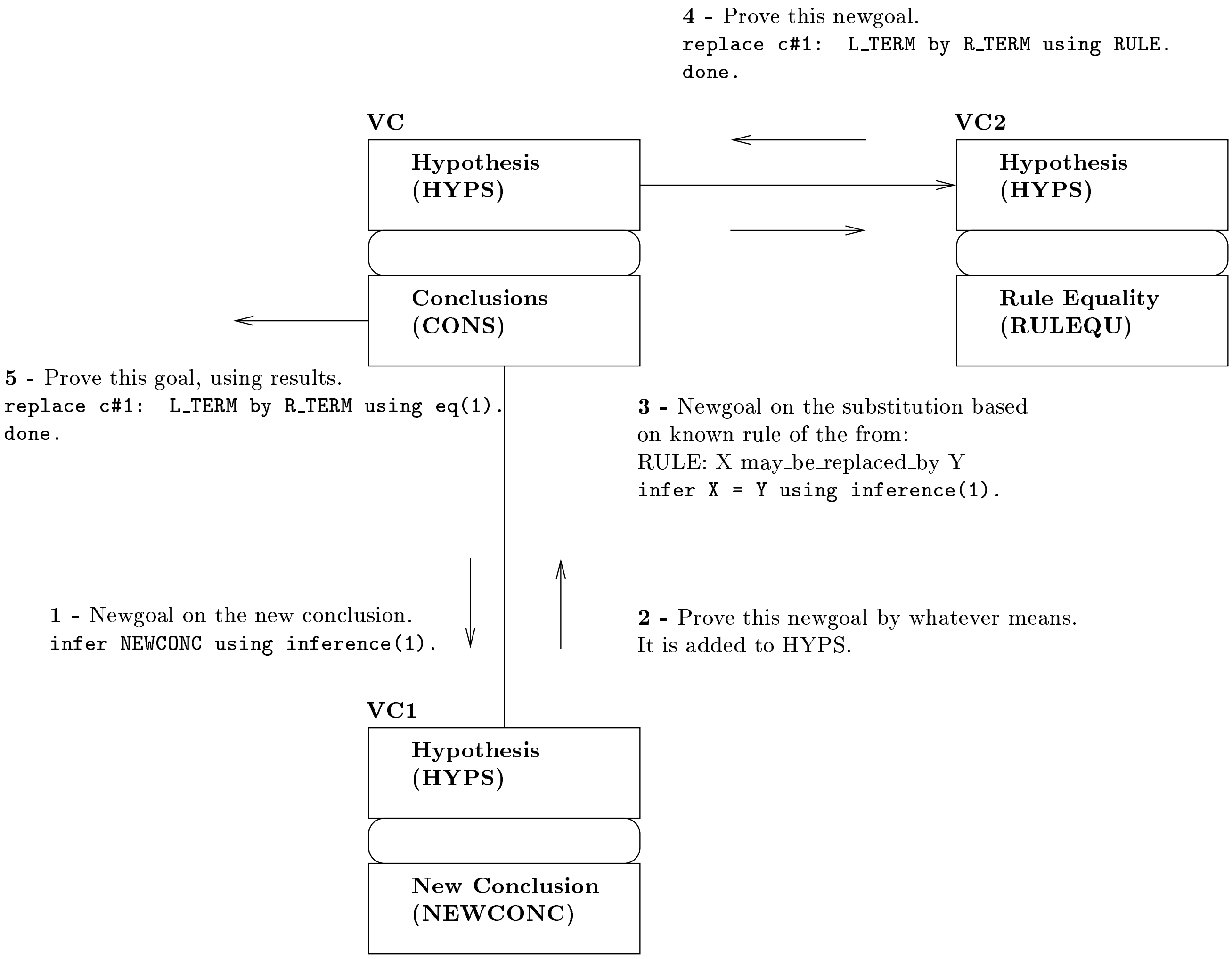

Figure 6: Conclusion equality rewrite plan

```
H1:  true
H2:  b or b
-->
C1:  a = b
```

This can be proved through being rewritten into an obvious equality using `rule(2)`:

```
replace c#1: a by b using rule(2). y. n.
infer c#1 using nuspaderules(1).
```

Or if preferred the `done` command can be used instead of the extra inserted rule `nuspaderules(1)`:

```
replace c#1: a by b using rule(2). y. n.
done.
```

Either of these will return to the original VC, with the substitution available as a new hypothesis:

```
H1:  true
H2:  b or b
H3:  a = b
-->
C1:  a or b
```

The SPADE built in rule `eq(1)` can then be instantiated with `H3` to allow rewriting `C1` to `b or b`. This then matches hypothesis `H2`, proving the original VC:

```
replace c#1: a by b using eq(1). y. n.
done.
```

5 Rewriting Based On Implication

All of the SPADE inference rules are indirectly based on an implication formula of the form $(CONDITIONS' \rightarrow X)$ or $CONDITIONS' \rightarrow (X \rightarrow Y)$. Each of these will be considered separately after some notes on using the SPADE `infer` command.

5.1 Notes On SPADE `infer` Command

The substitution rules in SPADE can be used as equality rewrite rules through the `replace` command. However, the equivalent command for inference, `infer`, is not as accommodating. This command takes the general form:

```
infer FORMULA
      using RULES
      from HYPOTHESIS_LIST.
```

Where the formula can not be inferred, it allows the user the option of newgoaling[1]. Newgoaling is quite similar to subgoaling. The difference can be explained in terms of the cut rule supporting the inference of new formulas:

Modify:
A new goal conducted in the same proof context as **Problem**.

Justify:
Proof that proving **Modify** is proving **Problem**.

$$\frac{\overbrace{\Gamma \vdash \alpha} \qquad\qquad \overbrace{\Gamma, \alpha \vdash \beta}}{\underbrace{\Gamma \vdash \beta}}$$

Problem:
The original goal.

[1] See CS-note 4 section 4 for a technical discussion of newgoaling in SPADE.

- **Newgoaling -** A newgoal attempt is a proof of **Modify**. If **Modify** is shown the problem becomes **Justify**. It is quite possible that this will be solved without using the extra information gained through **Modify**.

  Newgoaling places emphasis on adding new hypotheses to the current problem.

- **Subgoaling -** A subgoal attempt is a proof of **Justify**, usually automatically. Where **Justify** is shown the problem becomes **Modify**. To perform **Justify** automatically the forms of **Modify** are restricted.

  Subgoaling places emphasis on changing the current problem.

The differences between these two are at a conceptual rather than logical level. It is quite possible to use SPADE newgoaling to implement subgoaling. This is very important as subgoaling is the logical device supporting rewriting. This issue was avoided in the substitution rules due to the power of `replace`.

5.2 Hypothesis Rewriting

Assuming that SPADE is operating in restrictive mode, the plan shown in Figure 7 used to implication rewrite a hypothesis. Note that this will, and should, introduce a new hypothesis rather than replace the original. This is desired to support cases where the same starting hypothesis has more than one rewrite rule applied to it.

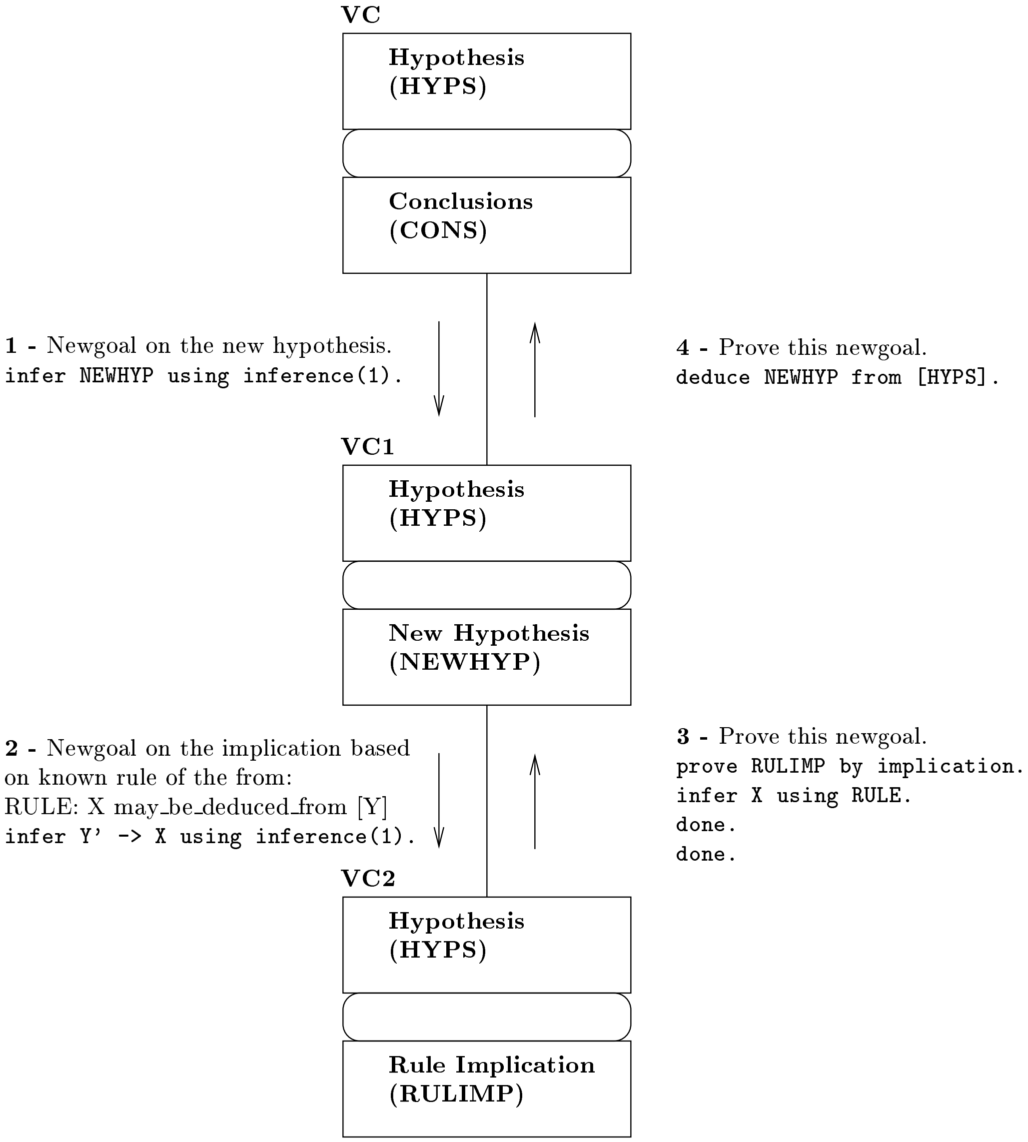

Figure 7: Hypothesis implication rewrite plan

5.2.1 Example

A concrete example of this rewrite plan is given below. Suppose the following rule is available:

```
rule(3): a may_be_deduced_from [b].
```

And the VC being tackled is:

```
H1:  a or b
-->
C1:  c or d
```

It should be possible to rewrite `a or b` into `a or a` using `rule(3)` as a hypothesis implication rewrite rule. The first step is to newgoal on the required new hypothesis, in this case `a or a`:

```
infer a or a using inference(1). y. y.
```

This gives the new goal VC1:

```
H1:  a or b
-->
C1:  a or a
```

The next step is to introduce the implication that is justifying the rewrite:

```
infer b->a using inference(1). y. y.
```

This gives the new goal VC2:

```
H1:  a or b
-->
C1:  b -> a
```

This can be proved using a proof by implication to introduce `b` as a hypothesis, then inferring `a` from `b` using `rule(3)`:

```
prove c#1 by implication.
infer a using rule(3).
done.
done.
```

This returns to goal VC1, with the implication available as a new hypothesis:

```
H1:  a or b
H2:  b -> a
-->
C1:  a or a
```

The `deduce` command is powerful enough to discharge these:

```
deduce c#1 from [1,2].
done.
```

Giving the final VC:

```
H1:  a or b
H2:  a or a
-->
C1:  c or d
```

5.3 Conclusion Rewriting

Assuming that SPADE is operating in restrictive mode, the plan shown in Figure 8 may be used to implication rewrite a conclusion. Note that this will replace the existing conclusion rather than introduce a new conclusion. This behaviour is essential, otherwise the initial conclusion would still need to be solved.

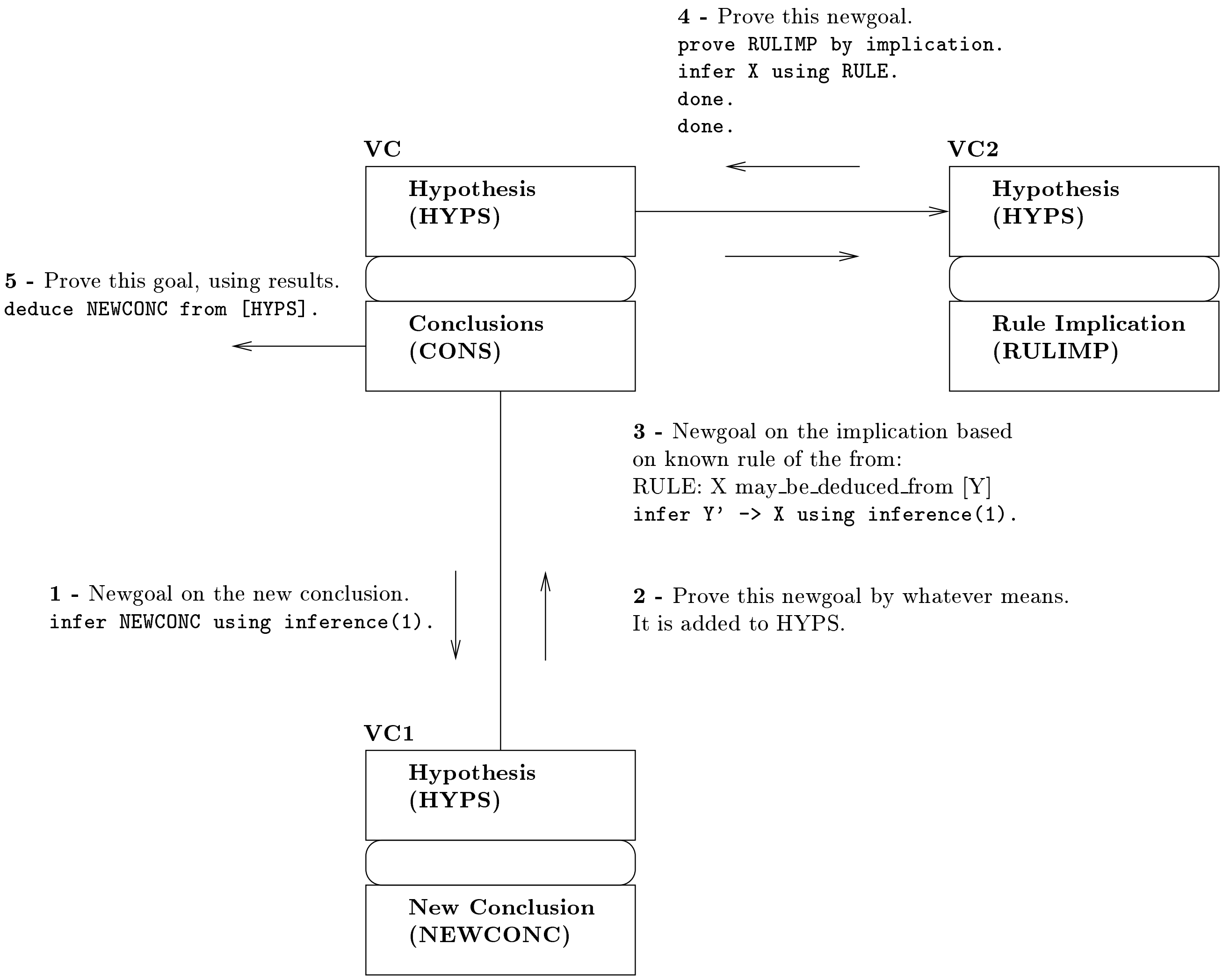

Figure 8: Conclusion implication rewrite plan

5.3.1 Example

A concrete example of this rewrite plan is given below. Suppose the following rule is available:

```
rule(4): a may_be_deduced_from [b].
```

And the VC being tackled is:

```
H1:  true
-->
C1:  a or b
```

It should be possible to rewrite `a or b` into `b or b` using `rule(4)` as a conclusion implication rewrite rule. The first step is to newgoal on the required new conclusion, in this case `b or b`:

```
infer b or b using inference(1). y. y.
```

This gives the new goal VC1:

```
H1:  true
-->
C1:  b or b
```

Assuming by some means this is solved, the VC will become:

```
H1:  true
H2:  b or b
-->
C1:  a or b
```

It is then necessary to justify that the extra information gained from proving VC1 is sufficient to prove the original VC. The next step is to introduce the implication that justified the rewrite:

```
infer b->a using inference(1). y. y.
```

This gives the new goal VC2:

```
H1:  true
H2:  b or b
-->
C1:  b -> a
```

This can be proved using a proof by implication to introduce `b` as a hypothesis, then inferring `a` from `b` using `rule(4)`.

```
prove c#1 by implication.
infer a using rule(4).
done.
done.
```

This returns to the original VC, with the implication available as a new hypothesis:

```
H1:  true
H2:  b or b
H3:  b -> a
-->
C1:  a or b
```

The deduce command is powerful enough to discharge these, proving the original VC:

```
deduce c#1 from [1,2,3].
done.
```

6 Hidden Details in Proof Segments

6.1 Interactive Yes and No Questions

In a number of the proof segments the short commands `y.` and `n.` have been inserted. These answer questions asked from the SPADE interface. In all cases, these questions are fully predictable and appropriate answers can always be generated. However, the need for these illustrates how even the slightest change to SPADE may require changes to the way commands are automatically generated.

6.2 Subterm Selection

When rewriting a subterm, if more than one occurrence of the subterm exists within a formula the user is asked which occurrence to replace. To automatically cater for this interaction NuSPADE will need to have a facility to determine where a term has duplicate subterms, and which one is to be replaced.

An alternative solution is to always replace with a single meta variable. This will force SPADE to list every possible subterm, which then be selected appropriately by NuSPADE. The advantage of this method is that all replace operations will occur in the same manner, irrespective if the subterm to be replaced occurs more than once or not.

6.3 Instantiating Meta Variables

In all of the rewrite plans there is a step where the logical formula justifying the rewrite is introduced. The formula introduced is based on a SPADE rule, transformed into a logical formula, with meta-variables instantiated. For example, suppose the following rule is available:

```
rule(5): A+1 may_be_replaced_by succ(A).
```

Then applying it to a subterm `x+1` will require introducing the formula `x+1=succ(x)` while subterm `(a+b)+1` will require introducing the formula `(a+b)+1=succ((a+b))`. Performing the instantiation is always predictable inside a NuSPADE proof plan. However, it will require additional effort to support.

7 Powerful SPADE Rules

A source of flexibility in using SPADE is the ability to introduce new rules. Such a rule was proposed as one method to perform the trivial proof that A=A. SPADE allows rules to be defined with conditions, using a small collection of Prolog predicates if desired. Rules defined in this manner have the potential to be quite powerful. However, an error in one of these rules could have a large impact on the operation on SPADE.

It is argued that adding such clever SPADE rules might actually be modifying SPADE, breaking the core principles. Thus there is a reluctance in adding rules to SPADE solely for the purposes of integrating with NuSPADE.

8 Conclusion

Originally it was thought that X may_be_replaced_by Y defined a rewrite rule that was strictly applicable in the one direction. Thus the formula required to prove this rule would be of the form $Y \to X$ and the rule would only be applicable to conclusions. Only by better appreciating the subtleties of rewriting and the treatment of these rules in SPADE did the misconception present itself.

This note has attempted to provide a sound logical basis for all of the SPADE rules, to uncover any remaining misconceptions and provide a formal grounding from which to build NuSPADE. It has also been shown that the artificial limits that appear to be set on some rules in SPADE can be overcome with suitable tricks. Implementing these tricks is another consideration in extracting executable SPADE command logs from NuSPADE proof plans.

In the process of identifying methods for implementing rewriting the idea of a restrictive mode for SPADE was proposed. Although implementing this will require changes to SPADE, there is not a viable method to incorporate these changes inside NuSPADE. It is felt that a few hours working on SPADE to implement this will save several weeks implementing an alternative solution inside NuSPADE.

References

[1] Praxis Critical Systems. *SPADE Proof Checker Version 1.3.* 1998.