Chapter 4: FELIX: A Theorem Prover for Classical and Infon Logics

The logic of situation theory presented in the preceding chapters provideS a computational approach to knowledge representation. This chapter develops a theorem prover, named FELIX, which is an interest-driven, suppositional reasoning system which works with this representation. This theorem prover reasons with both classical and infon predicate calculi. Extensions are provided in Chapter 6 which enable it to prove theorems involving perceptual statements and nested beliefs held by multiple agents (a nested belief is a belief about a belief). This chapter proves the second hypothesis of the thesis:

> *Second Hypothesis:* There is a semi-decision procedure for this new infon logic, and a theorem prover can be devised which implements it. Further, many of the techniques of automated theorem proving developed for classical logic can be applied to automated theorem proving in this new infon logic.

The approach implemented in FELIX is an extension of the ideas used in Pollock's OSCAR.[1] Formally, it is a natural deduction-based theorem prover. Procedurally, the system implements "interest-driven suppositional reasoning", to use Pollock's terms. It uses both forward and backward chaining in its reasoning.

Why Use Natural Deduction?

There are many ways to build theorem provers. These different ways of building theorem provers can be characterized by the method of formalization of logic on which they are based. The major approaches to formalizations of logic are clausal form, sequent calculus, natural deduction, and axiomatic. Of these approaches, the first one does not apply to infon logic (as characterized by NH) because wffs in NH do not necessarily have an equivalent "clausal form" (a representation using only negation, disjunction, and conjunction). This means that a very powerful automated proof

[1] [Pollock 1990]. The two systems are only conceptually related. FELIX is an entirely novel implementation in Prolog by the author. OSCAR was implemented in LISP by Pollock.

technique, resolution, is not applicable to infon logic.

According to Dummett[2], the sequent calculus provides a decision procedure for FOL by virtue of the "cut-elimination" theorem and the fact that all wffs of FOL can be expressed in a "prenex" normal form. This is a form where all of the quantifiers are at the beginning of the wff - no quantified formula is used as an argument to a logical operator. But, wffs in infon logic cannot always be transformed into prenex form. As Dummett points out (with respect to intuitionistic logic), this is due to the failure of the converses of the following laws:

$$\forall x\, A(x) \vee B \quad \vdash \forall x\, (A(x) \vee B)$$
$$\exists x\, (B \Rightarrow A(x)) \vdash B \Rightarrow \exists x\, A(x)$$
$$\exists x\, (A(x) \Rightarrow B) \vdash \forall x\, A(x) \Rightarrow B$$

The axiomatic approach is awkward to use in constructing proofs, although it was useful in the development of infon logic in the previous chapter.

The natural deduction system approach is adopted here for building a theorem prover. This is presented in the rest of this chapter. It is readily adaptable to NN and it tends to produce readable concise proofs (which is not true of resolution and sequent calculus based theorem provers). These proofs may prove more amenable to automated analysis (e.g., Pollock's approach to default reasoning).

The Poker Game

An example problem described below has been adopted in this work as a benchmark of a minimal ability to deal with multiple agents, perception and belief. This example is from Allan Gibbard[3], and is discussed at length by Barwise[4] and Stalnaker[5]:

> Sly Pete and Mr. Stone are playing poker on a Mississippi riverboat. It is now up to

[2] p. 150 of Dummett.

[3] Originally from p. 231 of [Gibbard 1981] and discussed on pp. 231-234. This description is as given on p. 112 of [Barwise 1986]. Barwise states that he is using the version as given on pp. 108-109 of [Stalnaker 1984].

[4] pp. 112-113 and pp. 131-132 in [Barwise 1986].

[5] pp. 108-110 in [Stalnaker 1984].

> Pete to call or fold. My henchman Zack sees Stone's hand, which is quite good, and signals its contents to Pete. My henchman Jack sees both hands, and sees that Pete's hand is rather low, so that Stone's the winning hand. At this point the room is cleared. A few minutes later Zack slips me a note which says "if Pete called, he won," and Jack slips me a note which says "if Pete called, he lost..." I conclude that Pete folded.

This example is introduced by Gibbard to demonstrate that conditional statements (e.g. "if Pete called, he won") do not have any "propositional content". Stalnaker and Barwise continue the discussion of propositional content. Stalnaker modifies Gibbard's position by saying that "open conditionals" (a kind of conditional which Jack and Zack's statements exemplify) do have a propositional content, but it is "highly context dependent". The context to which Stalnaker here refers is that of the speaker and listener. The propositional content which Barwise attributes to nearly *any* kind of sentence is "context dependent" - as interpreted in this thesis it is a claim about an infon being supported by a situation. This approach can be used to represent the conditionals of the example.

The poker game example is formalized as two theorem proving problems. One is to show that Jack's statement is reasonable and the other is to show that Zack's statement is reasonable. To do this, a few general principles about poker need to be added to the facts of the example. The formalization of this example is discussed in the chapter on belief and how these two theorems are proved by FELIX is discussed in the chapter on multiple-context FELIX. The basic operation of FELIX is described in this chapter.

FELIX

In developing OSCAR, Pollock was interested in having a theorem prover which reasoned in a fashion similar to that of people. The similarity is not strong, but it is much stronger than with most theorem provers, particularly resolution-based theorem provers. The structure of OSCAR lent itself to extension to other logics, since many of the inference rules of a natural deduction system for classical predicate logic have a direct representation in OSCAR. FELIX is a re-implementation of

OSCAR that inherits these features. Extending FELIX to support infon logic (NN, the natural deduction form of NH) is primarily a matter of adjusting these directly represented natural deduction system inference rules.

There are two aspects of FELIX - the theorem proving framework and the definition of the logic used in theorem proving. It is the framework which defines how FELIX organizes the search for a proof.

FELIX's theorem proving framework

A problem is posed to FELIX as a set of formulae which are the premises and the formula which is the conclusion to be proved from the given premises. FELIX is also told whether to use the classical or infon logic systems. Since the deduction theorem holds in both logic systems, it is equivalent to state a problem as "given P, prove Q" or "given nothing, prove P⇒Q".

The major data structures of FELIX are shown in Exhibit 4. 1 on page 79. The largest structure is the *supposition.* This contains several items, including the *agenda* and the *contexts.* The agenda contains a list of what *tasks* are currently "pending" for the supposition. An intensional context contains the set of formulae that are currently of *interest* in that intensional context and the set of formulae that have already been *adopted* in that intensional context. The *pattern* indicates how formulae of the intensional context are expanded to equivalent formulae in that intensional context's parent intensional context.

Adoptions and Interests

In searching for a proof, FELIX has interests and adoptions. An *interest* is a formula which FELIX is attempting to prove. Usually, proving an interest leads more or less directly to proving the theorem. An *adoption* is a formula which has been proven. Processing an interest P can generate one or more new interest tasks Q, where Q and the adoptions Γ held at the time of processing interest P together prove P (Γ, Q |- P). This generation of interest tasks is part of the implementation of backward chaining

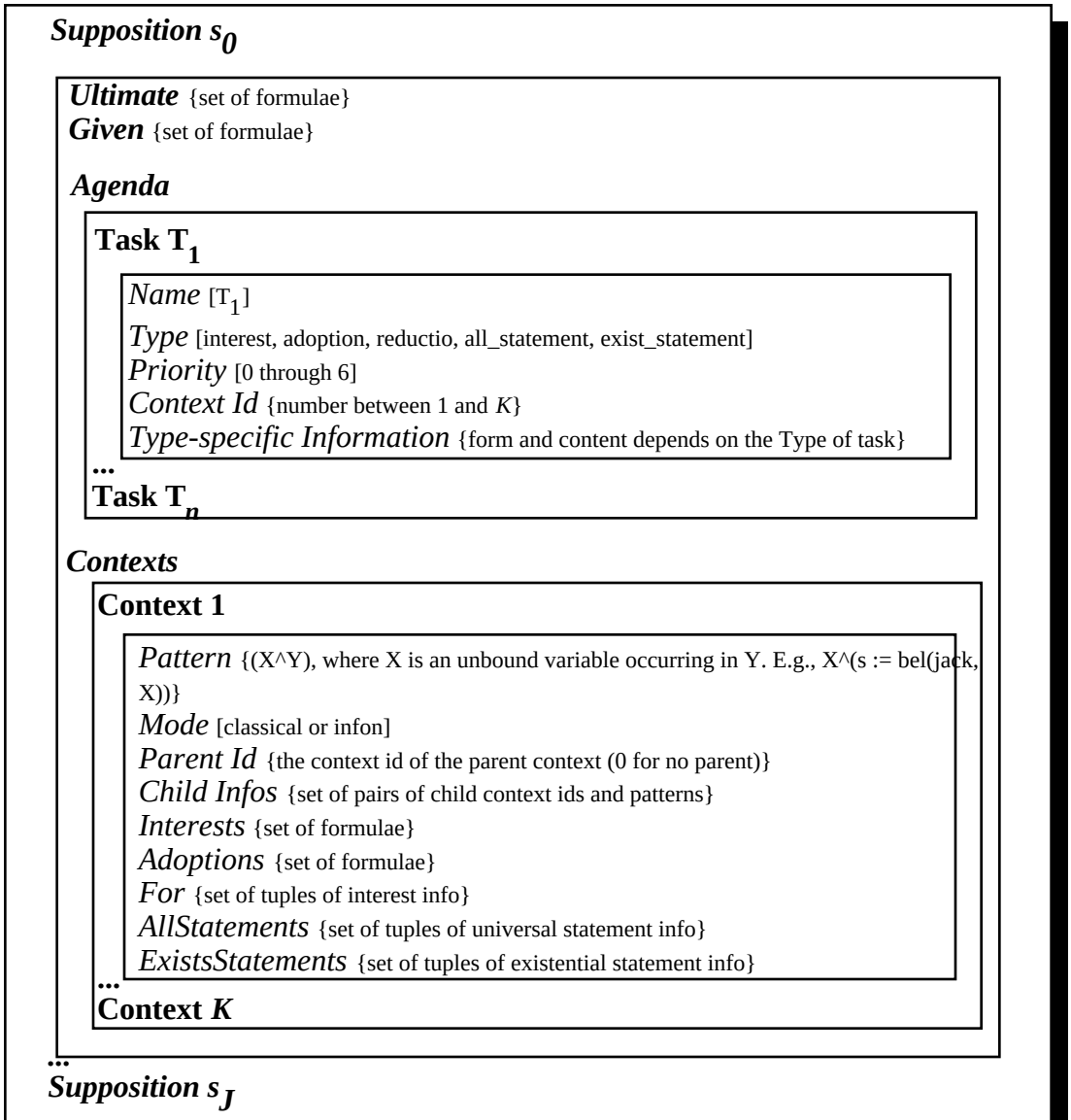

Exhibit 4. 1: FELIX Data Structures

reasoning in FELIX. Processing an adoption P can generate one or more new adoption tasks Q, where Q follows from P and the adoptions Γ held at the time of processing adoption P (Γ, Q |- P). This generation of new adoption tasks is the implementation of forward chaining reasoning\in FELIX.

The Task Agenda

FELIX has an ordered list of *tasks*, its agenda. It generally operates by repeatedly executing the first task on the agenda. The execution of a task may add new tasks to the agenda, and it may completely reorder the agenda. When a task is selected to be executed it is removed from the agenda (no task is executed more than once). FELIX stops when it adopts the theorem, or when it runs out of tasks. Adopting the theorem is a successful conclusion, running out of tasks is a failure. FELIX is a semi-decision procedure, so if FELIX is given a false theorem to prove, it may never halt.

Suppositions

FELIX defines a reasoning problem as a *supposition.* The input to a supposition consists primarily of some ultimate interests and some "given" formulae (those which are supposed to be true). Thus, the primary theorem proving task is expressed to FELIX as a supposition where the conclusion of the theorem is the ultimate interest and the premises of the theorem are the "givens".

Under certain circumstances FELIX creates a new supposition, invokes itself on this new supposition, then returns to the original supposition. This is essentially proving a lemma to support a step of a proof, where the lemma is the new supposition. There are four supposition-creating rules: conditional supposition, *reductio* supposition, universal supposition, and dilemma supposition. These supposition rules are part of the implementation of backward chaining reasoning in FELIX. A supposition contains all of the data FELIX needs to keep track of its work. Thus part of the structure of a supposition is an agenda for that supposition, a set of processed adoptions, a set of processed interests, the ultimate interests, and the given formulae. Handling the agenda for a particular supposition is called *linear processing*.

Logic Modes

FELIX has two logic modes in which it may operate: *classical* and *infon*. The setting of the mode controls the generation of interest and adoption tasks such that the new tasks are valid in that mode. Also, the supposition rules are adapted to the current mode.

Intensional Contexts

An additional concept used to organize suppositions is *intensional context*. Intensional contexts are used in reasoning about situations and beliefs (and can be applied to any of the propositional attitudes). The *root* intensional context is that intensional context in which a supposition starts. An intensional context has a defining pattern, a parent intensional context, any number of child intensional contexts, a set of processed adoptions, and a set of processed interests. The task agenda is *not* per intensional context, but per supposition. This allows FELIX to move easily between in-

tensional contexts. The formulae in an intensional context are interpreted in that intensional context's parent intensional context via the defining pattern of the intensional context. For instance, let 'bel(terry, P)' mean that "terry believes P" where P is some proposition. Then there can be an intensional context with a defining pattern of 'X ^ bel(terry, X)'. The proposition '1 < 2' in this intensional context is interpreted in the parent intensional context as 'bel(terry, 1 < 2)'. Intensional contexts are discussed in more detail later.

The FELIX Algorithm

The Problem Space

FELIX interleaves two different kinds of searches - a "forward" tree search from the givens toward the theorem conclusion and a "backward" AND/OR search from the theorem conclusion toward the givens. The invocation of suppositional reasoning suspends work in the parent supposition and starts work in the child supposition. The parent supposition will resume when the child is finished.

The FELIX composite problem space can be defined as follows. A state of a FELIX problem is described by those formulae which have been adopted, those formulae in which interest has been registered, the processed exists statements and processed universal statements which are available to be used (these are special elements of the registered interest formulae and adopted formulae, respectively), and the "for" set (which connects the interest formulae into an AND/OR graph by recording what FELIX is interested in a particular formula *for*[6]). The operations which transform one state into another are the tasks. The problem formulation used by FELIX is a "monotonic" system in the sense that if a task is appropriate for a particular state, then it can be applied to any subsequent state (i.e., tasks can be done in any order, one can always "adopt P" or "register interest in Q", although the effect may be to make no change if P is already adopted or interest is already registered for Q). The FELIX system is "partially commutative" in that a set of tasks applied to a state will

[6] The name "for" set was used by Pollock in describing and implementing OSCAR. Perhaps "interest reasons" might be a better name.

produce the same final state for any valid order of the tasks. Since FELIX is "monotonic" and "partially commutative", it is "commutative".

The nodes of the FELIX problem space are the states described above. A FELIX problem consists of an initial state which is some set of given adoptions and an initial interest, and a goal state which is any state which includes all of the given adoptions plus the initial interest formula as an adopted formula. The path from the initial state to a goal state contains sufficient information to construct a proof of the given theorem. This path is described by the sequence of tasks processed to make each of the state transitions. A reduced version of this path, called the "basis", is actually all that is kept. It consists of only the adoption tasks which are on the solution path. The proof of the theorem is extracted from the basis by chaining backward from the step which adopted the consequence of the theorem through the justification steps. Thus, the the proof tasks may be a small subset of the basis, which may in turn be a small subset of the solution path.

The search algorithm for FELIX is agenda-driven. It does not keep a list of "closed" nodes. All of the tasks which FELIX uses only *grow* the previous state to produce the next state. Thus, FELIX can never return to an already visited state. Further, within a supposition FELIX never backs up and it only pursues a single path (in a depth-first fashion). Thus, FELIX cannot encounter the same state on two different paths.[7] Therefore, since FELIX cannot encounter the same state twice in a search, it does not keep track of what states it has visited (closed nodes).

The "open" nodes of the FELIX search are recorded indirectly via the tasks which can be used to generate new states. Since the state which a task generates depends on the state to which it is applied, the "open" tasks do *not* specify an invariant set of "open" states (or nodes). A state of the search algorithm (as opposed to a state of the problem) is a problem state plus the task agenda plus some other information. For any state of the search, the task agenda identifies a set of possible next states. The same task agenda given a different problem state specifies a different set of next

[7] FELIX *can* encounter the same supposition from two different states (in a "superior" supposition or in two different "superior" suppositions). For this reason, some attempt is made to reuse suppositions.

states, however.

A task not only generates a new problem state, but it also may create any number of new tasks which are added to the agenda.

The interests (and the “for” set) in a problem state can be viewed as representing an AND/OR tree, with the initial (ultimate) interest the top or root of the AND/OR tree (the initial problem), and the interests generated from that as subproblems (which, if “solved” then solve the root), each of which may have subproblems, and so on. The "for" set keeps track of the arcs between the interest formulae, and whether the arcs are AND arcs or OR arcs. An interest “subproblem” is solved when the interest formula is found in the current set of adoption formulae for the problem.

These two searches are interleaved by having their tasks commingle on a single agenda. The ordering of these tasks decides not only which step is “best” to take next in each of the searches, but also which of the searches (forward or AND/OR) to advance next. Processing a particular task may generate tasks for *both* kinds of searches, e.g. an adoption task may generate some adoption tasks and it may also generate some interest tasks. Thus, advancing the forward adoption search may change the “open” tasks for the backward interest AND/OR search. The converse is also possible.

Basic Activities

There are several kinds of “steps” which FELIX takes in trying to find a proof: register interest in proving a formula, identify formulae for which interest can be registered, adopt a formula as having been proved, identify formulae to be adopted, and create/process a supposition to prove a formula (which is subsequently identified to be adopted). The difference between registering interest in a formula and identifying a formula for which interest can be registered is that registering interest “triggers” a variety of activities which may identify formulae for which interest can be registered or identify formulae to be adopted, but identifying a formula for which interest can be registered simply notes the existence of that formula so that FELIX can process

that formula later if it deems it appropriate. There is a similar distinction between adopting a formula and identifying a formula to be adopted. Thus identifying a formula has no *immediate* repercussions, but registering interest or adopting can have many immediate repercussions.

There are four situations in which a supposition is created and processed; to prove a conditional of the form $P \Rightarrow Q$ (conditional supposition), to prove a universal statement of the form '$\forall$ **x** P(**x**)' (universal supposition), to prove a formula P by *reductio ad absurdam* reasoning (*reductio* supposition), and to prove an ultimate interest in formula P from an adopted disjunction ($Q \vee R$) by reasoning by cases (or "dilemma" reasoning: dilemma supposition). Conditional supposition reasoning is heavily used by FELIX. It is one of the three key operations in FELIX's success at finding proofs with a relatively small number of useless searches. The other two key operations are "all detachment" processing and "exists detachment" processing. These last two operations are discussed later when the predicate calculus operation of FELIX is presented. Dilemma supposition reasoning is built on conditional supposition reasoning. *reductio* supposition reasoning introduces a resolution-theorem-prover-style hunt for contradictions.

Conditional supposition reasoning for ($P \Rightarrow Q$) works by creating a supposition which is an extension of the current supposition where P is "supposed" - added to the set of supposed formulae - and Q is made the sole ultimate interest of this new supposition. If Γ is the set of supposed formulae for the current supposition, then conditional supposition reasoning is attempting to prove that Γ and P derives Q. By the deduction theorem, this implies that Γ derives $P \Rightarrow Q$.

Universal supposition reasoning for ($\forall$**x**P(**x**)) works by creating a supposition which is a copy of the current supposition where P* is made the ultimate interest of this new supposition. P* is created from P by replacing all occurrences of **x** in P(**x**) with an otherwise unused free variable. If Γ is the set of supposed formulae for the current supposition, then universal supposition reasoning is attempting to prove that Γ derives P*. By universal generalization, this implies that Γ derives ($\forall$**x**P(**x**)).

Reductio supposition reasoning for P works by creating a supposition which is an extension of the current supposition where –P is "supposed" - added to the set of supposed formulae - and P is made the sole ultimate interest of this new supposition. If Γ is the set of supposed formulae for the current supposition, then *reductio* supposition reasoning is attempting to prove that Γ and –P derives P. By *reductio ad absurdam* inference rule of classical logic, this implies that Γ derives P. This is adapted to infon logic by using weak negation ('^P' means "P ⇒ ***false***" or "P is not supported"), where Γ and P derives ^P implies that Γ derives ^P.

There is an indirect *reductio* reasoning rule where one supposes –P and derives Q and –Q, then one can infer P. This is adapted to infon logic by using weak negation instead of strong negation: one can infer ^P by supposing P and deriving Q and ^Q. The target of the inference must be weakly negated for either infonic *reductio* inference - *reductio* inference can not be use in infon logic to derive a positive or strongly negative formula.

Dilemma supposition reasoning for proving P given (Q ∨ R) works by creating two suppositions A1 and A2 which are a extensions of the current supposition, where Q is "supposed" in A1 and R is supposed in A2 and P is made the sole ultimate interest of both new suppositions. If A1 works, then A2 is attempted. If Γ is the set of supposed formulae for the current supposition, then dilemma supposition reasoning is attempting to prove that Γ and Q derives P, and that Γ and R also derive P. By the "dilemma" theorem '(A⇒C) ⇒ ((B⇒C) ⇒ (A∨B ⇒ C))'[8], this implies that Γ derives P.

A special kind of dilemma reasoning is indirect supposition. This is used when there is an adopted formula of the form 'P -> Q', with Q some compound formula and neither P nor Q has been adopted. FELIX becomes interested in proving 'Q ->R' and 'P', where R is any ultimate interest. This reasoning method is needed in infon logic mode to prove some things which can be proved via *reductio ad absurdam* reasoning in classical mode. In classical mode, this reasoning method produces more perspicuous proofs than those produced via *reductio ad absurdam* reasoning.

[8] This is an axiom in many formalizations of classical logic, and it is an axiom of NH.

The non-intensional, non-quantificational algorithm

The top-level of FELIX operates on suppositions - FELIX starts by invoking linear-processing on the initial supposition for the theorem to be proved. Linear-processing returns to the top-level with a success indicator, a failure indicator, or a “change to supposition X” indicator. In the first two cases, the top-level is done proving the theorem. In the last case, linear-processing is invoked on the indicated supposition (X).

All of the input and results of linear processing are restricted to the supposition which is being processed with the exception that linear processing may create new suppositions, or extend the set of interests of an existing supposition. Linear processing consists of:

1) taking the first task from the agenda of the current supposition,

2) switching the current intensional context to the intensional context given in the task,

3) processing the task type-specific information according to the task type

and repeating the above steps

until: either all of the formulae which were initially ultimate interests of the current supposition have been adopted, there are no more tasks in the current supposition, or the result of processing the current task was a directive to change suppositions. Exhausting the set of ultimate interests returns a “success” indicator from linear processing. Exhausting the tasks in the agenda returns a “failure” indicator from linear processing. Encountering a change supposition directive returns a change supposition indicator from linear processing.

There are several kinds of tasks, and the processing of each kind is complex. This is the heart of the description of the algorithm. The ordering of the agenda is crucial to limiting the search for a proof in that it determines the order in which tasks are done. This is discussed after the tasks have been introduced.

Task Processing

A task has several components: the task type, the type-specific information, the task priority, and the task intensional context. The type, type-specific information, and intensional context are used in processing the task. The priority and intensional context are used in ordering the tasks in the agenda. Tasks with lower priority numbers are processed before tasks with higher ones. The intensional context is used to order the processing of tasks which have the same priority. Intensional contexts are discussed in Chapter 6.

The kinds of tasks are: *interest*, *adoption*, *reductio*, *exists_statement*, and *all_statement*. The major kinds of tasks are *interest* and *adoption*. The *exists_statement* and *reductio* kinds are special cases of *interest* processing. The *all_statement* kind is a special case of *adoption* processing.

The details of the existential, universal, and *reductio* processing algorithms are discussed later. The invocations of these algorithms are shown where appropriate in the task processing algorithms given below.

Interest Task Processing

The type-specific information of an interest task is a formula 'P' in which interest is to be adopted, a formula Q which is the formula which led to this interest task being created, the name of the supposition in which Q is of interest, a set of all of the formulae (and their associated supposition names) which must be adopted to completely support the derivation of Q (one of which is P), the subset of the support formulae which were unadopted (beside P) when the interest task was created, and the 'reason' for the interest in P (what rule was used in creating the interest).

Processing interest is:

If P-in-support-of-Q is already in the 'for' set, then do nothing.
Else do:

Add P to the interests set.
Add type-specific information to the ‘for’ set as a single tuple.
If P is of the form (Q ⇒ R), then invoke conditional supposition processing.
Else if P is of the form (∀**x**Q(**x**)), then invoke universal supposition
processing.
Otherwise;
If P is of the form (exists X (Q)), then invoke existential interest processing.
If existential interest processing determines that P can be adopted,
then do no more.
Otherwise;
Find supporting formulae for P via backward chaining reasons
and current adoptions
and add these as interest tasks to the agenda.
Invoke *reductio* interest processing.

Processing conditional supposition for (Q ⇒ R) is:

Create a new supposition which has as its supposed formulae the supposed formulae of the current supposition and Q. Make R the ultimate interest of the new supposition. Add a ‘for’ set tuple with (Q ⇒ R) as the target formula, the current supposition as the target supposition, R as the ‘deriving’ formula and the only support formula. Linear process the new supposition.

Processing *reductio* interest in P is:

If NotP is adopted, or there is an adoption task for NotP, or there is a ‘for’ *reductio* interest in P, or there is a *reductio* interest task for P,
then do nothing.
Otherwise:
add a *reductio* task for P.

Adoption Task Processing

The type-specific information of an adoption task is a formula ‘P’ which is to be adopted, the ‘reason’ for the adoption (which may be inherited from an interest-generating activity), and a set of pairs of formulae and supposition names which are the formulae which support the reason for the adoption of P.

Processing adoption is :

If P is already in the set of adopted formulae, then just discharge interest in P.
Else do:

Add P to the set of adopted formulae.
Discharge interest in P.
Cancel interest in P.
Cancel interest in ‘not P’.
Find concludable formulae using the forward chaining rules,
current adoptions, and P,
add adoption tasks for them.
If P is of the form ($Q \vee R$), then invoke dilemma supposition processing.
Invoke indirect *reductio* processing.
Invoke all detachment processing.
Invoke universal statement update processing.
Invoke existential statement update processing.

Discharging interest in P is:

If there is a tuple in the ‘for’ set with P as the deriving formula and some Q as the target formula,
then do:

If there are any formulae in the unadopted support in the tuple,
then do:
Let R be any one of these unadopted formula.

Add an interest task for R, with the same target formula Q.
add a new tuple to the ‘for’ set which has R as the deriving formula and
Q as the target.
end.
else add an adoption task for Q.
end.
else do nothing.

Processing dilemma supposition for (Q ∨ R) is:

For every ultimate interest U:
add a dilemma interest task with;
Q⇒U as deriving formula,
R⇒U as unadopted support,
{Q⇒U, R⇒U} as total support,
U as target formula,
current supposition as target supposition.

Processing indirect *reductio* reasoning for P is:

If the current supposition is a *reductio* supposition
then do:
For every suitable ultimate interest U:
add a *reductio*2 interest task with;
(*reductio* negation of P) as deriving formula,
empty unadopted support,
[P, (*reductio* negation of P)] as total support of U
U as target formula,
current supposition as target supposition.
Otherwise, do nothing.

(The “suitable ultimate interest U” is tailored to the logic mode - any U for classical logic, only U of the form ‘^V’ for infon logic.)

Reductio task processing

The type-specific information is the proposition ‘P’ and its negation ‘not P’, where P is to be proved by supposing ‘not P’ and finding a proof of ‘P’. Strong negation is used for classical logic and weak negation is used in infon logic.

Processing *reductio* is:

Assign the first priority 5 exists statement or all statement task a priority of 4 and move it the appropriate place in the agenda order. (This is part of the arrangement for *reductio*, exists statement, and all statement tasks to alternate, regardless of what order they were added to the agenda.)

If NotP has already been adopted, or there is already a ‘for’ set *reductio* item for P,
or there is a *reductio* interest task pending,
then do nothing.
Otherwise;
Make a *reductio* supposition which supposes the current given formulae plus ‘not P’, and which has a ‘for’ set item which designates ‘P’ in the current supposition as the target with ‘not P’ in the *reductio* supposition as the only supporting formula.
Invoke linear process on the *reductio* supposition.

This task relies on the *reductio* supposition having been constructed such that if P is proved in the *reductio* supposition, then discharging interest in P during the adoption of P in the *reductio* supposition will lead to adding an ‘immediate’ adoption task of P to the parent supposition and a ‘change supposition to parent’ result. This causes an ‘exit’ from the *reductio* supposition by linear processing and the top-level will switch to the parent supposition, which then shortly executes the P adoption task.

Exists statement task processing

The type specific information for this kind of task is simply the existential formula ‘P’ which is to be processed.

Processing exists statement is:

Assign the first priority 5 all statement or *reductio* task a priority of 4 and move it the appropriate place in the agenda order.
Remove the existential statement P from the ExistentialStatements set.
Create an instantiation of P for each term known,
and add an interest task to the agenda for each such instantiation. (Proving one of these instantiations proves P).

This task is a backup for the exists detachment inference rule. Generally, ‘exists detachment’ makes the desired inference without increasing the number of tasks, interests, or adoptions. When FELIX has run out of higher priority tasks (such as adoption and interest tasks), however, then it will try to prove the existential interest “exhaustively” by generating interest in all possible instantiations of the existential interest formula. If there are many terms and the existential formula has several existential variables, this can be exceedingly expensive.

All statement task processing

The type specific information for this kind of task is simply the universal formula ‘P’ which is to be processed.

Processing all statement is:

Assign the first priority 5 *reductio* or exists statement task a priority of 4 and move it the appropriate place in the agenda order.
Remove the universal statement P from the UniversalStatements set.
Create an instantiation of P for each term known,
and add an adoption task to the agenda for each such instantiation.

This task is a backup for the ‘all detachment’ inference rule. Generally, ‘all detachment’ makes the desired inference without increasing the number of tasks, interests,

or adoptions. When FELIX has run out of higher priority tasks (such as adoption and interest tasks), however, then it will “forward chain” from the universal interest “exhaustively” by generating adoptions for all possible instantiations of the universal formula. If there are many terms and the universal formula has several universal variables, this can be exceedingly expensive.

Execution of FELIX

Notation

The logical notation used in FELIX differs from that found elsewhere in this thesis. The difference is due to the limits of the character set available within the implementation of FELIX. The translation is:

FELIX		Thesis
p/\q	=	p∧q
p\/q	=	p∨q
~p	=	¬p
p->q	=	p⇒q
p==q	=	p⇔q
∏ x? : p(x?)	=	∀x(p(x))
∑ x? : p(x?)	=	∃x(p(x))
p(f0*)/\q(f1*)	=	p(x)∧q(y)

Free variables are represented in FELIX by a structure of the form ‘fN*’, where N is an integer. Bound variables are represented in FELIX by a structure of the form ‘X?’, where X is any atom. No two different uses of quantifiers should have the same bound variable (all bound variable names should be unique). Existential instantiation creates new terms of the form ‘XN@’, where X is the original bound variable name (preceding the ‘?’), and N is an integer which guarantees that the name is unique.

Interpreting a FELIX Proof

The output from FELIX includes both a trace of FELIX’s reasoning process and a final summary of the proof steps which FELIX found to achieve the ultimate interest.

The first example proof is given in Exhibit 4. 2 on page 96. The exhibit contains only the proof portion of the output from FELIX. This is the proof from FELIX for ‘test 5’. The theorem which FELIX is proving in this example (‘test 5’) is: if one is

PROVE: p∧q

GIVEN: p∨q
p⇒q
q⇒p

Step :	*Adopted Formula*	*Support Steps*	*Justification*
1 :	p∨q	given	input
2 :	p⇒p∧q	LEMMA s1	conditional B
3 :	q⇒p∧q	LEMMA s2	conditional B
4 :	p∧q	[1, 2, 3]	dilemma B

LEMMA s1

PROVE: p∧q
SUPPOSE: p

Step :	*Adopted Formula*	*Support Steps*	*Justification*
1 :	p	given	supposed
2 :	p⇒q	given	input
3 :	q	[2, 1]	modus_ponens F
4 :	p∧q	[1, 3]	conjunction B

LEMMA s2

PROVE: p∧q
SUPPOSE: q

Step :	*Adopted Formula*	*Support Steps*	*Justification*
1 :	q	given	supposed
2 :	q⇒p	given	input
3 :	p	[2, 1]	modus_ponens F
4 :	p∧q	[3, 1]	conjunction B

Exhibit 4. 2: Propositional Example (test 5)

given that $p \vee q$, $p \Rightarrow q$, and $q \Rightarrow p$ all hold, then one can infer in infon logic that $p \wedge q$[9]. This is written more compactly as: $\{p \vee q, p \Rightarrow q, q \Rightarrow p\} \vdash_{NH} p \wedge q$. If the NH is not present on the '|-' operator, then the inference is done in classical logic. This example is done for infon logic (NH), but the same proof is found by FELIX in classical mode.

The proof steps (adoptions) made in the initial supposition, 's0', are given first. If any additional supposition is used in constructing the proof, it is given as a *lemma*. All of the proof steps in a particular supposition are displayed together, even though FELIX may have switched between suppositions in the course of finding the proof.

Suppositions are named 'sN' where N is an integer greater than or equal to 0. Therefore, the lemmas have names of this form (since each lemma is the display of a supposition).

A proof step is of the form 'L: P S J', where L is the number of the step, P is the proposition which this step asserts ("adopts"), S is the support for the step (usually a list of 1 or more step numbers), and J is the justification for why S supports the adoption of P. There is a "flag" character following the justification of 'B' or 'F'. 'B' indicates that the step was generated by backward chaining. 'F' indicates forward chaining. Justifications and their interpretation are given in the following table:

Type of explanation	*Support*	*Interpretation*
input	given	P is from the input to FELIX.
conditional B	LEMMA *n*	P is true via "conditional supposition", based on the proof of LEMMA *n*. This is a backward chaining justification.
dilemma B	[D, C1, C2]	P is true via "dilemma supposition", since step C1 is of the form $A \Rightarrow P$, step C2 is of the form $B \Rightarrow P$, and D is of the form $A \vee B$. Steps D, C1, C2 are all in the same

[9] This theorem also holds in classical logic.

		supposition as the current step.
reductio B	LEMMA *n*	P is true via "*reductio* supposition", where the negation of P was "supposed" in LEMMA *n* and P was derived in that lemma.
universal_generalization	LEMMA *n*	P is an "universal generalization" of the free variable formula derived in LEMMA *n*, which has the same supposed formulae as the current supposition.
universal_instantiation	[A]	P is an "universal instantiation" of the universal formula of step A.
all_detachment	[A, B1,…,Bn]	P is true by "all detachment" of the proposition of step A against the propositions of steps B1 through Bn. The universal proposition of step A requires that the dual of at least one of –P, and the propositions of steps B1 through Bn is true. Thus, since propositions of steps B1 through Bn *are* true, non of their duals can be true. Thus, P must hold.
existential_generalization	[A]	P is an "existential generalization" of the formula of step A.
existential_instantiation	[A]	P is an "existential instantiation" of the existential formula of step A.
Forward justifications:		
modus_ponens	[A1, A2]	One of the support steps is Q⇒P and the other is Q, deriving P.
Backward justifications:		
conjunction	[A1, A2]	The formulae for the support steps are P and Q, deriving P∧Q.

The interpretation of the proof in Exhibit 4. 2 on page 96 is as follows:

The goal is to prove ‘p∧q’ given ‘p∨q’, ‘p⇒q’, and ‘q⇏p’. Step 1 is in supposition s0 and simply asserts a given formula - ‘p∨q’. Step 2 asserts a conditional, ‘p⇒q∧p’, which is proved in LEMMA s1. Step 3 asserts a conditional, ‘q⇒q∧p’, which is proved in LEMMA s2. Step 4 completes the proof by asserting ‘p∧q’, using “dilemma” reasoning based on steps 1 to 3.

In LEMMA s1: Step 1 and 2 assert formulae which the lemma “supposes”. The formula of step 1 is the formula which this lemma supposes which is *not* a given for the theorem (supposition s0). The formula of step 2 is one of the givens of the theorem. Step 3 is the ‘modus ponens’ derivation from steps 1 and 2. Step 4 is a ‘conjunction’ derivation from steps 1 and 3, and it completes the lemma’s proof.

The structure of LEMMA s2 is very similar to that of LEMMA s1.

Interpreting a FELIX Trace

The example in Exhibit 4. 2 on page 96 does not indicate *how* FELIX found the proof, it only gives the proof which FELIX did ultimately find. The full output from FELIX includes both processing information *and* the final proof. This processing information is the "trace". There is a long form of this trace and a brief form. The brief form is discussed here. The trace for the derivation of the proof given in Exhibit 4. 2 on page 96 is given in three parts: Exhibit 4. 3 on page 101, Exhibit 4. 4 on page 102, and Exhibit 4. 5 on page 103.

There are several different kinds of activities which are indicated by the lines of the trace:

"Add t*N* : *Type* - *Info*"

indicates the adding of task t*N* of type *Type* with type-specific information *Info.*

"Process interest: *Formula*"

indicates the processing of an interest task for formula *Formula*.

"Interest ramification: interest - *Info*"

indicates the registering of interest as specified in *Info*

"Process adoption (*N*): *Formula*"

indicates the processing of an adoption task for adopting formula *Formula* in intensional context *N*.

"REGISTER FOCUS: Context *N*, Proposition *P*"

indicates the registering of intensional context *N* as a focal intensional context, based on interest in the formula (proposition) *P*.

"REMOVE FOCUS: Context *N*, Proposition *P"*

indicates the removal of formula *P* as a basis for focus on intensional context *N* (if there is no other basis for focus on intensional context *N*, then intensional context *N* is no longer a focal intensional context).

This processing output is organized around suppositions. As long as FELIX is work-

```
*************** FELIX *****************
     Logic Mode: classical

     Test: 5

Add t0 : interest - p/\q
Add t1 : adoption - p\/q
Add t2 : adoption - p->q
Add t3 : adoption - q->p

--------------
Linear-process: Supposition s0
     Prove: [p/\q]
     Suppose: []
     Agenda:t1 t2 t3 t0

Process adoption ( 1 ) : p\/q
Add t4 : interest - p->p/\q
Process adoption ( 1 ) : p->q
Process adoption ( 1 ) : q->p
Process interest: p/\q
Interest ramification: interest - i(p, con(backward, conjunction), [sup(q,
s0)], [sup(p, s0), sup(q, s0)], p/\q, s0)
Add t5 : interest - p
Add t6 : reductio - (p/\q)-(~ (p/\q))
Process interest: p
Interest ramification: interest - i(q, con(backward, modus_ponens), [],
[sup(q, s0), sup((q->p), s0)], p, s0)
Add t7 : interest - q
Interest ramification: interest - i(~ q, con(backward,
disjunction_and_negation), [], [sup(~ q, s0), sup(p\/q, s0)], p, s0)
Add t8 : interest - ~ q
Add t9 : reductio - p-(~ p)
Process interest: q
Interest ramification: interest - i(p, con(backward, modus_ponens), [],
[sup(p, s0), sup((p->q), s0)], q, s0)
Add t10 : interest - p
Interest ramification: interest - i(~ p, con(backward,
disjunction_and_negation), [], [sup(~ p, s0), sup(p\/q, s0)], q, s0)
Add t11 : interest - ~ p
Add t12 : reductio - q-(~ q)
```

Exhibit 4. 3: Execution Trace for Simple Propositional Example. (test 5) Part 1 of 3

ing on a particular supposition, it is “linear processing”. Whenever the supposition FELIX is working on is changed, then a new “linear-process” header for the “new” supposition is displayed (the “new” supposition may actual be one on which FELIX has previously done some work). The linear process header gives the name of the current supposition, what that supposition must prove, what the supposed formulae in that supposition are, and what task identifiers are in the agenda.

```
Process interest: ~ q
Add t13 : reductio - ~ q-q
Process interest: p
Process interest: ~ p
Add t14 : reductio - ~ p-p
Process interest: p->p/\q
Add t15 : interest - p/\q
Add t16 : adoption - p
Add t17 : adoption - p\/q
Add t18 : adoption - p->q
Add t19 : adoption - q->p


--------------
Linear-process: Supposition s1
     Prove: [p/\q]
     Suppose: [p]
     Agenda:t16 t17 t18 t19 t15

Process adoption ( 1 ) : p
Process adoption ( 1 ) : p\/q
Process adoption ( 1 ) : p->q
Add t20 : adoption - q
Process adoption ( 1 ) : q->p
Process interest: p/\q
Interest ramification: interest - i(q, con(backward, conjunction), [], [sup(p,
s1), sup(q, s1)], p/\q, s1)
Add t21 : interest - q
Add t22 : reductio - (p/\q)-(~ (p/\q))
Process interest: q
Process adoption ( 1 ) : q
Add t23 : adoption - p/\q
Add t24 : adoption - p
Process adoption ( 1 ) : p/\q
Add t25 : adoption - p->p/\q
LINEAR PROCESS TERMINATION: change_supposition
```

Exhibit 4. 4: Execution Trace for Simple Propositional Example (test 5). Part 2 of 3.

'Process adoption (*N*) : *P*' indicates that FELIX is considering adopting *P*. Generally a formula *P* is considered for adoption because FELIX processed an adoption *task* for *P*, which invoked the adoption processing of *P*. It is also possible for the processing of some other kind of task to lead directly to considering a formula for adoption, side-stepping the agenda mechanism. If *P* has already been adopted, the current supposition is left unchanged. In part 2 of the example trace (see Exhibit 4. 4, page 102) near the end of the processing of supposition s1, FELIX adopts q ("Process adoption

```
--------------
Linear-process: Supposition s0
     Prove: [p/\q]
     Suppose: []
     Agenda:t25 t6 t9 t12 t13 t14

Process adoption ( 1 ) : p->p/\q
Add t26 : interest - q->p/\q
Add t27 : interest - p/\q->p/\q
Process interest: q->p/\q
Add t28 : interest - p/\q
Add t29 : adoption - q
Add t30 : adoption - p\/q
Add t31 : adoption - p->q
Add t32 : adoption - q->p

--------------
Linear-process: Supposition s2
     Prove: [p/\q]
     Suppose: [q]
     Agenda:t29 t30 t31 t32 t28

Process adoption ( 1 ) : q
Process adoption ( 1 ) : p\/q
Process adoption ( 1 ) : p->q
Process adoption ( 1 ) : q->p
Add t33 : adoption - p
Process interest: p/\q
Interest ramification: interest - i(p, con(backward, conjunction), [],
[sup(p, s2), sup(q, s2)], p/\q, s2)
Add t34 : interest - p
Add t35 : reductio - (p/\q)-(~ (p/\q))
Process interest: p
Process adoption ( 1 ) : p
Add t36 : adoption - p/\q
Add t37 : adoption - q
Process adoption ( 1 ) : p/\q
Add t38 : adoption - q->p/\q
LINEAR PROCESS TERMINATION: change_supposition

--------------
Linear-process: Supposition s0
     Prove: [p/\q]
     Suppose: []
     Agenda:t38 t27 t6 t9 t12 t13 t14

Process adoption ( 1 ) : q->p/\q
Add t39 : adoption - p/\q
Process adoption ( 1 ) : p/\q

LINEAR PROCESS TERMINATION: success_termination
```

Exhibit 4. 5: Execution Trace for Simple Propositional Example (test 5). Part 3 of 3.

(1) : q"). On adopting 'q', FELIX notices via an element in the 'for' set that adopting 'q' allows it to adopt 'p∧q'. Thus, FELIX adds an adoption task to the agenda and this appears in the trace as "Add t23 : adoption - p/\q". This enabling element in the 'for' set was placed there when interest was registered in 'p/\q'. The lines of the trace for this in Exhibit 4. 4, page 102, are:

Process interest: p/\q
Interest ramification: interest - i(q, con(backward, conjunction), [], [sup(p, s1), sup(q, s1)], p/\q, s1)

'Process interest: *P*' indicates that FELIX is considering registering interest in *P*. Depending on circumstances, interest in P may be registered (changing the interests and 'for' set), or *P* may be adopted directly (bypassing the agenda mechanism). Also, interest tasks for new formula may be added to the agenda. When processing interest in *P*, FELIX looks to see what backward chaining rules it has for proving *P*. These provide additional formulae for which interest tasks are added to the agenda. If there is a set of formulae provided by a backward chaining rule which supports *P* and all of the formulae in that set have already been adopted, then FELIX adopts *P* directly, rather than either simply registering interest in *P* or adding an adoption task for *P*.

Processing an interest task for a conditional formula causes FELIX to do "conditional supposition" processing in an immediate attempt to prove this conditional. In the example, this is the circumstance which causes the initial processing of supposition s0 to be suspended and supposition s1 to be entered. This occurs at the line of the trace near the beginning of Exhibit 4. 4, page 102, "Process interest: p->p/\q".

'Interest ramification: interest - i(Q, Type, U, T, P, S)' indicates that FELIX is registering interest in proving Q, with reason of type Type, with reason support of T (U is the list of formulae in T, other than Q, which have not yet been adopted), P is the formula which proving Q supports, and S is the supposition which is interested in P. This information is placed in the 'for' set; this describes *for* what FELIX is interested in Q. As an example, the information in the interest ramification step referred to above ("Interest ramification: interest - i(q, con(backward, conjunction), [], [sup(p,

s1), sup(q, s1)], p/\q, s1)") is interpreted as follows:

'q' is the formula of interest (the "deriving formula").

'con(backward, conjunction)' is the "reason" for the interest.

'[]' indicates there are no other unadopted formulae.

'[sup(p, s1), sup(q, s1)]' is a list of two "support" items, 'p' and 'q', both in supposition s1, which are the support for the target formula.

'p/\q' is the target formula.

s1 is the target supposition.

The reason of 'con(backward, conjunction)' is interpreted as - a direct *con*clusion using the *backward* chaining *conjunction* rule. This rule is that if one needs to prove 'p/\q', then prove each of 'p' and 'q' since 'p' and 'q' together derive 'p/\q': p, q |- p∧q.

Special Procedures for Quantification

There are several special procedures for quantification; universal supposition, universal instantiation, existential generalization, existential instantiation, all detachment, and exists detachment. Universal supposition has already been introduced. Universal instantiation, existential generalization, and existential instantiation are traditional inference rules. The last two special procedures are unique to FELIX and deserve special attention.

Universal Instantiation

Universal instantiation is an adoption procedure which can be invoked whenever a universal formula is adopted. FELIX keeps track of all of the terms (arguments to predicates) and instantiates the bound variable of the formula to each known term. This process clearly produces an enormous number of new formulae to be adopted, and very few of these formulae are generally useful in the final proof. The all detachment procedure is used to avoid universal instantiation under certain circumstances. Universal instantiation is only done after all other reasoning techniques except universal instantiation and *reductio* supposition have been tried. The all_statement task is a task to finish all-statement processing on a particular formula by invoking universal instantiation on that formula.

Existential Generalization

Existential generalization is an interest-registration procedure. It is analogous in its operation to the universal instantiation procedure discussed above. When an interest task is processed for an existential formula, then new interest tasks are generated for all possible instantiations of that formula using the known terms. This can create an enormous number of interest tasks if there are many terms or the existential formula has several variables, when at most one of these interests is needed to prove the original existential formula. The exists detachment procedure is used to delay existential generalization as long as possible. Existential generalization is only done after all other techniques except universal instantiation and *reductio* supposition have been

tried. The exists_statement task is a task to finish exists-statement processing on a particular formula by invoking existential generalization on that formula.

Universal Supposition (Generalization)

Universal supposition is used to implement universal generalization. Given interest in a universal formula *P*, let *P** be that formula with the bound variables of the initial quantifiers replaced by new free variables and the initial quantifiers stripped off. Create a new supposition which has the same supposed formulae (called Γ) as the current supposition and an ultimate interest of *P**. If the new supposition can succeed, proving that Γ derives *P**, then by universal generalization one can infer that Γ derives *P* and therefore that *P* holds in the current supposition.

Existential Instantiation

Existential instantiation occurs whenever an existential formula *P* is adopted. Let *P'* be that formula with the existential variables replaced by newly created terms (one per variable). A task is created to adopt *P'*. There is an interaction with universal instantiation and existential generalization here; all of the universally instantiated formulae have new instantiations created to be adopted using these new terms, and all of the existentially generalized formulae have new instantiations created for interest tasks using these new terms. Since universal supposition is used to prove formulae of the form "$\forall$ x Px", and universal supposition creates specially marked free variables which are always distinct from existentially instantiated terms, FELIX will not incorrectly derive "$\exists$ x P(x) $\rightarrow$ $\forall$ y P(y)". Thus, FELIX's (and OSCAR's) implementation of existential instantiation is sound.

All Detachment

All detachment, as indicated above, is a technique which is used to avoid universal instantiation. It focuses on certain instantiations, instead of all possible instantiations. The major step in the all detachment procedure is to convert the universal formulae into a clause-like form. The universal formula *P* is stripped of its leading universal

quantifiers, and the associated bound variables are converted into meta-variables (uninstantiated Prolog variables, in this implementation) to create *P*1. *P*1 is *negated* and converted into a normal form *P*2 using a set of transformations which preserve logical equivalence. The normal form is very similar to a disjunctive normal form, but is not that thorough a dismemberment of the original formula. *P*2 has the form of '*D*1 ∨ *D*2 ∨ ...', where each *Di* has the form of '*Ci*1 ∧ *Ci*2 ∧ ...'. The *Cij* may be positive or negative literals, or they may be arbitrarily complex formulae. Generally, they are literals. *P*2 is then converted to *P*3 by removing all existential quantifiers and replacing their bound variables by meta-variables. *P*3 is then converted into a set *S* of sets *T*, where each *Di* is converted to *Ti* with the *Cij* for all *j* becoming the elements of *T*.

Since *P*2 is a negation of *P*1, and *P*1 is known to be universally true, then *P*2 must be universally false. Since *P*2 is true if any *Di* is true, then all of the *Di* must be false. Thus, if some elements of *Ti* are known to be true, then the conjunction of the remaining elements of *Ti* must be false. In all detachment processing, all of the *Ti* in *S* are compared (via pattern matching with the meta-variables) to the known adopted formulae. In Pollock's original version of this algorithm, if all but one of the elements of some *Ti* are found to have been adopted, then the negation of the remaining element is adopted. The conversion of a universal formula *P* into *S* is done only once per formula, but the comparison of *S* to the adopted formulae is made every time a new formula is adopted.

This procedure makes only a small subset of the instantiations which are possible, and it only instantiates generally small elements of the original formula instead of versions of the entire original formula. As can be seen in the quantificational example, this narrow selection of instantiations can produce just the right adoptions. It may not be powerful enough in some circumstances, however, so universal instantiation is still needed for the overall theorem proving algorithm to be logically complete.

In FELIX, Pollock's approach has been relaxed in two ways. First, if some of the elements of *Ti* have been adopted, and all of the remaining elements have been completely grounded by the meta-variable bindings in the pattern matching, then gener-

ate an adoption task for the negation of the conjunction of these remaining elements of *Ti*. Second, if some of the elements of *Ti* have been adopted (let *A* be these elements), and all of the remaining elements *except one* have been completely grounded by the meta-variable bindings in the pattern matching (let the grounded elements be *G* and the single ungrounded element be *U*), then create an existential interest task in the one non-ground element *U* which has as a target the negation of the conjunction of the ground non-adopted formulae *G*, no "other unadopted support", and total support of *A*, *U* and the original universal formula. This latter extension to the all detachment formula has the curious effect of introducing an interest which is not "connected" to the ultimate interests of the supposition. This latter extension is a key step in the proof of one of the "poker game" theorems.

The all detachment procedure is tailored to a specific logic (e.g., classical or infon) by the rules used in making the transformation to the logically equivalent normal form.

There is a related procedure which has not yet shown itself to be useful (it hasn't shortened the search for any proof). The idea is to consider a formula *Q* in which there is interest, and use the all-detachment sets to find other formulae in which to be interested. This is done by considering the negation of *Q* as an adopted formula. If there is a *Ti* which has *NotQ* in it and all but one of the other elements has been adopted, and the remaining element is grounded, then add and interest task for that remaining element to the agenda. Proving that remaining element will prove *Q*.

Exists Detachment

Exists detachment is a procedure which is analogous to all detachment. The existential formula *P* is transformed into *P*1 by stripping off the leading existential quantifications and replacing the associated bound variables with meta-variables. *P*1 is transformed by nearly the same set of rules as those used for all detachment into a normal form, *P*2. In all detachment processing *P*2 is logically equivalent to the *negation* of *P*1, but for exists detachment *P*1 and *P*2 are logically equivalent. *P*2 is converted to *S* in the same fashion as for all detachment. If there is a *Ti* in *S* such that all of the elements of *Ti* have been adopted, then an adoption task for *P* is added to the agenda

and the interest in *P* is removed. If there is a *Ti* where all of the elements except one has been adopted, and the remaining element is grounded by the pattern matching which found the others to have been adopted, then an interest task for that remaining element is added to the agenda.

Adapting FELIX to Classical Logic

In the above discussion the framework of FELIX has been presented. Now the manner in which a natural deduction system formulation of classical logic is expressed in this framework is given. After having done this, the implementation of infon logic (NN) is given.

The adaptations are found in the forward reasons used in propagating adoptions, the backward reasons used to support interests, the all/exists detachment formula normalization rules, the manner in which negation is used in setting up a *reductio* supposition, and the use of all of the supposition and quantification rules. Infon logic uses different adaptations of the forward reasons, backward reasons, and normalization rules. *Reductio* negation is handled differently. As in classical logic, infon logic uses all of the supposition and quantification rules.

Classical Reductio Negation

When trying to prove any formula *P*, suppose – *P* and prove Q and – Q for any Q (a contradiction).

Classical Forward Reasons

The "forward reasons" given below are of the form X |– Y. If X is already proven, then Y can be inferred (X is a reason to infer Y).

FELIX Reason	*Derivation*	*NN Rule*
conjunction(1):	$A \wedge B \mid\!- A$	[$\wedge$ elimination]

conjunction(2):	$A \wedge B \vdash B$	[$\wedge$ elimination]
modus_ponens:	$A_1, \ldots, A_n,$	
	$[(A_1 \wedge \ldots \wedge A_n) \Rightarrow B] \vdash B$	[extension of $\Rightarrow$ elimination.]
disjunction_and_negation(1):	$(A \vee B), \neg A \vdash B$	[theorem; using modus ponens and theorem that $\neg P \vee Q \Rightarrow (P \Rightarrow Q)$, with $P = \neg A$ and $Q = B$.]
disjunction_and_negation(1):	$(A \vee B), \neg B \vdash A$	[theorem; using modus ponens and theorem that $\neg P \vee Q \Rightarrow (P \Rightarrow Q)$, with $P = \neg B$ and $Q = A$.]
negated_conjunction:	$\neg(A \wedge B) \vdash (\neg A \vee \neg B)$	[$\neg\wedge$ elimination.]
negated_disjunction:	$\neg(A \vee B) \vdash \neg A, \neg B$	[$\neg\vee$ elimination.]
double_negation:	$\neg\neg A \vdash A$	[$\neg\neg$ elimination.]
negated_conditional:	$\neg(A \Rightarrow B) \vdash A, \neg$B	[$\neg\Rightarrow$ elimination.]
contrapositive:	$(A \Rightarrow B), \neg B \vdash \neg A$	[contrapositive - modus tolens]
equivalence_defn:	$(A \Leftrightarrow B) \vdash (A \Rightarrow B), (B \Rightarrow A)$	[definition of $\Leftrightarrow$]
negated_equivalence_c:	$\neg(A \Leftrightarrow B) \vdash (A \Leftrightarrow \neg B)$	[negation of iff]
negated_universal:	$\neg(\forall\, x\, A) \vdash \exists\, x\, \neg A$	[$\neg\forall$ elimination.]
negated_existential:	$\neg(\exists\, x\, A) \vdash \forall\, x\, \neg A$	[$\neg\exists$ elimination.]

Classical Backward Reasons:

The backward reasons are of the form X |– Y. If a proof of Y is desired, then try proving X.

FELIX Reason	*Derivation*	*NN Rule*
conjunction:	$A, B \vdash (A \wedge B)$	[$\wedge$ introduction.]
negated_disjunction:	$-A, -B \vdash -(A \vee B)$	[$-\vee$ introduction.]
equivalence_defn:	$(A \Rightarrow B), (B \Rightarrow A)$ $\vdash (A \Leftrightarrow B)$	[definition of $\Leftrightarrow$]
negated_equivalence_c:	$(A \Leftrightarrow -B) \vdash -(A \Leftrightarrow B)$	[negation of iff]
disjunction_to_conditional(1):	$(-A \Rightarrow B) \vdash (A \vee B)$	[definition of $\Rightarrow$, and double negation axiom]
disjunction_to_conditional(2):	$(-B \Rightarrow A) \vdash (A \vee B)$	[definition of $\Rightarrow$, and double negation axiom]
double_negation:	$A \vdash - - -A$	[– – introduction.]
negated_conditional:	$A, -B \vdash -(A \Rightarrow B)$	[$-\Rightarrow$ introduction.]
negated_conjunction:	$(-A \vee -B) \vdash -(A \wedge B)$	[$-\wedge$ introduction.]
modus_ponens:	If $(A \Rightarrow B)$ already established: $A, (A \Rightarrow B) \vdash B$	[$\Rightarrow$ elimination.]
negated_universal:	$\exists\, x\, -A \vdash -(\forall\, x\, A)$	[$-\forall$ introduction.]
negated_existential:	$\forall\, x\, -A \vdash -(\exists\, x\, A)$	[$-\exists$ introduction.]

Normal Form Transformations

The following rules specify the transformations which are applied to a classical formula to create the normal form used in all detachment processing and exists detachment processing. These rules are applied recursively such that every subformula which can be transformed is transformed. Further, when a subformula is transformed the subformulas containing the transformed subformula are also transformed (again, perhaps).

$$P \wedge (Q \vee R) \quad \rightarrow \quad (P \wedge Q) \vee (P \wedge R)$$
$$(Q \vee R) \wedge P \quad \rightarrow \quad (Q \wedge P) \vee (R \wedge P)$$

$(P \Rightarrow Q) \quad \rightarrow \quad \neg P \vee Q$

$(P \Leftrightarrow Q) \quad \rightarrow \quad (P \wedge Q) \vee (\neg P \wedge \neg Q)$

$\exists X\,(P \vee Q) \quad \rightarrow \quad (\exists X\,P) \vee (\exists X\,Q)$

$\neg \neg P \quad \rightarrow \quad P$

$\neg (P \vee Q) \quad \rightarrow \quad \neg P \vee \neg Q$

$\neg (P \wedge Q) \quad \rightarrow \quad \neg P \wedge \neg Q$

$\neg (P \Rightarrow Q) \quad \rightarrow \quad P \wedge \neg Q$

$\neg (P \Leftrightarrow Q) \quad \rightarrow \quad (P \wedge \neg Q) \vee (\neg P \wedge Q)$

$\neg (\forall X\,P) \quad \rightarrow \quad \exists X\,\neg P$

$\neg (\exists X\,P) \quad \rightarrow \quad \forall X\,\neg P$

A Quantificational Example

This example is presented three parts; Exhibit 4. 6 on page 115, Exhibit 4. 7 on page 117, and Exhibit 4. 8 on page 118. It demonstrates several of the quantificational reasoning rules presented above; universal instantiation, all detachment, universal generalization, and existential instantiation. These can be seen in the justifications in the proof in Exhibit 4. 8 on page 118. The trace is in Exhibit 4. 6 on page 115 and Exhibit 4. 7 on page 117.

The theorem of the example is:
Given:

$\forall$ x $\exists$ y r(x, y),

$\forall$ x $\forall$ y (r(x, y)$\Rightarrow$r(y, x)),

$\forall$ x $\forall$ y $\forall$ z (r(x, y)$\wedge$r(y, z)$\Rightarrow$r(x, z)),

Prove: $\forall$ x r(x, x).

This can be paraphrased as: if ‘r’ is a relation such that for every x there exists a “right” element y (‘r(x,y)’), the ‘r’ relation is symmetric, and the ‘r’ relation is transitive, then conclude that the ‘r’ relation is reflexive.

The representation of formulae is somewhat different in FELIX to simplify its processing. A bound variable is marked with a trailing ‘?’. A free variable is an integer followed by a ‘*’. New terms created by existential instantiation are marked with a trailing ‘@’. Quantified formulae are written $\prod$ X : P and $\sum$ X : P, where X is some bound variable such as ‘y?’ and P contains one or more references to X. There may be multiple quantifications, such as ‘Q1 : Q2 : ... : Qn : P’, where Qi is either ‘$\prod$ Xi’ or ‘$\sum$ Xi’, and all of the Xi are pairwise distinct.

At the beginning of Exhibit 4. 6 on page 115 there is the addition of a task to adopt ‘$\prod$ (xc?): $\sum$ (yc?):r(xc?, yc?)’. This formula has the traditional syntax of ‘$\forall$x $\exists$y (r(x, y))’. The variables are named to be unique across all of the formulae of the problem, although it formally only necessary that variables be unique within their quantifier’s scope.

```
*************** FELIX ****************
     Logic Mode: classical

     Test: 8

Add t0 : interest - ∏ (x?):r(x?, x?)
Add t1 : adoption - ∏ (xc?): Σ (yc?):r(xc?, yc?)
Add t2 : adoption - ∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
Add t3 : adoption - ∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?,
zb?)->r(xb?, zb?))

--------------
Linear-process: Supposition s0
     Prove: [∏ (x?):r(x?, x?)]
     Suppose: []
     Agenda:t1 t2 t3 t0

Process adoption ( 1 ) : ∏ (xc?): Σ (yc?):r(xc?, yc?)
Add t4 : all_statement - ∏ (xc?): Σ (yc?):r(xc?, yc?)
Process adoption ( 1 ) : ∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
Add t5 : all_statement - ∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
Process adoption ( 1 ) : ∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?,
zb?)->r(xb?, zb?))
Add t6 : all_statement - ∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?,
zb?)->r(xb?, zb?))
Process interest: ∏ (x?):r(x?, x?)
Add t7 : interest - r(1*, 1*)
Add t8 : adoption - ∏ (xc?): Σ (yc?):r(xc?, yc?)
Add t9 : adoption - ∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
Add t10 : adoption - ∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?,
zb?)->r(xb?, zb?))
--------------
Linear-process: Supposition s1
     Prove: [r(1*, 1*)]
     Suppose: []
     Agenda:t8 t9 t10 t7

Process adoption ( 1 ) : ∏ (xc?): Σ (yc?):r(xc?, yc?)
Add t11 : all_statement - ∏ (xc?): Σ (yc?):r(xc?, yc?)
Process adoption ( 1 ) : ∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
Add t12 : all_statement - ∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
Process adoption ( 1 ) : ∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?,
zb?)->r(xb?, zb?))
Add t13 : all_statement - ∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?,
zb?)->r(xb?, zb?))
Process interest: r(1*, 1*)
Add t14 : reductio - r(1*, 1*)-(~ r(1*, 1*))
Add t15 : adoption - Σ (yc?):r(1*, yc?)
Process adoption ( 1 ) : Σ (yc?):r(1*, yc?)
Add t16 : adoption - Σ (yc?):r(yc0@, yc?)
Process adoption ( 1 ) : r(1*, yc0@)
```

Exhibit 4. 6: Reflexivity (test 8). Quantification Example - Trace. Part 1 of 3.

The “Process interest: ∏ (x?):r(x?, x?)” step in Exhibit 4. 6 on page 115 is the step which invokes universal supposition processing and thus creates supposition s1 and starts linear processing it. The four “Add ...” steps which follow it are the initial tasks being added to the agenda of supposition s1. The universal instantiation (all_statement) task is t11. Task t11 is processed after task t13 is added (there isn’t an explicit entry in the trace for processing t11). Processing task t11 creates task t15, the adoption of the instantiated version of the universal formula of t11. When the t15 adoption is processed, the existential formula of t15 is instantiated (creating the new term ‘yc0@’). This new term creation triggers a new instantiation of the universal formula of t11. This new instantiation is to be adopted by task t16. The instantiation of the existential of t15 is directly adopted, bypassing the agenda mechanism, producing the “Process adoption...” step following the adding of task t16.

In Exhibit 4. 7 on page 117 there is an example of a potentially endless cycle of adoption tasks between existential instantiation and universal instantiation. Other adoption tasks are created however which lead to the successful conclusion of the proof. A loop of the cycle starts at the first “Process adoption...” step of the exhibit: “Process adoption (1) : ∑ (yc?):r(yc0@, yc?)”. As a result of processing this existential formula, a new term is created for ‘yc?’, ‘yc1@’. The universal instantiation rule operating on the already processed universal formula ‘∏ (xc?): ∑ (yc?):r(xc?, yc?)’ leads to the next step “Add t18 : adoption - ∑ (yc?):r(yc1@, yc?)”, and also proceeds directly to the step following that: “Process adoption (1) : r(yc0@, yc1@)”. This last step leads to the adoption which ultimately breaks the cycle. A few steps later, task t18 is processed: “Process adoption (1) : ∑ (yc?):r(yc1@, yc?)”. This starts the universal/existential instantiation cycle again with ‘yc1@’ instead of ‘yc0@’.

The proof given in Exhibit 4. 8 on page 118 states that there were 10 “non-proof” adoptions compared with 9 proof adoptions. A non-proof adoption is an adoption which was not needed to support the theorem.

```
Add t17 : adoption - r(yc0@, 1*)
Process adoption ( 1 ) : Σ (yc?):r(yc0@, yc?)
Add t18 : adoption - Σ (yc?):r(yc1@, yc?)
Process adoption ( 1 ) : r(yc0@, yc1@)
Add t19 : adoption - r(yc1@, yc0@)
Add t20 : adoption - r(1*, yc1@)
Process adoption ( 1 ) : r(yc0@, 1*)
Add t21 : adoption - r(yc0@, yc0@)
Add t22 : adoption - r(1*, 2*)
Process adoption ( 1 ) : Σ (yc?):r(yc1@, yc?)
Add t23 : adoption - Σ (yc?):r(yc2@, yc?)
Process adoption ( 1 ) : r(yc1@, yc2@)
Add t24 : adoption - r(yc2@, yc1@)
Add t25 : adoption - r(yc0@, yc2@)
Process adoption ( 1 ) : r(yc1@, yc0@)
Add t26 : adoption - r(yc1@, 1*)
Add t27 : adoption - r(yc1@, yc1@)
Add t28 : adoption - r(yc0@, yc0@)
Process adoption ( 1 ) : r(1*, yc1@)
Add t29 : adoption - r(yc1@, 1*)
Add t30 : adoption - r(1*, yc2@)
Process adoption ( 1 ) : r(yc0@, yc0@)
Process adoption ( 1 ) : r(1*, 2*)
Add t31 : adoption - r(1*, 1*)
Process adoption ( 1 ) : r(1*, 1*)
Add t32 : adoption - ∏ (x?):r(x?, x?)
LINEAR PROCESS TERMINATION: change_supposition

--------------
Linear-process: Supposition s0
     Prove: [∏ (x?):r(x?, x?)]
     Suppose: []
     Agenda:t32 t4 t5 t6

Process adoption ( 1 ) : ∏ (x?):r(x?, x?)
LINEAR PROCESS TERMINATION: success_termination
Proof CPU Time: 76.71666666666666 seconds.

Total tasks: 33

     adoption = 24
     all_statement = 6
     interest = 2
     reductio = 1

Total unprocessed: 14

     adoption = 8
     all_statement = 5
     reductio = 1
```

Exhibit 4. 7: Reflexivity (test 8). Quantification Example - Trace. Part 2 of 3.

PROVE: ∏ (x?):r(x?, x?)

GIVEN: ∏ (xc?): ∑ (yc?):r(xc?, yc?)
∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))
∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?, zb?)->r(xb?, zb?))

1 Proof adoptions for this lemma.
3 Non-proof adoptions for this lemma.

Step :	*Adopted Formula*	*Support Steps*	*Justification*
1 :	∏ (x?):r(x?, x?)	LEMMA s1	universal_ generalization B

LEMMA s1

PROVE: r(1*, 1*)

8 Proof adoptions for this lemma.
7 Non-proof adoptions for this lemma.

Step :	*Adopted Formula*	*Support Steps*	*Justification*
1 :	∏ (xc?): ∑ (yc?):r(xc?, yc?)	given	input
2 :	∏ (xa?): ∏ (ya?):(r(xa?, ya?)->r(ya?, xa?))	given	input
3 :	∏ (xb?): ∏ (yb?): ∏ (zb?):(r(xb?, yb?)/\r(yb?, zb?)->r(xb?, zb?))	given	input
4 :	∑ (yc?):r(1*, yc?)	[1]	universal_ instantiation F
5 :	r(1*, yc0@)	[4]	existential_ instantiation F
6 :	r(yc0@, 1*)	[2, 5]	all_detachment F
7 :	r(1*, 2*)	[3, 6, 5]	all_detachment F
8 :	r(1*, 1*)	[7]	free_variable_ binding

9 Proof adoptions overall.
10 Non-proof adoptions overall.

Exhibit 4. 8: Reflexivity (test 8). Quantification Example - Proof. Part 3 of 3

Adapting FELIX to Infon Logic

The adaptations for infon logic are in the same places as those given above for classical logic.

Infon Reductio Negation

Reductio ad absurdam reasoning does not work in infon logic with respect to strong negation. It does hold for weak negation, however: when trying to prove any formula $\wedge P$, suppose P. This inference rule is of very limited application compared to classical *reductio* reasoning, since the formula to be proved must be a weak negation of a formula and it is rare that there is interest in proving the weak negation of a formula. *Reductio* interest registration (indirect *reductio* reasoning) is similarly modified to use weak instead of strong negation: when P is adopted in a *reductio* supposition, for each ultimate interest of the form $\wedge Q$, add an interest task in $\wedge P$ (instead of $-P$) as supporting $\wedge Q$.

Infon Forward Reasons

The "forward reasons" given below are of the form X |– Y. If X is already proven, then Y can be inferred (X is a reason to infer Y).

FELIX Reason	*Derivation*	*NN Rule*
conjunction(1):	$A \wedge B \mid\!- A$	[$\wedge$ elimination]
conjunction(1):	$A \wedge B \mid\!- B$	[$\wedge$ elimination]
modus_ponens:	$A_1, \ldots, A_n$, $[(A_1 \wedge \ldots \wedge A_n) \Rightarrow B] \mid\!- B$	[extension of $\Rightarrow$ elimination.]
disjunction_and_negation(1):	$(A \vee B), -A \mid\!- B$	[theorem; using modus ponens and theorem that $-P \vee Q \Rightarrow (P \Rightarrow$

		Q), with $P = -A$ and $Q = B$. No immediate counterpart in NN.]
disjunction_and_negation(2):	$(A \vee B), -B \vdash A$	[theorem; using modus ponens and theorem that $-P \vee Q \Rightarrow (P \Rightarrow Q)$, with $P = -B$ and $Q = A$. No immediate counterpart in NN.]
negated_conjunction:	$-(A \wedge B) \vdash (-A \vee -B)$	[$-\wedge$ elimination.]
negated_disjunction:	$-(A \vee B) \vdash -A, -B$	[$-\vee$ elimination.]
double_negation:	$- - A \vdash A$	[$- -$ elimination.]
negated_conditional:	$-(A \Rightarrow B) \vdash A, -B$	[$-\Rightarrow$ elimination.]
equivalence_defn:	$(A \Leftrightarrow B)$	
	$\vdash (A \Rightarrow B), (B \Rightarrow A)$	[definition of $\Leftrightarrow$]
negated_equivalence_i:	$-(A \Leftrightarrow B)$	
	$\vdash (A \wedge -B) \vee (B \wedge -A)$	[By definition of $\Leftrightarrow$, and axioms 1, 2, and c of NH]
negated_universal:	$-(\forall\ x\ A) \vdash \exists\ x\ -A$	[$-\forall$ elimination.]
negated_existential:	$-(\exists\ x\ A) \vdash \forall\ x\ -A$	[$-\exists$ elimination.]
strong_and_weak_negation:	$-(\wedge A) \vdash A$	[theorem for strong and weak negation].

Infon Backward Reasons

The backward reasons are of the form X |– Y. If a proof of Y is desired, then try proving X.

FELIX Reason	*Derivation*	*NN Rule*
conjunction:	$A, B \vdash (A \wedge B)$	[$\wedge$ introduction.]
negated_disjunction:	$-A, -B \vdash -(A \vee B)$	[$-\vee$ introduction.]

disjunction_and_negation:	$(A \vee B), -A \vdash B$	[theorem; using modus ponens and theorem that $-P \vee Q \Rightarrow (P \Rightarrow Q)$, with $P = -A$ and $Q = B$. No immediate counterpart in NN.]
equivalence_defn:	$(A \Rightarrow B), (B \Rightarrow A)$ $\vdash (A \Leftrightarrow B)$	[definition of $\Leftrightarrow$]
negated_equivalence_i:	$-(A \Leftrightarrow B)$ $\vdash (A \wedge -B) \vee (B \wedge -A)$	[By definition of $\Leftrightarrow$, and axioms 1, 2, and c of NH]
disjunction(1):	$A \vdash (A \vee B)$	[$\vee$ introduction.]
disjunction(2):	$B \vdash (A \vee B)$	[$\vee$ introduction.]
double_negation:	$A \vdash - - -A$	[– – introduction.]
negated_conditional:	$A, -B \vdash -(A \Rightarrow B)$	[–$\Rightarrow$ introduction.]
negated_conjunction:	$(-A \vee -B) \vdash -(A \wedge B)$	[–$\wedge$ introduction.]
modus_ponens:	If $(A \Rightarrow B)$ already established: $A, (A \Rightarrow B) \vdash B$	[$\Rightarrow$ elimination.]
negated_universal:	$\exists x -A \vdash -(\forall x A)$	[–$\forall$ introduction.]
negated_existential:	$\forall x -A \vdash -(\exists x A)$	[–$\exists$ introduction.]
strong_to_weak:	$-A \vdash \wedge A$	[Theorem: strong negation implies weak negation.]

Infon Normal Form Transformations

The following rules specify the transformations which are applied to a formula to create the normal form used in all detachment processing and exists detachment processing in infon logic mode.

$P \wedge (Q \vee R)$	$\rightarrow$	$(P \wedge Q) \vee (P \wedge R)$
$(Q \vee R) \wedge P$	$\rightarrow$	$(Q \wedge P) \vee (R \wedge P)$
$(P \Leftrightarrow Q)$	$\rightarrow$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
$\exists X (P \vee Q)$	$\rightarrow$	$(\exists X P) \vee (\exists X Q)$
$- - P$	$\rightarrow$	P
$-(P \vee Q)$	$\rightarrow$	$-P \vee -Q$
$-(P \wedge Q)$	$\rightarrow$	$-P \wedge -Q$
$-(P \Rightarrow Q)$	$\rightarrow$	$P \wedge -Q$
$-(P \Leftrightarrow Q)$	$\rightarrow$	$(P \wedge -Q) \vee (-P \wedge Q)$
$-(\forall X P)$	$\rightarrow$	$\exists X -P$
$-(\exists X P)$	$\rightarrow$	$\forall X -P$

Comparing infon and classical FELIX proofs

Infon logic is weaker than classical logic in the sense that everything which can hold in infon logic also holds in classical logic, but there are theorems of classical logic which do not hold for infon logic. There are simple examples of consequences which hold classically but not infonically: $- p \Rightarrow p \vdash p$, and $\vdash p \vee - p$. The first consequence statement is essentially the claim that *reductio ad absurdam* inference doesn't hold in infon logic. The second consequence statement is that one cannot always assume in infon logic that either a formula or its (strong) negation is true.

Theorems of classical logic which involve conditionals may not be theorems of infon logic, since the infon conditional is a weaker operator than the classical conditional, i.e. the infon conditional is not equivalent to the disjunction of its consequence and the negation of its antecedent. There is a "stronger" form of the classical theorem which *is* also a theorem of infon logic, however. An example of this is a theorem about "Russell" sets and the existence of complements.

A Russell set is a the set of all sets which do not contain themselves. Russell introduced this definition of a set to explore an apparent paradox in set theory. An anti-Russell set contains all of the sets which *do* contain themselves. A theorem of classical logic is: if there exists an anti-Russell set, then there exists some set which does not have a complement. There are two classically equivalent formulations of this theorem such that one of the formulations is a theorem of infon logic and the other formulation is not.

To formalize the anti-Russell theorem, let 'f(X, Y)' mean "X is a member of Y". The statement that "there exists an anti-Russell set" can be formalized as: $(\exists x \forall y (f(y,y) \Leftrightarrow f(y,x)))$, 'x' is the anti-Russell set (if it exists). The statement that "a set 'u' has a complement" can be formalized as: $\exists v \forall w (\sim f(w,u) \Leftrightarrow f(w,v))$, where 'v' is the complement of 'u' (if it exists), and 'w' are the elements of 'u' and 'v'.[10] Thus, the

[10] Since the domain of 'w' is unrestricted, then the set to which 'u' is bound must be in either 'u' or 'v' (its complement). Thus, by this formulation, if every set has a complement, then every set must contain itself or must be contained in its complement. This is a much broader notion of complement than is generally encountered.

statement that “there exists some set which does not have a complement” can be formalized by: ∃ u ~(∃ v ∀ w (~f(w,u) ⇔ f(w,v))). The negation can be moved out to produce: ~(∀ u ∃ v ∀ w (~f(w,u) ⇔ f(w,v))). The theorem can be stated formally as: (∃ x ∀ y (f(y,y) ⇔ f(y,x))) => ~(∀ u ∃ v ∀ w (~f(w,u) ⇔ f(w,v))).

This version of the theorem is true classically but does not hold in infon logic. A simple equivalence of classical logic can be used to convert it to an infonically valid theorem, however: (P ⇔ Q) is classically (but not infonically) equivalent to ~(P ⇔ ~Q). Applying this equivalence to the antecedent produces the infonically valid: (∃ x ∀ y ~(f(y,y) ⇔ ~f(y,x))) => ~(∀ u ∃ v ∀ w (~f(w,u) ⇔ f(w,v))). The classical logic and infon logic proofs of this latter statement are given at the end of this chapter, since they are long.

The infon proof is longer than it needs to be due to a lack of intelligence in FELIX. A dilemma derivation is used in the main proof which is poorly considered. Three formulae are needed to support the result ‘R’ of a dilemma proof: ‘P∨Q’, ‘P=>R’, and ‘Q=>R’. FELIX, having proved ‘P∨Q’, sets out to establish the other two formulae. To prove ‘P=>R’, FELIX supposes ‘P’ and derives ‘R’. FELIX never actually used ‘P’ in its proof of ‘R’, however; the other given formulae were sufficient to the task. Thus, the lemma which ostensibly proves ‘P=>R’ actually proves the much stronger ‘R’. If FELIX were more clever, it would recognize this and not proceed with the dilemma proof on which it had originally embarked.

Even the more clever version of the infon proof must be longer than the classical proof. The difference is chiefly in the “more powerful” negated_equivalence_c rule used in the classical proof, compared with the negated_equivalence_i rule used in the infon proof.

Soundness of the FELIX Algorithm

Each of the various aspects of the FELIX algorithm presented above can be justified in terms of classical or infon logic. This shows that the FELIX algorithm is *sound* with respect to these logics. The completeness of the algorithm is discussed later.

Conditional supposition

The fundamental suppositional reasoning rule is "conditional supposition". This rule embodies the idea that 'q ⇒ r' can be derived from some set of wffs W by showing that if one supposes q (thereby adding q to W), then r can be derived. This is the '⇒' introduction rule for NN.

Also, conditional supposition can be viewed as implementing the deduction theorem: $\varphi \cup \{A\} \mid\!-_{NH} B$ iff $\varphi \mid\!-_{NH} A \Rightarrow B$.

reductio supposition and reductio interest

The *reductio* supposition rule for classical logic can be described using the Scott consequence relation (SCR) for classical logic, '$\|\!-_C$', as: $\varphi \cup \{-A\} \|\!-_C \{A\}$ implies $\varphi \|\!-_C \{A\}$. That is, if A is a consequence of a set of formulae φ and its negation $-A$, then A is a consequence of the set of formulae φ alone. This can be proven using the deduction theorem and some well-known theorems of classical logic:

1) $\varphi \cup \{-A\} \|\!-_C \{A\}$ iff $\varphi \|\!-_C \{-A \Rightarrow A\}$. [By the deduction theorem]
2) $\varphi \|\!-_C \{-A \Rightarrow A\}$ iff $\varphi \|\!-_C \{--A \vee A\}$. [By classical logic: $(P \Rightarrow Q) \Leftrightarrow (-P \vee Q)$]
3) $\varphi \|\!-_C \{--A \vee A\}$ iff $\varphi \|\!-_C \{A\}$. [By classical logic: $--P \Leftrightarrow P$, and $(P \vee P) \Leftrightarrow P$]
4) $\varphi \|\!-_C \{-A \Rightarrow A\}$ iff $\varphi \|\!-_C \{A\}$. [By steps 2 and 3 and transitivity of iff.]
5) $\varphi \cup \{-A\} \|\!-_C \{A\}$ iff $\varphi \|\!-_C \{A\}$. [By steps 1 and 4 and transitivity of iff.]

QED.

The above proof relies on the classical logic supporting four theorems: the deduction theorem, $(P \Rightarrow Q) \Leftrightarrow (-P \vee Q)$, $--P \Leftrightarrow P$, and $(P \vee P) \Leftrightarrow P$.

The *reductio ad absurdam* rule does not hold for strong negation in infon logic. This can be seen by looking at transforming the classical proof of the rule to an infonic proof. Three of the supporting theorems are true in infon logic, and one is not. The deduction theorem is true by a lemma of [Gabbay 1981]. $(- -P \Leftrightarrow P)$ is a theorem of infon logic - it is an axiom of NH. $(P \vee P) \Leftrightarrow P$ is a theorem of infon logic as proved in NH Theorem 2.

NH Theorem 1: $|-_{NH} (P \Rightarrow P)$.

Proof:

1) $\{P\}|-_{NH} P$. [By definition of TCR.]

2) $|-_{NH} (P \Rightarrow P)$. [By deduction theorem for $|-_{NH}$.]

QED, NH Theorem 1.

NH Theorem 2: $|-_{NH} ((P \vee P) \Leftrightarrow P)$.

Proof:

1) $(P \vee P) \Leftarrow P$. [By axiom of NH.]
2) $(P \Rightarrow P) \Rightarrow ((P \Rightarrow P) \Rightarrow ((P \vee P) \Rightarrow P))$. [By axiom of NH.]
3) $(P \Rightarrow P) \Rightarrow ((P \vee P) \Rightarrow P)$. [By step 2, NH Theorem 1, and modus ponens for NH]
4) $(P \vee P) \Rightarrow P$. [By step 3, NH Theorem 1, and modus ponens for NH]
5) $(P \vee P) \Leftrightarrow P$. [By steps 1 and 4 and definition of $\Leftrightarrow$.]

QED, NH Theorem 2.

The theorem which does not hold for infon logic on which the proof of the validity of the *reductio ad absurdam* rule rests is the equivalence of the conditional and disjunction operators, $((P \Rightarrow Q) \Leftrightarrow (-P \vee Q))$. NH Theorem 3 proves the negative result that the schema $((P \Rightarrow Q) \Leftrightarrow (-P \vee Q))$ does not hold for all schema substitutions in infon logic.

NH Theorem 3: $\sim(|-_{NH} ((P \Rightarrow Q) \Leftrightarrow (-P \vee Q)))$.

The proof is by contradiction.

Proof:

1) $((P \Rightarrow Q) \Leftrightarrow (-P \vee Q))$. [Contradiction of theorem.]

2) $(P \Rightarrow P) \Leftrightarrow (-P \vee P)$. [By substitution of P for Q in step 1.]

3) $(-P \vee P)$. [By step 2, NH Theorem 1, and modus ponens]

4) $\sim(|-_{NH} ((P \Rightarrow Q) \Leftrightarrow (-P \vee Q)))$. [Step 3 contradicts the fact that NH does not validate the disjunctive tautology, as shown by Gabbay. Therefore, the assumption of step 1 must be wrong.]

QED, NH Theorem 3.

Since $(P \Rightarrow Q) \Leftrightarrow (-P \vee Q)$ does *not* hold for infon logic, the proof used above for the *reductio ad absurdam* technique does not hold in infon logic.

A more direct argument against strong *reductio ad absurdam* reasoning in infon logic is: If φ is consistent (i.e., $\sim(\varphi |- \boldsymbol{f})$) and $\varphi |- -A \Rightarrow A$, then $\sim(\varphi |- -A)$. This is shown by the fact that modus ponens, $-A$ and $-A \Rightarrow A$ derive A, and inconsistency. Thus, $-A \Rightarrow A$ implies $-A \Rightarrow \boldsymbol{f}$. This latter statement is the weak (intuitionistic) negation of the dual of A, written "$\wedge -A$", which is *not* equivalent to A. This can be read as "The dual of A is not supported."

Thus, strong *reductio ad absurdam* reasoning can't be used for infon logic.

There is a weak form of *reductio ad absurdam* reasoning which the above argument *does* warrant, however. Let '$\wedge$' be the infon weak negation symbol defined as '$\wedge P$' $=_{df}$ '$P \Rightarrow \boldsymbol{f}$'. The weak *reductio ad absurdam* argument supported in NH is: $A \Rightarrow \wedge A$ implies $\wedge A$, and $(A \Rightarrow \wedge A) \Rightarrow \wedge A$. Also, $A \Rightarrow B$ and $A \Rightarrow \wedge B$ combine to derive $\wedge$ A (the weak negation introduction rule of NN). Thus, FELIX's *reductio ad absurdam* reasoning can be used in infon logic to infer weakly negated formulae, but not positive or strongly negated formulae.

Dilemma supposition

There is a theorem (schema) of both classical and infon logic which warrants the dilemma supposition reasoning. The theorem is: '$(A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \vee B \Rightarrow$

C))'. The dilemma supposition procedure can be stated in the terms of this theorem: To prove that '$A \vee B \Rightarrow C$', prove '$A \Rightarrow C$' and '$B \Rightarrow C$'. This is done by supposing 'A' and deriving 'C', to prove '$A \Rightarrow C$'(as in conditional supposition), then supposing 'B' (without supposing 'A') and deriving 'C' to prove '$B \Rightarrow C$'. This kind of reasoning is an implementation of the disjunction elimination rule of NN.

Universal supposition

Universal supposition reasoning implements universal generalization. It tries to prove a version of the formula with the universally quantified variable replaced by a new free variable, if this works then the original universally quantified formula holds. Universal generalization is valid in both classical and infon logic. This implements the '$\forall$' introduction rule of NN.

Universal instantiation

Universal instantiation may be applied by FELIX to any adopted universally quantified formula. Versions of the universal formula are created by substituting every known (to FELIX) term into the formula. Each of these substitution formulae is adopted. Universal instantiation is valid for classical logic. It is an implementation of the $\forall$ elimination rule of NN.

All Detachment

The "all detachment" technique is an optimization of "universal instantiation". According to universal instantiation, when $\forall x A(x)$ is adopted then every version of $A(c)$ is adopted for all c in the set of known terms (for the current supposition). This is valid as discussed above for both classical and infon logic. All detachment reasoning is a specialization of universal instantiation. First, $\forall x A(x)$ is processed into a normal

form $\{\{A_{11}, ..., A_{1n_1}\}, ..., \{A_{k1},..., A_{kn_k}\}\}$, where $-A(x) = ((A_{11} \wedge \ldots \wedge A_{1n_1}) \vee \ldots \vee (A_{k1} \wedge \ldots \wedge A_{kn_k}))$. Each time an adoption is made, if there is a set $\{A_{k1},..., A_{kn_k}\}$ such that all of the A_{ki} have been adopted except one, A_{ji} , then $-A_{ji}$ is adopted. The key to the validity of this approach is that the normal form is logically equivalent to the unnormalized form, since the input and output forms of each of the transformation rules which is applied to create the normal form are logically equivalent.

Existential instantiation

Existential instantiation is may be applied when an existentially quantified formula, *P*, is adopted. A new formula is created by substituting a newly created atomic term for the existentially quantified variable in *P*.[11] This new formula is adopted. This is an indirect implementation of the existential elimination rule of NN, which is derived directely from OSCAR.

Existential generalization

Existential generalization proves an existentially quantified formula *P* by proving any version of *P* where some (known) term has been substituted for the existentially quantified variable. Existential generalization is classically valid. It is infonically valid since it implements the existential introduction rule of NN.

Existential Detachment

Existential detachment creates a normal form of a formula *P* of interest via the same transformation mechanism used in all detachment reasoning. This produces a set *S* of sets *T*, where if all of the elements in any *Ti* are true, then there is an instance of *P*

[11] To avoid violating the unique names assumption, the new existential term is a skolem *function*.

which is true, and therefore *P* is true. This is a special mechanism for existential instantiation reasoning.

Forwards and backwards reasons

The forward and backward reasons are each logically valid. The element of logic which warrants each reason is given next to that reason in the lists of these reasons given in sections titled “Adapting FELIX to Classical Logic” and “Adapting FELIX to Infon Logic”.

Completeness

The theorem prover outlined above, Infon FELIX, is not complete for the infon logic. The problem stems from the (A and –A) |- B rule. This rule is not explicitly present in the theorem prover, and theorems which are instances of it cannot be proved by the techniques available to Infon FELIX. In Classical FELIX, this can be established via the *reductio ad absurdam* technique. This technique is not available to Infon FELIX, however. This is only a problem when dealing with theorems with inconsistent antecedents. Since Infon FELIX implements all of the other rules of NN (as discussed below), Infon FELIX is complete with respect to non-quantificational theorems with consistent antecedents. There is a problem in the current implementation of FELIX (in both infon and classical modes) where the search procedure can get into an infinite cycle of alternating existential instantiations and universal instantiations. It should be possible to fix FELIX so that it recognizes this situation and avoids it, but it has not been done. Thus, FELIX is *nearly* complete for the full infon and classical logics.

Every theorem of infon logic can be proved by Infon FELIX, excepting certain quantificational theorems as mentioned above. The time needed to do this, and the number of steps of the resulting proof, is unbounded (although finite). Infon FELIX is only a semi-decision procedure - as is true of the best one can do for classical first order predicate calculus. That is, if the answer to the query ("is P a theorem of NH?") is yes, then Infon FELIX will correctly answer yes in finite (if unbounded) time (for all non-quantificational theorems and most quantificational ones). If the correct answer to the query is no, however, then Infon FELIX may not halt. Thus, if it answers, then its answer is correct.

The completeness of Infon FELIX (excepting weak negation) can be shown be showing that all of the rules of NN are used by Infon FELIX. As part of the soundness argument, it is also shown that the inference rules of Infon FELIX are all derivable from NN rules.

NN Rules in FELIX

All of the rules in NN are implemented in FELIX as follows:

⇒ introduction rule implemented via conditional supposition.

⇒ elimination rule implemented via modus ponens backwards and forwards reasons.

∨ elimination rule implemented via dilemma supposition

∨ introduction rule implemented via disjunction backwards reasons.

∧ introduction rule implemented via conjunction backwards reason.

∧ elimination rule implemented via conjunction forwards reasons.

¬ introduction and elimination rules are not currently implemented. These can be implemented via *reductio* supposition, but they substantially reduce the efficiency of the system.

¬ introduction rule implemented via strong-to-weak backwards reason.

– – introduction rule implemented via double negation backwards reason.

– – elimination rule implemented via double negation forwards reason.

–¬ introduction rule not implemented. This can be done via a strong and weak negation backwards reason.

–¬ elimination rule implemented via strong and weak negation forwards reason.

– ⇒ introduction rule implemented via negated conditional backwards reason.

– ⇒ elimination rule implemented via negated conditional forwards reasons.

– ∧ introduction rule implemented via negated conjunction backwards reason.

– ∧ elimination rule implemented via negated conjunction forwards reason.

– ∨ introduction rule implemented via negated disjunction backwards reason.

– ∨ elimination rule implemented via negated disjunction forwards reason.

∃ introduction implemented via existential generalization and exists detachment.

∃ elimination implemented (indirectly) via existential instantiation.

∀ introduction implemented via universal supposition (universal generalization).

∀ elimination implemented via universal instantiation and “all” detachment.

–∃ introduction implemented via negated existential backwards reason.

–∃ elimination implemented via negated existential forwards reason.

–∀ introduction implemented via negated universal backwards reason.

–∀ elimination implemented via negated universal forwards reason.

The above shows that all of the rules of NN have counterparts in FELIX.

Additional FELIX rules

There are additional reasons implemented in FELIX which are based on theorems of infon logic. These theorems and their implementations are as follows:

The rule that (A ∨ – B) and B derive A is derived from the basic rules of NN.
This rule is implemented via the disjunction and negation forwards and backwards reasons.

⇔ introduction and elimination rules are derived from the definition of equivalence in terms of the =>.

⇔ introduction is implemented via equivalence definition backwards reason.

⇔ elimination is implemented via equivalence forwards reasons.

– ⇔ elimination and introduction rules are derived from definition of ⇔.

– ⇔ introduction rule is implemented via negated equivalence infon backwards reason.

– ⇔ elimination rule is implemented via negated equivalence infon forwards reason.

There is a theorem of NN that given P => Q, Q => R, and P, one can derive R. A special case of this theorem, when Q is a “compound” formula, is implemented via indirect interest adoption.

The above shows that all of the rules of Infon FELIX are derived from NN rules.

The Implementation of FELIX

FELIX is implemented in LPA MacProlog and runs on an Apple Macintosh II computer. The source code of FELIX is divided into several files along functional lines:

Control, Logic, Test Cases, Interface, and Utilities. The Control file implements the search mechanism for FELIX. The Logic file implements the specializations of FELIX for the various kinds of logic which it supports: infon, classical, and belief. The Test Cases file provides all of the test cases executed for this thesis. The Utilities file implements the output procedures for identifying the search steps which contribute to the final proof and displaying that proof, and various FELIX-specific "low-level" procedures used by the other files. Much of the code of the Utilities file is devoted to displaying the proof. The Write Columns file implements a procedure for displaying text in columns, with appropriate line wrapping with columns. The Interface file implements the user interface (primarily, convenient ways to execute the test cases and control of the tracing of the execution of FELIX). The Master file pulls all of the other files together. There are also some general-purpose low-level utility files used: List Utilities, String Utilities, and Term Utilities. The List Utilities file provides various procedures for handling lists (for example; finding elements in lists, intersecting lists, and unioning lists). The String Utilities file provides procedures for handling strings. The Term Utilities file implements a procedure for "copying" Prolog terms.

The implementation of FELIX is large. Rather than simply count source lines, which is subject to formatting vagaries and thus a highly inaccurate estimate of code size, a "conceptual" line count is used. The total size of the program can be calculated as the clause count plus the goal count. This is a "conceptual" line count: one line of source code for each goal (since each goal should be on a line by itself for good programming style), and one additional line of source code for each clause since the *head* of a clause is not counted by the "goal" count and the head should also be on a line by itself. This "conceptual" line count does not count any whitespace (blank lines) or comments. Nor does it allow for a single goal spanning several lines (which can be necessary when the goal has many arguments and/or the arguments to the goal are long). The counts for FELIX are as follows:

Source File	*Lines*	*Procedures*	*Clauses*	*Goals*
Control	596	37	115	481
Logic	1080	89	196	884
Utilities: General	1615	110	354	1261

Utilities: Trace&Proof Output	707	73	140	567
Write Columns	168	14	37	131
Test Cases	91	3	44	47
Interface	60	3	11	49
Master	31	2	3	28
List Utilities	136	24	40	96
String Utilities	115	9	21	94
Term Utilities	4	1	1	3
Total	4603	365	962	3641

The above measures show that FELIX is a fairly large system, among systems implemented in Prolog.

Barwise 1986 "Conditionals and Conditional Information" by Jon Barwise, in *On Conditionals* by Traugott, et. al. (eds.). Cambridge University Press, 1986. Reprinted on p. 97-135 in *The Situation in Logic* by Jon Barwise, Center for the Study of Language and Information: Stanford University, 1988.

Gabbay 1981 *Semantical Investigations in Heyting's Intuitionistic Logic* by Dov M. Gabbay. Boston : D. Reidel Publishing Company. 1981.

Gibbard 1981 "Two Recent Theories of Conditionals" by Allan Gibbard in: *Ifs: Conditionals, Belief, Decision, Chance and Time* edited by W. L. Harper, R. Stalnaker, and G. Pearce. Dordrecht: Reidel. 1981.

Pollock 1990 "Interest Driven Suppositional Reasoning" by John L. Pollock, in *Journal of Automated Reasoning* **6**:419-461, 1990.

Stalnaker 1984 *Inquiry* by Robert Stalnaker. Cambridge, Massachusetts: MIT Press. 1984.