Formal Logic via Functional Programming

(A final-year project submitted in support of a BA.)

J.J. LEIFER
Worcester College
Walton St.
Oxford OX1 2HB
Email: m92jl@ecs.ox.ac.uk

Supervised by

C.A.R. HOARE B.A. SUFRIN
Oxford University Programming Research Group
Wolfson Building
Parks Road
Oxford OX1 3QD

April 1995

1 Introduction

We investigate how formal logic can be introduced to students who are familiar with functional programming in a way that takes advantage of their familiarity with higher order functions, free data-types, homomorphisms, and induction principles. In our experience, students often struggle with formal logic because they are unclear about the distinction between theorems and metatheorems, the distinction between syntactic constructors and semantic operators (and hence the meaning of models and valuations), and the induction and recursion principles over proofs. Using a functional programming notation as a metalanguage clears up these ambiguities because of the imposed type discipline: theorems and metatheorems have distinctive types, so are easily distinguished; when operators are overloaded (for example, when they are used both in syntax and semantics) their different types can be written out; and the recursion and induction principles over proofs become straightforward because the data-type for proofs is explicitly described. As an added benefit, proofs —which are tree-structured— need not be arbitrarily linearized just so that natural number recursion and induction can be performed on them.

We present a Gofer [Jones91] functional program script that defines data-types for representing well-formed formulas, proofs, and sequents in the propositional logic. We then discuss the implementation of theorem and inference schemas and illustrate the latter by defining a function *ded* that provides a constructive proof of the Deduction Theorem.

We display a type for representing a boolean model of our logic and define the notion of a valuation for mapping well-formed formulas to booleans. We describe a proof of consistency with respect to the model.

We then introduce new data-types for representing tactics for proving results in our logic and define higher-order tactic combinators ("tacticals") for composing tactics and unification support functions for generating tactics from the specifications of inference rules. We describe an automatic theorem prover, defined as a composition of inference tactics, whose domain is exactly the set of all tautological sequents — thus justifying constructively the completeness of our logic. As a corollary, we show that in the presence of the Deduction Theorem any uniform inference rule (i.e., one that respects substitutions to its arguments) has a normal form that does not explicitly involve the examination of the history its proof arguments. This result implies that hiding or discarding the history of a proof as it is built does not restrict the set of uniform inference rules that can be constructed.

We conclude by comparing our approach to that taken in LCF and related systems.

2 Wffs, proofs, and sequents

In the next two sections we build up the Hilbert propositional logic by representing well-formed formulas and proofs as freely-generated tree types and sequents as pairs containing the list of hypotheses and conclusion of a proof. The type for proofs allows constructions that represent unsound proofs as well as sound ones, so we define a proof-checker *gp* that tests a proof tree for internal consistency. We conclude by defining a a "claim"-checker (*just*) that tests if a proof justifies a particular sequent.

— * —

We begin by defining the fundamental data-structures for well-formed formulas (wffs), proofs, and sequents. Wffs are constructed from propositional variables, two constants, binary implication, and unary negation:

```
data Wff  =  Var String
          |  T
          |  F
          |  Not Wff
          |  Wff :=> Wff
```

The following code defines *isImp*, which tests if the top level constructor of a wff is :=>, and two projection functions *limp* and *rimp* for extracting the left and right sides of an implication.

```
isImp            ::  Wff -> Bool
isImp (_ :=> _)   =  True
isImp _           =  False

limp, rimp       ::  Wff -> Wff
limp (x :=> _)    =  x
rimp (_ :=> x)    =  x
```

Function *vars* calculates a list (with no duplicates) of the variables used in a wff:

```
vars           :: Wff -> [String]
vars (x :=> y) = vars x  `union`  vars y
vars (Not x)   = vars x
vars (Var s)   = [s]
vars _         = []
```

— * —

Proofs are trees with axioms as the leaves and primitive inference rules as the internal nodes. In the approach described next, we represent proofs by a minimal encoding with each node containing *only* the reason —the theorem or inference schema— and its parameters justifying the step. The sequent justified by the entire proof (the "last line of the proof" so to speak) is extracted by function *seq*.

A proof is formed from six axiom schemas (*K*, *S*, *CP*, *Top*, *Bot*, *Hyp*) with their instantiation parameters and the application of inference rule Modus Ponens (*MP*) to two subproofs:

```
data Prf  =  K   Wff Wff
          |  S   Wff Wff Wff
          |  CP  Wff Wff
          |  Top Wff
          |  Bot Wff
          |  Hyp Wff
          |  MP  Prf Prf
```

Sequents are pairs containing the hypotheses and the conclusion of a proof. Function *seq* extracts a sequent from a proof by instantiating each schema with its specified parameters (and, in the process reveals the definition of each axiom schema and of Modus Ponens):

```
data Seq        =  [Wff] :|- Wff

seq             :: Prf -> Seq
seq (K x y)     =  []   :|-  x :=> (y :=> x)
seq (S x y z)   =  []   :|-  (x :=> (y :=> z))
                             :=> ((x :=> y) :=> (x :=> z))
seq (CP x y)    =  []   :|-  (Not x :=> Not y) :=> (y :=> x)
seq (Top x)     =  []   :|-  x :=> T
seq (Bot x)     =  []   :|-  F :=> x
seq (Hyp x)     =  [x]  :|-  x
seq (MP p q)    =  ws `union` hs  :|-  y,          if w == x
                   where      ws  :|-  w       =  seq p
                              hs  :|-  x :=> y  =  seq q
```

In the above, *union* calculates the union of two sets represented as lists; it is defined in Appendix D along with *without* —for removing a singleton from a list— and *subset* —for comparing two lists by the set-inclusion relation.

Function *con* extracts the conclusion of a proof and *hyps* extracts the hypotheses:

```
con     ::  Prf -> Wff
con p    =  x   where _  :|- x  =  seq p
hyps    ::  Prf -> [Wff]
hyps p   =  hs  where hs :|- _  =  seq p
```

— * —

Function *seq* is partial because its *MP* case is not guarded exhaustively, so it causes an error if applied to a proof involving an incorrect use of *MP*. We define a total function *gp* —for "good proof"— that tests if a proof is in the domain of *seq* by checking that each instance of an application of *MP* involves two subproofs that correctly match:

```
gp          ::  Prf -> Bool
gp (MP p q)  =  gp p  &&  gp q  &&  isImp cq  &&  cp == limp cq
                where cp  =  con p
                      cq  =  con q
gp _         =  True
```

(Function *gp* is total because `&&` evaluates lazily from left to right.)

Finally, the total function *just* tests if a proof justifies a sequent — i.e., if the proof is good and has the same conclusion and a subset of the hypotheses that the sequent has.

```
just               :: Prf -> Seq -> Bool
p `just` (hs :|- x) = gp p  &&  con p == x  &&  hyps p `subset` hs
```

3 Judgments and theorem schemas

We have finished constructing the fundamental data types for representing objects in our logic and now turn our attention in the following sections to the problems of finding a proof that justifies a particular sequent. In this section, we define the turnstile symbol '$\vdash$' for making judgments about the existence of proofs and we demonstrate how to satisfy such claims in two examples: in the first we exhibit a value of type *Prf*, and, in the second, a function that takes any wff and generates a proof parameterized by that wff.

— * —

The turnstile '$\vdash$' is defined formally as follows: For $w: \mathit{Wff}$, $ws{:}[\mathit{Wff}]$,

$$ws \vdash w \quad \text{iff} \quad (\exists p{:}\mathit{Prf} \bullet\ p\ \ `\mathit{just}`\ \ ws\ \texttt{:|-}\ w)$$

As an example of a judgment about our proof system, consider the following claim:

$$[\,] \;\vdash\; (\mathit{Var}\,\text{"}\mathtt{z}\text{"}) \mathrel{\mathtt{:=>}} (\mathit{Var}\,\text{"}\mathtt{z}\text{"})$$

Proof p justifies this claim:

```
p  ::  Prf
p   =  MP (K x x) (MP (K x (x :=> x)) (S x (x :=> x) x))
       where x = Var "z"
```

because Gofer reduces

```
p  `just`   [] :|- (Var "z") :=> (Var "z")
```

to `True`. The choice of Var "z" is rather arbitrary so we now consider a more general statement:

$$(\forall x : \mathit{Wff} \;\bullet\; [\,] \;\vdash\; x \mathrel{\mathtt{:=>}} x) \qquad . \tag{1}$$

It is justified constructively by writing a total function *reflex* that generates a proof of [ ] :|- x :=> x for any wff x :

$$\begin{array}{l} \mathit{reflex} : \mathit{Wff} \rightarrow \mathit{Prf} \\ \mathit{reflex}\ x \quad \text{`}\mathit{just}\text{`} \quad [\,] \mathrel{\mathtt{:|-}} x \mathrel{\mathtt{:=>}} x \end{array}$$

Here is such a function:

```
reflex   ::  Wff -> Prf
reflex x  =  MP (K x x) (MP (K x (x :=> x)) (S x (x :=> x) x))
```

The proof of its correctness is simple (involving only normal order unfoldings) and is constructed automatically by a system such as Jape or Boyer-Moore. Fig. 1 shows an example of the use of *reflex* and the pretty-printer *showPrf* : *Prf* $\rightarrow$ *String* (defined in Appendix C) which linearizes a proof and decorates each step with the sequent known to hold at that point. (To avoid clutter, the sequents are printed without the colons in :|- and :=> and without an explicit *Var* and quote marks around each variable.)

— * —

Logicians refer to (1) as a *theorem schema* because $[\,] \vdash x$:=> x holds for *any* wff x. The corresponding notion in our context is of a function (*reflex*) that maps any wff argument x to a proof of [ ] :|- x :=> x — where universal quantification in the former ("for all x ...") is replaced by functional abstraction in the latter ("any argument x ..."). Because of this connection,

```
0    |- z => (z => z)                       K z, z
1    |- z => ((z => z) => z)                K z, z => z
2    |- (z => ((z => z) => z))
        => ((z => (z => z)) => (z => z))    S z, z => z, z
3    |- (z => (z => z)) => (z => z)         MP 1, 2
4    |- z => z                              MP 0, 3
```

Figure 1. `showPrf (reflex (Var "z"))`.

we say that *reflex* is as an example of a theorem schema, which we define as a value of type

$$\underbrace{\mathit{Wff} \rightarrow \cdots \rightarrow}_{\text{zero or more}} \mathit{Prf}$$

that justifies a particular sequent parameterized by the arguments. Note that *K*, *S*, *CP*, and *Hyp* all satisfy this definition trivially, so are theorem schemas.

— * —

In this section, and throughout this paper, we use symbols such as '$\vdash$', '$\iff$', '$\forall$', and '$\exists$' for expressing specifications and assertions about our functional programs. These symbols are *not* constructors of objects of type *Wff*—as one can see by looking at the definition of *Wff*— but are part of our specification language, which we do not define formally. However, *gp* and *just* are functional programs, not predicates in the specification language, so, to be absolutely correct, we should write, for example,

$$(p\ \text{`}\mathit{just}\text{`}\ \ ws\ \ \texttt{:|-}\ w) = \mathit{True}$$

—where *True* is a constant of built-in Gofer type *Bool*— instead of

$$p\ \text{`}\mathit{just}\text{`}\ \ ws\ \ \texttt{:|-}\ w$$

though this verges on the pedantic. Clearly, if *gp* or *just* were not total, we would have to be much more careful.

4 The Deduction Theorem and inference schemas

As seen in the definition of *reflex* (and in Fig. 1) constructing proofs of properties of :=> is tedious. The unadulterated Hilbert logic is almost impossible to use in practice for proving propositions since the only primitive inference rule, *MP*, is not suited to "backward" (i.e., goal-directed) proof: when attempting to find a proof for hs :|- y one must invent a new wff x and find proofs justifying hs :|- x :=> y and hs :|- x in order to apply *MP*. The problem is that, in general, there is nothing about the form of y that indicates which x to choose. The first step in addressing this problem is to define a functional program corresponding to the following inference rule, called the Deduction Theorem, which simplifies the process of finding proofs considerably. The rule is usually stated as follows:

$$\frac{ws \text{ `}union\text{` } [h] \ \vdash \ w}{ws \ \vdash \ h \text{ :=> } w}$$

(Throughout this report, we use the horizontal bar and symbol '$\Longrightarrow$' interchangeably to denote metalogical implication. Note that '$\Longrightarrow$' has *nothing* to do with :=> and objects of type *Wff*.) We will examine a stronger statement:

$$\frac{ws \ \vdash \ w}{ws \text{ `}without\text{` } h \ \vdash \ h \text{ :=> } w}$$

which is justified constructively by writing a total function *ded* that satisfies the following specification:

$$ded : Wff \rightarrow Prf \rightarrow Prf$$
$$gp\, p \quad \Longrightarrow \quad ded\, h\, p \ \text{ `}just\text{` } \ (hyps\, p \text{ `}without\text{` } h \ \text{ :|- } \ h \text{ :=> } con\, p)$$

Given the task of constructing a proof of h:=> w, this specification shows that it is sufficient to construct a proof of w using h as an added hypothesis and then apply *ded* — in general, a far easier task. As an example, the problem of constructing a proof of

```
|- x => ((x => y) => x)
```

(written in the same concrete syntax as described in the previous section) is reduced to constructing a simpler 3 line proof (see Fig. 3) followed by two applications of *ded* (see Figs. 4 and 5), resulting in a 23 line proof.

The following implementation of *ded* can, by structural induction on its second argument, be proved to meet its specification (see Appendix E):

```
ded              ::  Wff -> Prf -> Prf
ded h (MP p q)   =   MP (ded h p) (MP (ded h q)
                                  (S h (con p) (rimp (con q))))
ded h (Hyp x)    =   reflex h,  if  h == x
ded h p          =   MP p (K (con p) h)
```

— * —

Function *ded* is an example of an *inference schema* — a value of type:

$$r : \underbrace{Wff \to \cdots \to}_{\text{zero or more}} \underbrace{Prf \to \cdots \to}_{\text{one or more}} Prf$$

whose specification is of the form:

$$\begin{array}{l} (\ \forall\, x_0, \ldots, x_n : Wff,\ p_0, \ldots, p_m : Prf \\ \quad \mid gp\ p_0 \wedge \cdots \wedge gp\ p_m \ \wedge\ G\ x_0 \ldots x_n\ p_0 \ldots p_m \\ \quad \bullet\, gp(r\ x_0 \ldots x_n\ p_0 \ldots p_m) \\ ) \end{array}$$

where

$$G : \underbrace{Wff \to \cdots \to}_{\text{zero or more}} \underbrace{Prf \to \cdots \to}_{\text{one or more}} Bool$$

is the *guard* of the schema. Notice that by these definitions, *MP* is an inference schema guarded by

$$G\ p\ q \quad = \quad isImp\ (con\ q) \quad \wedge \quad con\ p = limp\ (con\ q)$$

5 Consistency

Our present knowledge of the properties of '$\vdash$' is limited: though we have exhibited proofs of some specific sequents we have not yet developed a sufficient theory to address the question, "For an arbitrary sequent, does there exist a proof of it?" We answer in the affirmative if we can find a proof justifying the sequent but we have no way of answering to the negative — in general, our failure to find a proof does not preclude its existence. Though we have given names to the syntactic constructors —'`Not`' and '`:=>`'— that suggest proof properties of wffs built by them, we have not yet shown these properties hold. For example, we expect that for all wffs x, not both

$$hs \vdash x \qquad \text{and} \qquad hs \vdash \texttt{Not}\ x \qquad .$$

But this statement remains a conjecture because we cannot as yet show the unprovability of any sequent.

The solution is to provide a semantics for our syntactic constructions by defining a boolean algebra (as a model) and mappings (valuations) that send wffs to values of the model. We introduce the idea of a tautology and describe the Consistency Theorem, which is stated informally, "if a sequent is provable then it is a tautology". The contrapositive —"if a sequent isn't

```
p,q,r ::  Prf

p   =  MP (Hyp (Var "x")) (Hyp (Var "x" :=> Var "y"))
q   =  ded (Var "x" :=> Var "y") p
r   =  ded (Var "x") q
```

Figure 2. The definition of three proofs illustrating the use of *ded*. See Figs. 3, 4, and 5

```
0   x |- x              Hyp x
1   x => y |- x => y    Hyp x => y
2   x, x => y |- y      MP 0, 1
```

Figure 3. The output from `showPrf p`.

```
0   x |- x                                 Hyp x
1   |- x => ((x => y) => x)                K x, x => y
2   x |- (x => y) => x                     MP 0, 1
3   |- (x => y) => ((x => y) => (x => y))  K x => y, x => y
             :
             :
8   |- ((x => y) => (x => y))
         => (((x => y) => x) => ((x => y) => y))
                                           S x => y, x, y
9   |- ((x => y) => x) => ((x => y) => y)  MP 7, 8
10  x |- (x => y) => y                     MP 2, 9
```

Figure 4. The output from `showPrf q` in abridged form.

```
0   |- x => (x => x)                              K x, x
1   |- x => ((x => x) => x)                       K x, x => x
2   |- (x => ((x => x) => x))
       => ((x => (x => x)) => (x => x))          S x, x => x, x
3   |- (x => (x => x)) => (x => x)               MP 1, 2
             :
             :
20  |- (x => (((x => y) => x) => ((x => y) => y)))
        => ((x => ((x => y) => x)) => (x => ((x => y) => y)))
                          S x, (x => y) => x, (x => y) => y
21  |- (x => ((x => y) => x))
        => (x => ((x => y) => y))                MP 19, 20
22  |- x => ((x => y) => y)                       MP 10, 21
```

Figure 5. The output from `showPrf r` in abridged form.

a tautology then it isn't provable"— gives us a method of addressing the problems raised above. We prove the converse, called the Completeness Theorem, in a later section. Because of the close connections between valuations and substitutions we intertwine the discussion of the two.

— * —

We consider a two-valued boolean lattice as a model of our propositional logic:

```
data BL = TT | FF
```

Two functions provide the basic operations:

```
imp         ::  BL -> BL -> BL
FF `imp` _   =  TT
TT `imp` x   =  x

neg         ::  BL -> BL
neg TT       =  FF
neg FF       =  TT
```

(As promised in section 1, the type discipline distinguishes :=> from `` `imp` `` :

$$\begin{array}{lcl} (:=>) & : & \mathit{Wff} \to \mathit{Wff} \to \mathit{Wff} \\ (\ \text{`}\mathit{imp}\text{`}\ ) & : & \mathit{BL} \to \mathit{BL} \to \mathit{BL} \end{array}$$

These two functions could even be overloaded with the same name —as logicians often do— without resulting in any confusion to the student because the two possible interpretations have distinct types. There would be no confusion to Gofer either if we defined a type class that contained both interpretations as instances.)

— * —

An environment is a list associating variable names with values of some type:

```
type Env a = [(String, a)]
```

Function *dom* calculates the domain of an environment and *lookup* finds the value associated with a particular name:

```
dom         ::  Env a -> [String]
dom          =  nub . map fst

lookup      ::  Env a -> String -> a
lookup e s   =  head [x | (s',x) <- e, s' == s]
```

Function *eval* evaluates a wff in a given environment:

```
eval               ::  Env BL -> Wff -> BL
eval v (x :=> y)    =  eval v x  `imp`  eval v y
eval v (Not x)      =  neg (eval v x)
eval v T            =  TT
eval v F            =  FF
eval v (Var s)      =  lookup v s
```

Function *sub* performs a (simultaneous) substitution on a wff:

```
sub              ::  Env Wff -> Wff -> Wff
sub e (x :=> y)   =  sub e x  :=>  sub e y
sub e (Not x)     =  Not (sub e x)
sub e T           =  T
sub e F           =  F
sub e (Var s)     =  lookup e s , if s `elem` dom e
                  =  Var s      , otherwise
```

Function *syntax* maps an environment of booleans to an environment of wffs:

```
syntax    ::  Env BL -> Env Wff
syntax e  =   [(s, syn x) | (s,x) <- e]

syn       ::  BL -> Wff
syn TT     =  T
syn FF     =  F
```

We say that v : *Env BL* is a *valuation environment* for x and *eval* v a *valuation* for x iff

$$vars\ x \quad \text{`}subset\text{`} \quad dom\ v.$$

If for every valuation *eval* v for x,

$$eval\ v\ x \quad = \quad TT \qquad ,$$

then x is a *tautology*, written $\models x$. Sequent hs `:|-` x is a tautology — written $hs \models x$— iff every valuation maps the conclusion to TT whenever it maps all the hypotheses to TT:

$$foldr\ imp\ (eval\ v\ x)\ (map\ (eval\ v)\ hs) \quad = \quad TT \qquad .$$

The reader may verify by structural induction on x that for each valuation environment v for x:

$$\models\ sub\ v'\ X \qquad \text{iff} \qquad eval\ v\ X\ =\ TT \tag{2}$$

where

$$v'\ =\ syntax\ v \qquad .$$

— $*$ —

The Consistency Theorem of the logic is stated as follows:

$$\frac{hs \vdash\ x}{hs \models x}$$

The result is a consequence of two observations about good proofs: the sequent of each leaf (axiom) is a tautology; if the sequents of two subproofs are tautologies then the sequent of the Modus Ponens of the subproofs is also a tautology. By induction, the sequent of any good proof is a tautology.

As an immediate corollary, for any wff x, not both

$$hs \vdash\ x \qquad \text{and} \qquad hs \vdash \texttt{Not}\ x$$

because by the definition of *eval* they cannot both be tautologies.

6 Tactics and tacticals

In the next two sections we justify the Completeness Theorem by building an automatic theorem-prover (a function mapping sequents to proofs) that terminates with a proof of its argument when presented with tautological sequent and with error message otherwise.

The prover is based on the strategy of goal-directed simplification: to find a proof for a goal (a sequent), find a proof for simpler subgoals and then apply an inference rule to proofs of the subgoals that yields a proof of the goal — where the choice of the subgoals and the inference rule is driven by the shape of the goal. (We did this by hand in Figs. 3, 4, and 5.) Such a strategy is encoded formally by a *tactic*, a function that maps a goal (i.e., a sequent to be proved) to a pair containing a list of subgoals (hopefully simpler than the original goal) and a validation. A validation maps proofs of the subgoals (if such proofs are ever discovered) to a proof of the original goal. (Tactics were first described this way by the research group working on LCF, [Gordon79]. The general use of tacticals is outlined in [Milner84].)

There are two important problems to solve in designing our prover. First, we need to be able to take descriptions of inference rules and generate primitive tactics. For example, consider the following description of *ded*

$$\frac{hs\ \text{`}union\text{`}\ [x] \vdash y}{hs \vdash x :\Rightarrow y}\ ded\ x$$

(This is a "description" in that *ded x* is the constructive justification of the rule described via the horizontal line.) We can create a tactic that checks if the goal is of the form hs :|- x :=> y, and if it is, generates

$$hs\ \text{`}union\text{`}\ [x]\ \texttt{:|-}\ yz$$

as the subgoal and inference rule *ded x* as the validation. Function *mktac*, which is described later, handles this translation automatically: it maps a description of an inference rule and and its implementation to the appropriate tactic.

Second, we need to be able to combine primitive tactics via alternation, sequential composition, and repetition. These combinators are known as *tacticals*, [Gordon79].

We address both problems in this section: we define types for representing tactics, we describe the implementation of *mktac*, and we introduce the basic tacticals.

— * —

The types for tactics and validations are as follows:

```
type Tactic          = Goal -> Opt ([Goal],Validation)
data Opt a           = Ok a | Fail
type Validation      = [Prf] -> Prf
type ParamValidation = [Wff] -> Validation
type Goal            = Seq
```

Because a tactic may fail (i.e., not be able to simplify the goal) the range type for tactics is defined via the the exception type *Opt*.

There are two fundamental tactics: *idtac*, which always succeeds but does nothing, and *failtac*, which always fails:

```
idtac, failtac  ::  Tactic
idtac g          =  Ok ([g], (\[p] -> p))
failtac g        =  Fail
```

The fundamental tacticals have the following types:

```
alttac, thentac :: Tactic -> Tactic -> Tactic
reptac          :: Tactic -> Tactic
```

They are described informally as follows:

t_1 *'alttac'* t_2	tries to apply t_1 and, if it fails, applies t_2;
t_1 *'thentac'* t_2	applies t_1 and then applies t_2 to the resulting subgoals;
reptac t	repeatedly applies t zero or more times until it fails.

Here are the implementation of *'alttac'* and *reptac* (*'thentac'* is more complicated and is defined in Appendix A):

```
(t1 'alttac' t2) g  =  t1 g  ?  t2 g
(?)        ::  Opt a -> Opt a -> Opt a
Ok x ? _    =  Ok x
Fail ? y    =  y

reptac t = (t 'thentac' reptac t) 'alttac' idtac
```

The tacticals have many algebraic properties, for example: *'alttac'* has identity *failtac*; *'thentac'* has identity *idtac*; both are associative; the latter tactical distributes through the former; *reptac* is a least fixed-point operator; etc. Because *'alttac'* is associative with identity *failtac*, we define *alttacs* (for the alternation of a list of tactics) as a fold-reduction:

```
alttacs   ::  [Tactic] -> Tactic
alttacs    =  foldr alttac failtac
```

Function *usetac* takes a tactic and a goal, applies the tactic, and, if no subgoals remain, returns the completed proof. If subgoals do remain or if the tactic failed then *usetac* causes an error:

```
usetac     ::  Tactic -> Goal -> Prf
usetac t g  =  f []
               where Ok ([],f) = t g
```

— * —

Function *mktac* maps four arguments to a tactic. The first argument is a sequent (written as a string in the concrete syntax described in section 3) with which the tactic tries to match the goal; i.e., the tactic tries to find a substition under which the sequent has the same conclusion as the goal and a subset of the hypotheses of the goal. The second argument is a semicolon-separated list of sequents which the tactic instantiates with the substitution environment to form the subgoals. The third argument is the validation. The fourth argument is a comma-separated list of wffs: these are instantiated with the substition environment generated by the goal-matching and given to the validation as parameters. For example, the tactic corresponding to the Deduction Theorem is

```
mktac  "|- X => Y"  "X |- Y"  ci  "X"
```

where *ci* is the parametrized validation:

```
ci          ::  ParamValidation
ci [x] [p]   =  ded x p
```

Compare the above tactic to the following description of *ci* (essentially the same function as *ded*) which appears with several other inference rules in Fig. 6:

$$\frac{X \vdash Y}{\vdash X \Rightarrow Y}\ ci\,[X]$$

The tactic could just as easily be coded by hand because the only matching done by the tactic is in the conclusion of the goal. However, in more complicated cases —for example in the following fragment

```
mktac "X, ~X |- Y" ...
```

which is intricate to code by hand— *mktac* saves us from worrying about many of the details.

$$\frac{X \vdash Y}{\vdash X \mathrel{:=>} Y}\; ci\,[X] \qquad \frac{\texttt{Not}\, X \vdash Z \qquad Y \vdash Z}{X \mathrel{:=>} Y \vdash Z}\; hi\,[X, Y]$$

$$\frac{\vdash X \qquad \vdash \texttt{Not}\, Y}{\vdash \texttt{Not}\ (X \mathrel{:=>} Y)}\; cni\,[\,] \qquad \frac{X, \texttt{Not}\, Y \vdash Z}{\texttt{Not}\ (X \mathrel{:=>} Y) \vdash Z}\; hni\,[X, Y]$$

$$\frac{\vdash X}{\vdash \texttt{Not}\ (\texttt{Not}\ X)}\; cnn\,[\,] \qquad \frac{X \vdash Y}{\texttt{Not}\ (\texttt{Not}\ X) \vdash Y}\; hnn\,[X]$$

Figure 6. Six inference rules used by the prover. (Two notational liberties have been taken to avoid cluttering the above: square brackets around hypotheses have been omitted and sequents have not been decorated with added hypotheses to the left of the $\vdash$.)

The implementations of the tacticals are similar to those in LCF and related systems but the designs were written independently by the author.

7 Completeness

Here we sketch our implementation of an automatic theorem prover defined via a tactic that repeatedly applies certain inference rules. The prover's domain is exactly the set of tautological sequents, so the prover justifies constructively the Completeness Theorem of the Hilbert logic:

$$\frac{hs \models x}{hs \vdash x}$$

Furthermore, the prover is a decision procedure because it terminates with an error when given a non-tautological sequent.

— * —

The rules in Fig. 6 induce the following tactics:

```
tacCI, tacCNN, tacCNI, tacHNN, tacHI, tacHNI  ::  Tactic
tacCI   = mktac "|- X => Y"       "X |-  Y"          ci  "X"
tacCNN  = mktac "|- ~~X"          "|- X"             cnn ""
```

$$\frac{}{X, \texttt{Not}\ X \vdash Y}\ ctrad\,[X,Y] \qquad \frac{}{X \vdash X}\ hyp\,[X]$$

$$\frac{}{\vdash T}\ ct\,[\,] \qquad \frac{}{\vdash \texttt{Not}\ F}\ cnf\,[\,]$$

$$\frac{}{F \vdash X}\ hf\,[X] \qquad \frac{}{\texttt{Not}\ T \vdash X}\ hnt\,[X]$$

Figure 7. Six theorem schemas used by the prover.

```
sizeSeq           :: Seq -> Int
sizeSeq (hs :|- x) = sum (map g hs) + g x
                     where g (y :=> z) = 2 + g y + g z
                           g (Not y)   = 1 + g y
                           g _         = 1
```

Figure 8. The definition of *sizeSeq*, a variant diminished by the application of each tactic.

```
tacCNI  = mktac "|- ~(X => Y)"     "|- X;  |- ~Y"      cni ""
tacHNN  = mktac "~~X |-  Z"        "X |- Z"            hnn "X"
tacHI   = mktac "X => Y |-  Z"     "~X |- Z;  Y |- Z"  hi  "X,Y"
tacHNI  = mktac "~(X => Y) |- Z"  "X, ~Y |- Z"        hni "X,Y"
```

These tactics have three significant properties. First, each generates subgoals that are strictly smaller (in the sense of the variant function *sizeSeq*, see Fig. 8) than the original goal. Second, each is validity preserving (the invariant property): if the goal is a tautology, all the subgoals are too. Third, any sequent whose hypotheses and conclusion are not all atomic (i.e., variables or constants and their negations) is simplified by at least one of the tactics.

Hence repeated application of these tactics to a tautological sequent terminates with a set of subgoals, each of which is a tautology and composed wholly of atomic wffs — so (as the reader can check) in the form of one of the theorem schemas shown in Fig. 7. These schemas induce tactics with empty lists of subgoals:

```
tacCTRAD, tacHYP, tacCT, tacCNF, tacHF, tacHNT :: Tactic
tacCTRAD = mktac "X, ~X |- Y"  ""  ctrad "X,Y"
tacHYP   = mktac "X |- X"      ""  hyp   "X"
tacCT    = mktac "|- T"        ""  ct    ""
tacCNF   = mktac "|- ~F"       ""  cnf   ""
tacHF    = mktac "F |- X"      ""  hf    "X"
tacHNT   = mktac "~T |- X"     ""  hnt   "X"
```

Tactic *tacprove* applies repeatedly the tactics described above:

```
tacprove = reptac (alttacs [                 -- ‘theorem’ tactics:
                            tacCTRAD, tacHYP,
                            tacCT,    tacCNF,
                            tacHF,    tacHNT,
                                             -- ‘inference’ tactics:
                            tacCI,    tacCNN, tacCNI,
                            tacHI,    tacHNN, tacHNI])
```

The prover *prove* employs *tacprove* to generate a proof for any tautological sequent. The specification of *prove* is the following

$$hs \models x \quad \Longrightarrow \quad \mathit{prove}\ (hs\ \texttt{:|-}\ x)\ \ \text{`}\mathit{just}\text{`}\ \ hs\ \texttt{:|-}\ x$$

which is implemented by

```
prove :: Seq -> Prf
prove  = usetac tacprove
```

Fig. 9 contains the traces of the inference rules used in two sample proofs generated automatically by *prove*. As a further illustration of just how difficult it is to use the Hilbert logic in practice, Fig. 10 shows the tail of the 3286 line Hilbert proof generated by *prove* of the same result justified by the three line proof (presented second) in Fig. 9.

— * —

We now address the problem of showing that the prover terminates in failure when given a non-tautological sequent. The implementations of the rules (validations) described in Figs. 6 and 7 (see Appendix B) meet their specifications, so the tactics employing them are *valid* [Gordon79]: for each tactic any list of proofs justifying the subgoals is transformed by the validation into a proof justifying the goal. Hence, if the theorem prover is applied to a sequent and terminates with a proof, the proof justifies the sequent. By the variant-function argument given earlier *tacprove* terminates when applied to any sequent. By the Consistency Theorem, a non-tautological sequent is unprovable. Hence, when the prover is applied to a non-tautological sequent, *tacprove* yields *Fail* and *prove* terminates with an error. Thus the prover is a decision procedure, as claimed.

Incidentally, the tactics used by *tacprove* are *strongly valid* [Gordon79]: for each tactic whenever the goal is provable and the tactic succeeds, the subgoals are provable too. This holds trivially for the tactics corresponding to the rules in Fig. 7 because each has an empty list of subgoals. For the tactics corresponding to the rules in Fig. 6, consider the following argument. By Consistency, if a goal is provable it is a tautology. By the "invariant" property of the tactics (see above), if one of the tactics is successfully applied to the tautological goal the subgoals are tautologies as well. By Completeness, the subgoals are provable.

— * —

The approach to Completeness taken here was discovered independently by the author. It is very similar to Wang's Algorithm [McCarthy65] and Semantic Tableaux analysis [Ben-Ari93].

8 Normal forms for uniform inference rules

The implementations of the inference rules used by the prover satisfy a common property: almost every one uses *ded* but none examines the "history" of its arguments proofs (i.e., none is defined recursively over the structure of its argument proofs) *except implicitly via ded*. We now show as a corollary to

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{Not}\ y,\ \mathtt{Not}\ x,\ \mathtt{Not}\ (\mathtt{Not}\ x)\ \vdash\ z}\,\mathit{ctrad}\,[\mathtt{Not}\ x,\ z] \qquad \dfrac{}{\mathtt{Not}\ y,\ \mathtt{Not}\ x,\ y\ \vdash\ z}\,\mathit{ctrad}\,[y,\ z]}{\mathtt{Not}\ y,\ \mathtt{Not}\ x,\ \mathtt{Not}\ x\ \mathtt{:=>}\ y\ \vdash\ z}\,\mathit{hi}\,[\mathtt{Not}\ x,\ y]}{\mathtt{Not}\ y,\ \mathtt{Not}\ x,\ \mathtt{Not}\ (\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y))\ \vdash\ z}\,\mathit{hnn}\,[\mathtt{Not}\ x\ \mathtt{:=>}\ y] \qquad \dfrac{}{\mathtt{Not}\ y,\ \mathtt{Not}\ x,\ z\ \vdash\ z}\,\mathit{hyp}\,[z]}{\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y)\ \mathtt{:=>}\ z,\ \mathtt{Not}\ x,\ \mathtt{Not}\ y\ \vdash\ z}\,\mathit{hi}\,[\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y),\ z]}{\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y)\ \mathtt{:=>}\ z,\ \mathtt{Not}\ x\ \vdash\ \mathtt{Not}\ y\ \mathtt{:=>}\ z}\,\mathit{ci}\,[\mathtt{Not}\ y]}{\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y)\ \mathtt{:=>}\ z\ \vdash\ \mathtt{Not}\ x\ \mathtt{:=>}\ (\mathtt{Not}\ y\ \mathtt{:=>}\ z)}\,\mathit{ci}\,[\mathtt{Not}\ x]}{\vdash\ (\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y)\ \mathtt{:=>}\ z)\ \mathtt{:=>}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ (\mathtt{Not}\ y\ \mathtt{:=>}\ z))}\,\mathit{ci}\,[\mathtt{Not}\ (\mathtt{Not}\ x\ \mathtt{:=>}\ y)\ \mathtt{:=>}\ z]$$

$$\dfrac{\dfrac{\dfrac{}{x,\ \mathtt{Not}\ (\mathtt{Not}\ y)\ \vdash\ x}\,\mathit{hyp}\,[x]}{\mathtt{Not}\ (x\ \mathtt{:=>}\ \mathtt{Not}\ y)\ \vdash\ x}\,\mathit{hni}\,[x,\ \mathtt{Not}\ y]}{\vdash\ \mathtt{Not}\ (x\ \mathtt{:=>}\ \mathtt{Not}\ y)\ \mathtt{:=>}\ x}\,\mathit{ci}\,[\mathtt{Not}\ (x\ \mathtt{:=>}\ \mathtt{Not}\ y)]$$

Figure 9. Traces (generated mechanically) of the inference rules used by *prove* in two sample proofs.

```
:
:
3279 |- ((~x => ~~(x => ~y)) => (~(x => ~y) => x))
        => (~(x => ~y) => ((~x => ~~(x => ~y)) => (~(x => ~y) => x)))
                      K (~x => ~~(x => ~y)) => (~(x => ~y) => x), ~(x => ~y)
3280 |- ~(x => ~y) => ((~x => ~~(x => ~y)) => (~(x => ~y) => x))
                                                             MP 3278, 3279
3281 |- (~(x => ~y) => ((~x => ~~(x => ~y)) => (~(x => ~y) => x)))
        => ((~(x => ~y) => (~x => ~~(x => ~y)))
           => (~(x => ~y) => (~(x => ~y) => x)))
                            S ~(x => ~y), ~x => ~~(x => ~y), ~(x => ~y) => x
3282 |- (~(x => ~y) => (~x => ~~(x => ~y)))
        => (~(x => ~y) => (~(x => ~y) => x))                 MP 3280, 3281
3283 |- ~(x => ~y) => (~(x => ~y) => x)                      MP 3277, 3282
3284 |- (~(x => ~y) => (~(x => ~y) => x))
        => ((~(x => ~y) => ~(x => ~y)) => (~(x => ~y) => x))
                                              S ~(x => ~y), ~(x => ~y), x
3285 |- (~(x => ~y) => ~(x => ~y)) => (~(x => ~y) => x)      MP 3283, 3284
3286 |- ~(x => ~y) => x                                         MP 4, 3285
(11414655 reductions, 27941489 cells, 42 garbage collections)
```

Figure 10. The tail of a pure Hilbert proof of the same result justified by part two of Fig. 9

our Completeness result that this is the usual situation with uniform inference rules (rules that respect substitutions): any uniform inference rule has a normal form that does not examine the structure of its argument proofs except implicitly via *ded*. Thus if *ded* were a primitive proof constructor, any uniform inference rule has a normal form that examines only the sequents justified by the rule's argument proofs, not the proofs' histories.

We say that the inference rule

$$\frac{\vdash X}{\vdash Y}\, r$$

is uniform iff it holds under all substitution environments $e : \mathit{Env\ Wff}$:

$$\frac{\vdash X_e}{\vdash Y_e}$$

where we write X_e for $\mathit{sub}\ e\ X$ throughout this section to avoid excess notation.

First we claim: $\models X :=> Y$. Let $\mathit{eval}\, v$ be any valuation for X and Y and suppose $v' = \mathit{syntax}\ v$:

$$\begin{array}{lll}
 & \mathit{eval}\, v\, X \;=\; TT & \\
\Longrightarrow & & \text{(2) in section 5} \\
 & \models\ X_{v'} & \\
\Longrightarrow & & \text{Completeness} \\
 & \vdash\ X_{v'} & \\
\Longrightarrow & & \text{rule } r \\
 & \vdash\ Y_{v'} & \\
\Longrightarrow & & \text{Consistency} \\
 & \models\ Y_{v'} & \\
\Longrightarrow & & \text{(2) in section 5} \\
 & \mathit{eval}\, v\, Y \;=\; TT &
\end{array}$$

(The above argument was suggested by Prof. Hoare.) Hence

$$\mathit{eval}\, v\, X\ \text{‘}\mathit{imp}\text{‘}\ \mathit{eval}\, v\, Y \quad = \quad TT$$

so $\models X :=> Y$ as claimed. By structural induction, $\models X_e :=> Y_e$, so

$$\dfrac{\vdash\ X_e \qquad \dfrac{\models X_e :=> Y_e}{\vdash\ X_e :=> Y_e}\,\text{Completeness}}{\vdash\ Y_e}\, \mathit{MP}$$

Writing the above as a program provides us with a definition of f, the normal form implementation of r:

```
f     :: Env Wff -> Prf -> Prf
f e p = MP p (prove ([] :|- sub e (X :=> Y)))
```

(In previous definitions of inference rules the environment was represented implicitly by the specific parameters passed and the sequent structure of the proof arguments. For the sake of generality, the substitution environment is used explicitly in f.)

Function *prove* generates (see the previous section) a composition of inference rules *ci*, *cni*, etc. —see Fig. 6— which examine the history of their argument proofs only implicitly via *ded*. Thus if *ded* were made a primitive proof constructor we could, as is done in LCF, represent proofs by their sequents alone —so the history of their construction would not be visible— without restricting the set of uniform inference rules that can be implemented.

This result extends easily to inference rules with sequents containing non-empty lists of hypotheses by using *ded* and *unded* —see Appendix B— to transform each sequent to a hypotheses-free form and back. The result also extends to inference rules containing multiple sequents "above the horizontal line" by combining these sequents via the conjunction rule:

$$\frac{ws \vdash x \qquad hs \vdash y}{ws \text{ `}union\text{`}\ hs \vdash x \text{ `}conj\text{`}\ y}$$

and its converse, where,

```
conj        ::  Wff -> Wff -> Wff
x `conj` y   =  Not (x :=> Not y)
```

9 The LCF Approach

The approach to logic outlined here differs from that taken by Robin Milner *et al.* in the design of LCF [Gordon79]. In LCF and its successors, wffs are represented by the abstract data type *Form* and theorems by *Thm*. The latter's constructors are the axioms and inference rules of the object logic. Proofs are not represented as data, but as programs that construct values of type *Thm*. Axiom schemas are functions of type $[Form] \rightarrow Thm$, and proof rules are functions of type $[Thm] \rightarrow Thm$. In LCF, the primitive *Thm* constructors can only generate sound theorems (each causing an exception if used incorrectly) with the advantage that if an object has type *Thm*, that is witness to its soundness. In our system, the soundness of a proof must be checked by *gp*. However, the burden of justifying that a derived inference rule meets its specification is equally hard in both system: in ours one verifies that the rule generates objects satisfying *gp* and in LCF one verifies that the rule calls each primitive (or derived) constructor correctly so as not to cause an exception.

In contrast to our explicit representation of a proof tree, LCF style systems represent the tree implicitly in the data used to implement the control structures of the metalanguage. Meta-level reasoning in our framework is conducted by induction over the structure of proofs, whereas in the LCF framework it is conducted by induction on the computation that results in a theorem. Hence, recursion over the structure of a proof is not possible in LCF: an inference rule such as *ded* that is defined by case-analysis on the history of its argument proof cannot be written in LCF except as a primitive constructor.

In some of the successors of LCF, for example Isabelle [Paulson94], it is possible to formalize the rules of a logic in such a way that an explicit record of the proof tree is made as a theorem is constructed.

10 Concluding Remarks

The goal of this research project was to present the fundamental ideas of formal logic in the context of functional programming, with the notion of wffs and proofs as unparsed textual objects replaced by data-structure definitions, and metalogical arguments about the former replaced by functional programs that manipulate the latter. A consequence of our implementation is that students can experiment with and test the programs on the computer. As an example, this provides them with a concrete justification for the introduction of hypothetical proof techniques because they can see *directly* the effect of applying *ded* to a hypothetical proof in order to generate a pure Hilbert one — as is illustrated in Figs. 3, 4, and 5. So *ded* and the inference rules described in the section on Completeness form a "compiler" that maps enriched natural-deduction style proofs to pure *Prf*-style ones. The correctness of the compiler follows from the correctness of the implementations of each of the inference rules. The task of designing the implementations of the inference rules may be given to students, who can build the functional programs that accomplish the "compiling": They can have the fun of seeing the compiler translate a high-level proof to a lower level one and then of comparing the lengths of the two.

In a highly unscientific study, some of the programs described in the earlier sections of this project were given to three friends to try on the computer. Their reaction was enthusiastic; one remarked, "I hadn't known that I hadn't understood the Deduction Theorem. Now I know I hadn't. And now I *do* understand it." The consensus was that the most important aspect in clarifying the meaning of the Deduction Theorem was that the functional notation made explicit the type declaration of *ded* —which describes the "interface" to the Deduction Theorem— and the type definitions of its arguments —which dictate the induction principle used in the Deduction Theorem.

— * —

This project is by no means a finished piece of work and there are limitations to the current implementation and many areas of possible improvement. The first main limitation is that the data-structures support a propositional, not predicate, logic. Extending the structures to include support for basic notions in the predicate calculus is not difficult and may be enlightening, e.g., a function for checking that a term is "safe for substitution" for a variable in a wff is a simple recursion over the structure of wffs. However, supporting the ability to add arbitrary new axiomatizations of theories to the logic (for example, the Peano theory of natural numbers) is difficult. LCF and its successors are built to do this but their source code is several thousand lines of ML employing a combination of functional and imperative programming styles; code of this complexity may not be the best way of introducing metalogical notions to students learning logic for the first time.

Our representation of proofs —where the entire history of a proof is passed to each inference rule— is not a realistic choice because proofs of even trivial theorems contain many thousands of nodes. However the Deduction Theorem cannot be written as an explicit recursion over the structure of proofs unless their histories are preserved, and as our goal was to present the functional programs for their pedagogical value, not their efficiency, the representation we decided on seemed to be the best choice.

Our implementations also suffer from certain unavoidable constraints inherent in the finite nature of the computer: we cannot write terminating programs that provide constructive justifications for metatheorems such as Compactness or such as Completeness when done via the theory of maximally consistent sets.

Despite the limitations outlined above, the work in this project has led us to believe that computing science —and, in particular, functional programming— can offer its students new ways of learning and mastering formal logic. All the program scripts described in this report, along with additional material, are available in [Leifer95] and we encourage you to experiment with and enhance our results.

11 Acknowledgements

I gratefully acknowledge the guidance provided by my joint supervisors. Bernard Sufrin supported my explorations of the different possible representations of proofs and inference rules in a functional language and debated with me the merits and demerits of various designs. Professor Hoare read through numerous drafts of this project and proposed ideas for improving greatly the structure and the clarity of the presentation. Without their advice and inspiration this project could not have been done.

I wish to thank the following for their detailed comments and criticisms on the sections of this project included in an earlier paper: R.S. Bird, E.W. Dijkstra (in conjunction with the Austin Tuesday Afternoon Club), D. Gries, J.M. Spivey, and P. Wadler.

The program for maintaining the consistency between the typeset and executable versions of the Gofer code was part of Bernard Sufrin's Antetex package. All the inference rules and proofs employing the "horizontal bar" notation were typeset with macros from the `proof.sty` package developed by Makoto Tatsuta.

References

[Ben-Ari93] M. Ben-Ari. *Mathematical Logic for Computer Science.* (Prentice Hall, 1993).

[Bornat93] R. Bornat and B.A. Sufrin. *The Gist of Jape.* (Draft of 25 November 1993.) Available via anonymous ftp from `ftp.comlab.ox.ac.uk` as `pub/Packages/JAPE/doc.ps`.

[Gordon79] M.J.C. Gordon, A.J.R.G. Milner, and C.P. Wadsworth. *Edinburgh LCF.* (Springer LNCS #78, 1979).

[Hamilton93] A.G. Hamilton. *Logic for Mathematicians.* (CUP, 1993).

[Jones91] M.P. Jones. *An Introduction to Gofer.* Available, with compiler source code, via anonymous ftp from `nebula.cs.yale.edu`.

[Leifer95] J.J. Leifer. "Source listings for *Formal Logic via Functional Programming*". Available via anonymous ftp from `ftp.comlab.ox.ac.uk` in `pub/Packages/JAPE/LogicviaFP/`.

[McCarthy65] J. McCarthy *et al. LISP 1.5 Programmer's Manual.* Second Edition. (MIT Press, 1965).

[Milner84] A.J.R.G. Milner. "The use of machines to assist in rigorous proof." *Phil. Trans., Royal Soc. London*, A **312**, pp. 411-422.

[Paulson94] L.C. Paulson. *Introduction to Isabelle.* (Computer Laboratory, Cambridge University, 1994).

A Tacticals and *mktac*

We implement the basic tacticals as well as the sequent-matching functions with which *mktac* is defined. See section 6 for the documentation.

```
{- isOK checks if its argument is an exception -}

isOk         :: Opt a -> Bool
isOk Fail     = False
isOk (Ok _)   = True

isFail :: Opt a -> Bool
isFail  = not . isOk


unOk (Ok x) = x


{- reptac is the repetition tactical. For efficiency, it is
defined as a cyclic structure -}

reptac  :: Tactic -> Tactic
reptac t = r
           where r = (t `thentac` r) `alttac` idtac


{- thentac is the sequential composition tactical. -}

thentac          ::  Tactic -> Tactic -> Tactic
(t1 `thentac` t2) g  =  Ok (gs,f) , if isOk ng1 && ng2Ok
                     =  Fail      , otherwise
                        where ng1        =  t1 g
                              (gs1, f1)  =  unOk ng1
                              ng2        =  map t2 gs1
                              ng2Ok      =  all isOk ng2
                              gs2        =  map (fst . unOk) ng2
                              fs         =  map (snd . unOk) ng2
                              gs         =  concat gs2
                              ls         =  map length gs2
                              f          =  f1 . mappart ls fs


{- part takes a list of ints and a list of arbitrary type and
partitions the latter into sublists of sizes specified in the
former.  -}
```

```
part          :: [Int] -> [a] -> [[a]]
part [] []     = []
part (n:ns) xs = take n xs : part ns (drop n xs)

mappart         :: [Int] -> [[a] -> b]  -> [a] -> [b]
mappart ns fs xs = zipWith ($) fs (part ns xs)

wrap x     =  [x]
unwrap [x] =  x


{- mktac parses and normalizes its arguments and passes them to
mktac' -}

mktac :: String -> String -> ParamValidation -> String -> Tactic
mktac g gs f xs = mktac' (nubSeq (parSeq g))
                         (map nubSeq (parSeqs gs))
                         f
                         (parWffs xs)


{- mktac' takes the description of an inference rule and generates
the corresponding tactic -}

mktac' :: Goal -> [Goal] -> ParamValidation -> [Wff] -> Tactic
mktac' g gs f xs g' = Ok (gs', f xs'), if isOk e
                    = Fail           , otherwise
                      where e        = matchSeq g g'
                            (e',hs)  = unOk e
                            gs'      = map (addhyps ys . subS e') gs
                            xs'      = map (sub e') xs
                            zs :|- _ = g'
                            ys       = zs `withoutlist` hs

{- matchWff finds a substitution environment to match its first
argument to its second -}

matchWff                    :: Wff -> Wff -> Opt (Env Wff)
matchWff (Var s)   y         = Ok [(s,y)]
matchWff (w :=> x) (y :=> z) = matchWff w y `joinEnv` matchWff x z
matchWff (Not x)   (Not y)   = matchWff x y
matchWff _         _         = Fail


{- matchSeq s1 s2 tries to find a substitution environment for
matching s1 's conclusion to s2 's conclusion and s1 's
```

```
hypotheses to a subset of s2' s hypotheses.  If successful, it
returns the environment and the list of the hypotheses in s2
leftover.  -}

matchSeq :: Seq -> Seq -> Opt (Env Wff, [Wff])
matchSeq (ws :|- x) (hs :|- y)
         =  Ok (head ms), if not (null ms)
         =  Fail        , otherwise
            where e' = matchWff x y
                  ms = [(unOk e,xs)
                       | (e'',xs) <- matchmatch ws hs,
                         e        =  e' `joinEnv` Ok e'',
                         isOk e]


{- matches takes a wff and a list of wffs and returns all possible
ways of matching the wff to an element of the list and, in each
case, the leftover wffs not matched. -}

matches     :: Wff -> [Wff] -> [(Env Wff, Wff)]
matches w hs = [(unOk e,h) | h <- hs, e = matchWff w h, isOk e]


{- matchmatch takes two lists of wffs and returns all possible ways
of matching the first list to a subset of the second and in each
case the leftover wffs not matched. -}

matchmatch          :: [Wff] -> [Wff] -> [(Env Wff, [Wff])]
matchmatch [] hs     = [([],[])]
matchmatch (w:ws) hs = [(unOk e'', h:hs')
                       | (e,h)    <- matches w hs,
                         (e',hs') <- matchmatch ws (hs `without` h),
                         e''      =  Ok e `joinEnv` Ok e',
                         isOk e'']


{-  joinEnv joins two compatible substitution environments -}

joinEnv Fail   _      = Fail
joinEnv _      Fail   = Fail
joinEnv (Ok v) (Ok e) = Fail            , if not (null xs)
                      = Ok (v `union` e), otherwise
                        where xs = [s
                                   |(s,x) <- v,
                                    s `elem` dom e
                                    && lookup e s /= x]
```

B Implementations of the inference rules

All the rules used by the theorem prover have the same type:

```
cni, cnn, hnn, hni, hi,
ct, cnf, hf, hnt, ctrad, hyp ::  ParamValidation
```

Here are their implementations. Each is preceeded by its description. Most of the following refer to functions defined later on in this appendix.

$$\frac{X \vdash Y}{\vdash X \mathrel{:=>} Y}\ ci\,[X]$$

```
ci [x] [p]  =  ded x p
```

$$\frac{\vdash X \qquad \vdash \texttt{Not}\,Y}{\vdash \texttt{Not}\,(X \mathrel{:=>} Y)}\ cni\,[\,]$$

```
cni [] [p,q]  =  MP q ((anticontrapos . ded xiy) (MP p (Hyp xiy)))
                 where (_ :|- x)     = seq p
                       (_ :|- Not y) = seq q
                       xiy           = x :=> y
```

$$\frac{\vdash X}{\vdash \texttt{Not}\,(\texttt{Not}\,X)}\ cnn\,[\,]$$

```
cnn [] [p]  =  MP p (impnotnot (con p))
```

$$\frac{X \vdash Y}{\texttt{Not}\,(\texttt{Not}\,X) \vdash Y}\ hnn\,[X]$$

```
hnn [x] [p]  =  MP (unded (notnotimp x)) (ded x p)
```

$$\frac{X, \texttt{Not}\, Y \ \vdash\ Z}{\texttt{Not}\ (X :=> Y) \ \vdash\ Z}\ hni\,[X, Y]$$

```
hni [x,y] [p]  =  (nottrade (con p) . ded x . nottrade y) p
```

$$\frac{\texttt{Not}\, X \ \vdash\ Z \qquad Y \ \vdash\ Z}{X :=> Y \ \vdash\ Z}\ hi\,[X, Y]$$

```
hi [x,y] [p,q]  =  nottrade z (cni [] [nottrade x p,
                                       anticptrade y q]), if z == z'
                   where (z, z') = (con p, con q)
```

$$\frac{}{X \ \vdash\ X}\ hyp\,[X]$$

```
hyp [x] []  =  Hyp x
```

$$\frac{}{X, \texttt{Not}\ X \ \vdash\ Y}\ ctrad\,[X, Y]$$

```
ctrad [x,y] []  =  nottrade y (Hyp x)
```

$$\frac{}{\vdash\ T}\ ct\,[\,]$$

```
ct [] []  =  MP (Top T) (Top (T :=> T))
```

$$\frac{}{\vdash \texttt{Not}\ F}\ cnf[\,]$$

```
cnf [] []  =  MP (cnn [] [ct [] []]) (anticontrapos (Bot (Not T)))
```

$$\frac{}{F\ \vdash\ X}\ hf[X]$$

```
hf [x] []  =  unded (Bot x)
```

$$\frac{}{\texttt{Not}\ T\ \vdash\ X}\ hnt\,[X]$$

```
hnt [x] []  =  nottrade x (ct [] [])
```

— * —

Here are the support functions referred to above:

$$\frac{\vdash\ X\ \texttt{:=>}\ Y}{X\ \vdash\ Y}\ unded$$

```
unded  :: Prf -> Prf
unded p = MP (Hyp (limp (con p))) p
```

$$\frac{\vdash\ \texttt{Not}\ X\ \texttt{:=>}\ \texttt{Not}\ Y}{\vdash\ Y\ \texttt{:=>}\ X}\ contrapos$$

```
contrapos   ::  Prf -> Prf
contrapos p  =  MP p (CP x y)
                where (_ :|- Not x :=> Not y) = seq p
```

$$\frac{X \ \vdash \ X \ \texttt{:=>} \ Y}{\vdash \ X \ \texttt{:=>} \ Y} \ \textit{dedantec}$$

```
dedantec    ::  Prf -> Prf
dedantec p   =  ded x (MP (Hyp x) p)
                where x = limp (con p)
```

$$\frac{}{\vdash \ \texttt{Not} \ (\texttt{Not} \ X) \ \texttt{:=>} \ X} \ \textit{notnotimp} \ X$$

```
notnotimp   :: Wff -> Prf
notnotimp x  = (dedantec . contrapos . contrapos)
                 (MP (Hyp nt2x) (K nt2x nt4x))
               where nt2x = Not (Not x)
                     nt4x = Not (Not nt2x)
```

$$\frac{}{\vdash \ X \ \texttt{:=>} \ \texttt{Not} \ (\texttt{Not} \ X)} \ \textit{impnotnot} \ X$$

```
impnotnot   :: Wff -> Prf
impnotnot x  = contrapos (notnotimp (Not x))
```

$$\frac{\texttt{Not} \ X \ \vdash \ Y}{\texttt{Not} \ Y \ \vdash \ X} \ \textit{nottrade} \ X$$

```
nottrade    :: Wff -> Prf -> Prf
nottrade x p = (unded . contrapos . ded (Not x) . cnn [] . wrap) p
```

$$\frac{X \vdash Y}{\texttt{Not}\ Y \vdash \texttt{Not}\ X}\ \textit{anticptrade}\ X$$

```
anticptrade    :: Wff -> Prf -> Prf
anticptrade x p = (unded . anticontrapos . ded x) p
```

$$\frac{\texttt{Not}\ Y \vdash \texttt{Not}\ X}{X \vdash Y}\ \textit{cptrade}\ Y$$

```
cptrade    :: Wff -> Prf -> Prf
cptrade x p = (hnn [rimp (con p)] . wrap . nottrade (Not x)) p
```

$$\frac{\vdash X \texttt{ :=> } Y}{\vdash \texttt{Not}\ Y \texttt{ :=> } \texttt{Not}\ X}\ \textit{anticontrapos}$$

```
anticontrapos  :: Prf -> Prf
anticontrapos p  =  (contrapos . ded nt2x . cnn [] . wrap
                     . hnn [x] . wrap . unded) p
                    where x    = limp (con p)
                          nt2x = Not (Not x)
```

C Parsing and pretty-printing

```
{- showWff pretty prints a wff -}
showWff x = sliceparens (showWff' x [])

showWff' (Var s) zs   = s ++ zs
showWff' (Not x) zs   = '~' : showWff' x zs
showWff' (x :=> y) zs = '(' : showWff' x (" => "
                         ++ showWff' y (')':zs))
showWff' T zs         = 'T':zs
showWff' F zs         = 'F':zs
```

```
sliceparens ""                    =  ""
sliceparens (x:xs) | x == '('   =  init xs
                   | otherwise  =  (x:xs)

showWffs  =  foldr f [] . map showWff
             where f xs []  =  xs ++ " "
                   f xs ys  =  xs ++ ", " ++ ys

showSeq (hs :|- w)  =  showWffs hs ++ "|- " ++ showWff w


{- numnodes counts the number of nodes in a proof -}
numnodes :: Prf -> Int
numnodes (MP p q) = numnodes p + numnodes q + 1
numnodes _        = 1

{- showPrf pretty prints a proof -}
showPrf p = fst (showPrf' p 0 [])

showPrf' p @ (MP q1 q2) n zs
   =  (sh1, s), if x == y
      where (sh1,s1) = showPrf' q1 n sh2
            (sh2,s2) = showPrf' q2 (n+n1)
                        (shPline p s (n+n1+n2) ++ show (n+n1-1)
                         ++ ", " ++ show (n+n1+n2-1)
                         ++ "\n" ++ zs)
            (n1,n2)  = (numnodes q1, numnodes q2)
            ws :|- x       = s1
            hs :|- y :=> z = s2
            s              = ws `union` hs :|- z

showPrf' p n zs
   =  (shPline p s n ++ "\n" ++ zs, s)
      where s = seq p

shPline p s n  =  shlinnum n ++ showSeq s ++ "    " ++ showrsn p


shlinnum ::  Int -> String
shlinnum  =  ljustify 4 . show

showrsn (K x y)    =  "K " ++ showWff x ++ ", " ++ showWff y
showrsn (S x y z)  =  "S " ++ showWff x ++ ", " ++ showWff y ++ ", "
                        ++ showWff z
showrsn (CP x y)   =  "CP " ++ showWff x ++ ", " ++ showWff y
showrsn (MP _ _)   =  "MP "
```

```
showrsn (Hyp x)    =  "Hyp " ++ showWff x
showrsn (Top x)    =  "Top " ++ showWff x
showrsn (Bot x)    =  "Top " ++ showWff x


{-
A parsing library.

   The following is lifted directly from the parser library of Mark
Jones's Mini-Prolog language written in Gofer (demos/prolog/parser
in the distribution) which, in turn, was taken from Richard Bird's
notes on parsers done in Orwell.  (I have lifted Jones'
explanations as well since they make the following
comprehensible.)  -}


infixr 6 ->>
infixr 4 |||
infixl 5 `do`

-- Type definition:

type Parser a = [Char] -> [(a,[Char])]

{- A parser is a function which maps an input stream of characters
into a list of pairs each containing a parsed value and the
remainder of the unused input stream.  This approach allows us to
use the list of successes technique to detect errors (i.e. empty
list ==> syntax error).  it also permits the use of ambiguous
grammars in which there may be more than one valid parse of an
input string.

Primitive parsers:

fail    is a parser which always fails.

okay v  is a parser which always succeeds without consuming any
        characters from the input string, with parsed value v.

tok w   is a parser which succeeds if the input stream begins with
        the string (token) w, returning the matching string and the
        following input.  If the input does not begin with w then
        the parser fails.

sat p   is a parser which succeeds with value c if c is the first
        input character and c satisfies the predicate p.
```

```
-}

fail         :: Parser a
fail is       = []

okay         :: a -> Parser a
okay v is     = [(v,is)]

tok          :: [Char] -> Parser [Char]
tok w is      = [(w, drop n is) | w == take n is]
                where n = length w

sat          :: (Char -> Bool) -> Parser Char
sat p []      = []
sat p (c:is) = [ (c,is) | p c ]

{-
Parser combinators:

p1 ||| p2      is a parser which returns all possible parses of the
               input string, first using the parser p1, then using
               parser p2.

p1 ->> p2      is a parser which returns pairs of values (v1,v2)
               where v1 is the result of parsing the input string
               using p1 and v2 is the result of parsing the
               remaining input using p2.

p `do` f       is a parser which behaves like the parser p, but
               returns the value f v wherever p would have returned
               the value v.

many p         returns a list of values, each parsed using the
               parser p.

many1 p        parses a non-empty list of values, each parsed
               using p.

listOf p s     parses a list of input values using the parser p,
               with separators parsed using the parser s.
-}

(|||)           ::  Parser a -> Parser a -> Parser a
(p1 ||| p2) is   =  p1 is ++ p2 is

(->>)           ::  Parser a -> Parser b -> Parser (a,b)
```

```
(p1 ->> p2) is   =  [((v1,v2),is2)
                     | (v1,is1) <- p1 is, (v2,is2) <- p2 is1]

do              ::  Parser a -> (a -> b) -> Parser b
(p `do` f) is    =  [(f v, is1) | (v,is1) <- p is]

many            ::  Parser a  -> Parser [a]
many p           =  q
                    where q = (p ->> q `do` makeList) ||| (okay [])

many1           ::  Parser a -> Parser [a]
many1 p          =  p ->> many p `do` makeList

listOf          ::  Parser a -> Parser b -> Parser [a]
listOf p s       =  p ->> many (s ->> p) `do` nonempty
                    ||| okay []
                    where nonempty (x,xs) = x:(map snd xs)

makeList        ::  (a,[a]) -> [a]
makeList (x,xs)  =  x:xs

{-
Parsing wffs and sequents.

Here is the EBNF style grammar for wffs:

pW   =  pW' "=>" pW'
      | pW'

pW'  =  literal
      | "~" pW'
      | "(" pW' ")"
      | "(" pW' "=>" pW' ")"

There is a special case in pW since "X => Y" should be a valid
string to parse, i.e., the outermost implication needn't be
enclosed in parentheses.  As a result, backtracking must be done to
resolve the two cases of pW, but, clearly, the grammar isn't
ambiguous. -}

parWff  =  fst . head . pW . filtSpaces

filtSpaces  =  filter ( /= ' ')

pW  =   pW' ->> tok "=>" ->> pW'         `do` getImp
    ||| pW'
```

```
pW' =   many1 (sat isAlpha)               `do` getident
    ||| tok "~"  ->>  pW'                 `do` getNot
    ||| tok "("  ->>  pW'  ->>  tok "=>"
                 ->>  pW'  ->>  tok ")"  `do` getparenImp
    ||| tok "("  ->>  pW'  ->>  tok ")"  `do` getparen

getNot (_,y)                    =  Not y
getident "T"                    =  T
getident "F"                    =  F
getident xs                     =  Var xs
getparenImp (_,(x,(_,(y,_))))  =  x :=> y
getImp (x, (_, y))              =  x :=> y
getparen (_,(x,_))              =  x


pWs     = listOf pW (tok ",")
parWffs = fst . head . pWs . filtSpaces

parSeq  =  fst . head . parSeq' . filtSpaces
parSeq'  =  pWs  ->>  tok "|-"  ->>  pW  `do` getSeq
getSeq (xs, (_, x))  =  xs :|- x

parSeqs  = fst . head . parSeqs'. filtSpaces
parSeqs' = listOf parSeq' (tok ";") `do` id
```

D Miscellaneous support functions

```
xs `union` ys       =  nub (xs ++ ys)
xs `subset` ys      =  all (`elem` ys) xs
xs `without` x      =  filter ( /= x) xs
xs `withoutlist` ys =  filter (`notElem` ys) xs

nubSeq (hs :|- x)    =  nub hs :|- x
addhyps hs (ws :|- x) = (ws `union` hs) :|- x

pprove :: String -> String
pprove = showPrf . prove . parSeq

{- subS performs a substitution on sequent -}
subS e (hs :|- x) = map (sub e) hs  :|-  sub e x

{- instance ... adds wffs to the equality type-class -}
instance Eq Wff where
   (w :=> x) == (y :=> z)  =  w == y && x == z
```

```
(Not x)   == (Not y)    =  x == y
(Var s)   == (Var t)    =  s == t
T         == T          =  True
F         == F          =  True
_         == _          =  False
```

E The Jape proof of correctness for *ded*

We present in the Jape language [Bornat93] an axiomatization of the *Wff* and *Prf* data-types and of *ded* and its associated functions. We then show a selection of Jape proofs used to verify the correctness *ded*.

```
/* logic.j  -- an axiomatization of wffs, prfs, and ded.

Because of certain syntactic conventions/restrictions of Jape,
the following list shows new names that are given to certain
objects in our axiomatization.  The rest remain the same:

  When we usually write ...      we now write ...
   :=>                              Imp
   []                               nil
   [x]                              {x}
   T                                Truth
   F                                Falsity
*/


/* The following are standard Jape modules that provide
support for logical reasoning, and equational rewriting: */

USE "aipc.j"
USE "eq.j"
USE "hci.j"


/* fixity declarations: */
CONSTANT nil

OUTFIX { }

INFIX  15000 15000 union
INFIX  10000 10000 without
INFIX  5000 5000 subset
INFIX  5000 5000 notin
INFIX  5000 5000 in
```

```
/* A simple theory of sets (to be used for handling
manipulations involving hypotheses: */

THEORY Sets (xs,ys,zs,x,y)

RULES union
ARE     nil union ys = ys
AND     ys union nil = ys

RULES notin
ARE     x notin nil
AND     FROM   ~(x=y) INFER  x notin {y}
AND     FROM   x notin xs AND x notin ys
        INFER  x notin  xs union ys

RULES in
ARE     FROM x=y INFER x in {y}
AND     FROM x in xs INFER x in xs union ys
AND     FROM x in ys INFER x in xs union ys

RULES subset
ARE     nil subset ys
AND     FROM x in ys INFER {x} subset ys
AND     FROM xs subset zs AND ys subset zs
        INFER xs union ys  subset  zs
AND     FROM xs subset ys INFER xs subset ys union zs
AND     FROM xs subset zs INFER xs subset ys union zs


RULES without
ARE     nil without y = nil
AND     FROM x=y INFER {x} without y = nil
AND     FROM ~(x=y) INFER {x} without y = {x}
AND     xs union ys without y
        = (xs without y) union (ys without y)

RULE  Setinduction(E, FRESH xs, FRESH ys,
                   FRESH x, FRESH xyzzy, P)
WHERE   VAR x, xs, ys
        AND FRESH x, xs, ys
        AND SAFE P[xyzzy\E], P[xyzzy\nil],
                 P[xyzzy\xs], P[xyzzy\ys]
        FROM P[xyzzy \ nil] AND P[xyzzy \ {x}]
        AND  P[xyzzy \ xs], P[xyzzy \ ys]
```

```
            |- P[xyzzy \ xs union ys]
        INFER |- P[xyzzy \ E]


MENU    ETC IS

        SEPARATOR

        TACTIC "Set Induction" IS
        /*     If an induction site is selected, then use it;
               if goal is extensionality, then use the variable
        */
               WHEN
               (   LETSEL(_v, Setinduction _v)
               ,   LETCONC(_F _v = _G _v, Setinduction _v)
               ,   FAIL "Specify an induction variable"
               )
END

END


/* Constants for representing wffs: */

CONSTANT
        Var, Not, Truth, Falsity

INFIX   20000 20000 Imp

MENU ETC

/* The induction principle for wffs: */

RULE    Wffinduct
        (E, FRESH s, FRESH w, FRESH w1, FRESH w2, FRESH z, P)
        WHERE   VAR xs,w,w1,w2,z
        AND     SAFE P[z\Lit s], P[z\w], P[z\not w],
                     P[z\w1], P[z\w2], P[z\E]
        AND     FRESH s,w,w1,s2,z
        FROM    P[z\Var s]
        AND     P[z\Falsity]
        AND     P[z\Truth]
        AND     P[z\w] |- P[z\Not w]
        AND     P[z\w1], P[z\w2] |- P[z\w1 Imp w2]
        INFER   P[z\E]
```

```
/* Constants for representing proofs: */

CONSTANT
        K, S, CP, Top, Bot, Hyp, MP

/* The induction principle for proofs: */

RULE    Prfinduct
        (F, FRESH w, FRESH w1, FRESH w2,
            FRESH w3, FRESH p1, FRESH p2, P)
        FROM |- P[z\K w1 w2]
        AND  |- P[z\S w1 w2 w3]
        AND  |- P[z\CP w1 w2]
        AND  |- P[z\Hyp w]
        AND  |- P[z\Top w]
        AND  |- P[z\Bot w]
        AND  P[z\p1], P[z\p2] |- P[z\MP p1 p2]
        INFER P[z\F]
END


/* Axiomatizations of the function programs that manipulate
wffs and proofs */

THEORY  Log(x, y, xs, ys, w, w1, w2, w3,
            wl, wr, p, p1, p2, q, r, h, C) IS

CONSTANT
        con, hyps, limp, rimp, isImp, reflex, gp

RULE    limp IS limp(wl Imp wr) = wl

RULE    rimp IS rimp(wl Imp wr) = wr

RULES con
ARE   con (K w1 w2)    = w1  Imp  (w2  Imp  w1)
AND   con (S w1 w2 w3) = (w1 Imp (w2 Imp w3))
                         Imp ((w1 Imp w2) Imp (w1 Imp w3))
AND   con (CP w1 w2)   = (Not w1 Imp Not w2)  Imp  (w2 Imp w1)
AND   con (Hyp w)      = w
AND   con (Top w)      = w Imp Truth
AND   con (Bot w)      = Falsity Imp w
AND   FROM  con p       = limp (con q)
      INFER con(MP p q) = rimp (con q)
```

```
RULES   hyps
ARE     hyps(K w1 w2)    = nil
AND     hyps(S w1 w2 w3) = nil
AND     hyps(CP w1 w2)   = nil
AND     hyps(Top w)      = nil
AND     hyps(Bot w)      = nil
AND     hyps(Hyp w)      = {w}
AND     hyps(MP p q)     = hyps p union hyps q

RULES gp
ARE   gp(K w1 w2)
AND   gp(S w1 w2 w3)
AND   gp(CP w1 w2)
AND   gp(Hyp w)
AND   gp(Top w)
AND   gp(Bot w)
AND   FROM   isImp(con q)
      AND    limp(con q) = con p
      AND    gp p
      AND    gp q
      INFER  gp(MP p q)
AND   FROM   gp p, gp q, isImp (con q),
                         limp(con q) = con p |- C
      INFER  gp(MP p q) |- C


/* The definition of reflex: */

RULE    reflex IS
        reflex w = MP (K w w) (MP (K w (w Imp w))
                                  (S w (w Imp w) w))


/* The definition of ded: */

INFIX   5000 5000 indep

RULES ded
ARE ded h (MP p q) = MP (ded h p)
                        (MP (ded h q)
                            (S h (con p) (rimp (con q))))
AND     FROM    x=h
        INFER   ded h (Hyp x) = reflex h

AND     FROM    p indep h
        INFER   ded h p  = MP p (K (con p) h)
```

```
RULES   indep
ARE     (K w1 w2)    indep h
AND     (S w1 w2 w3) indep h
AND     (CP w1 w2)   indep h
AND     (Top w)      indep h
AND     (Bot w)      indep h
AND     FROM  ~(x=h)
        INFER (Hyp x) indep h

RULES   isImp
ARE     isImp  (w1 Imp w2)
AND     isImp w |- limp w Imp rimp w = w

END


/* Case analysis rule: */

RULE   eqcases(x,h,P)
IS     FROM x=h |- P AND ~(x=h) |- P INFER |- P


/* reflex meets is specification: */

THMS reflexSpec'(w)
ARE   gp(reflex w)
AND   con(reflex w)   = w  Imp w
AND   hyps(reflex w)  = nil


/* ded meets is specification: */

THMS dedSpec(h,p,q)
ARE    gp p => (con(ded h p) = h Imp con p /\ gp(ded h p))
AND    gp p => hyps (ded h p)  subset  hyps p without h

/* The inductive cases involved in proving ded meets its
spec: */

THMS dedLemmas'(h,w,x,y)
ARE gp(K w1 w2)
    =>(con(ded $h(K w1 w2))=$h Imp con(K w1 w2)
      /\gp(ded $h(K w1 w2)))
```

```
AND gp(S w1 w2 w3)
    =>(con(ded $h(S w1 w2 w3))=$h Imp con(S w1 w2 w3)
      /\gp(ded $h(S w1 w2 w3)))

AND gp(CP w1 w2)
    =>(con(ded $h(CP w1 w2))=$h Imp con(CP w1 w2)
      /\gp(ded $h(CP w1 w2)))

AND gp(Top w)
    =>(con(ded $h(Top w))=$h Imp con(Top w)
      /\gp(ded $h(Top w)))

AND gp(Bot w)
    =>(con(ded $h(Bot w))=$h Imp con(Bot w)
      /\gp(ded $h(Bot w)))

AND gp(Hyp w)
    =>(con(ded h(Hyp w))=h Imp con(Hyp w)
      /\gp(ded h(Hyp w)))

AND gp p1=>(con(ded h p1)=h Imp con p1/\gp(ded h p1)),
    gp p2=>(con(ded h p2)=h Imp con p2/\gp(ded h p2))
    |- gp(MP p1 p2)=>(con(ded h(MP p1 p2))=h Imp con(MP p1 p2)
       /\ gp(ded h(MP p1 p2)))


/* Various lemmas about con: */

THMS conLemmas'(w,x,y)
ARE     isImp (con (K w x))
AND     isImp (con (S w x y))
AND     limp (con (S w x y)) = w Imp (x Imp y)
AND     rimp (con (S w x y)) = (w Imp x) Imp (w Imp y)
AND     limp (con (K w x))   = w
AND     rimp (con (K w x))   = x Imp w

TACTIC conLem IS ALT (Unfold(conLemmas'0),
                      Unfold(conLemmas'1),
                      Unfold(conLemmas'2),
                      Unfold(conLemmas'3),
                      Unfold(conLemmas'4)
                      Unfold(conLemmas'5))


/* Two lemmas about the relationship between ded and con: */
```

```
THMS dedconLemma'(w,h,p1,p2)
ARE isImp(con p2), limp(con p2)=con p1
    |- h Imp con p2=h Imp((con p1)Imp(rimp(con p2)))
AND isImp(con p2), limp(con p2)=con p1,
    con(ded h p2)=h Imp con p2
    |- con(MP(ded h p2)(S h(con p1)(rimp(con p2))))
       = rimp(con(S h(con p1)(rimp(con p2))))

TACTIC dedconLem IS ALT (Unfold(dedconLemma'0),
                         Unfold(dedconLemma'1))

/* A simple tactic and tries to apply some of the new rules
and lemmas, and if that fails, does the most ``obvious''
rewrite: */

TACTIC ObviousPlus IS ALT (Unfold("union"),
                           "conLem",
                           "indep", "gp", "isImp",
                           Obvious)


/* This tactic is capable of proving all the simple cases of
ded's induction (dedLemmas'0 through '4) and the statement
that reflex meets its spec.  ObviousPlus also automates most
of the work in proving the two cases of ded (dedLemmas'5 and
'6). */
```

Jape proof of the first part of dedSpec.

1 :	**gp(K w1 w2)=>(con(ded $h(K w1 w2))=$h Imp con(K w1 w2)/\gp(ded $h(K w1 w2)))**	Conjecture dedLemmas'0
2 :	**gp(S w1 w2 w3)=>(con(ded $h(S w1 w2 w3))=$h Imp con(S w1 w2 w3)/\gp(ded $h(S w1 w2 w3)))**	Conjecture dedLemmas'1
3 :	**gp(CP w1 w2)=>(con(ded $h(CP w1 w2))=$h Imp con(CP w1 w2)/\gp(ded $h(CP w1 w2)))**	Conjecture dedLemmas'2
4 :	**gp(Hyp w)=>(con(ded $h(Hyp w))=$h Imp con(Hyp w)/\gp(ded $h(Hyp w)))**	Conjecture dedLemmas'5
5 :	**gp(Top w)=>(con(ded $h(Top w))=$h Imp con(Top w)/\gp(ded $h(Top w)))**	Conjecture dedLemmas'3
6 :	**gp(Bot w)=>(con(ded $h(Bot w))=$h Imp con(Bot w)/\gp(ded $h(Bot w)))**	Conjecture dedLemmas'4
7 :	**gp p1=>(con(ded $h p1)=$h Imp con p1/\gp(ded $h p1)),**	
	gp p2=>(con(ded $h p2)=$h Imp con p2/\gp(ded $h p2))	assumptions
8 :	**gp(MP p1 p2)=>(con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2)))**	Theorem dedLemmas'6 7.1,7.2
9 :	**gp $p=>(con(ded $h $p)=$h Imp con $p/\gp(ded $h $p))**	Prfinduct $p 1,2,3,4,5,6,7-8

<table>
<tr><td>1 :</td><td>gp(Hyp $w)</td><td>assumption</td></tr>
<tr><td>2 :</td><td>$w=$h</td><td>assumption</td></tr>
<tr><td>3 :</td><td>con(reflex $h)=$h Imp $h</td><td>Theorem reflexSpec’1</td></tr>
<tr><td>4 :</td><td>con(reflex $h)=$h Imp con(Hyp $h)</td><td>Unfolding{con’3} 3</td></tr>
<tr><td>5 :</td><td>con(reflex $h)=$h Imp con(Hyp $w)</td><td>leibnizhyp $w 4,2</td></tr>
<tr><td>6 :</td><td>con(ded $h(Hyp $w))=$h Imp con(Hyp $w)</td><td>Unfolding{ded’1, hyp} 5</td></tr>
<tr><td>7 :</td><td>gp(reflex $h)</td><td>Theorem reflexSpec’0</td></tr>
<tr><td>8 :</td><td>gp(ded $h(Hyp $h))</td><td>Unfolding{ded’1, re=} 7</td></tr>
<tr><td>9 :</td><td>gp(ded $h(Hyp $w))</td><td>leibnizhyp $w 8,2</td></tr>
<tr><td>10 :</td><td>con(ded $h(Hyp $w))=$h Imp con(Hyp $w)/\gp(ded $h(Hyp $w))</td><td>|-/\ 6,9</td></tr>
<tr><td>11 :</td><td>~($w=$h)</td><td>assumption</td></tr>
<tr><td>12 :</td><td>$h Imp $w=$h Imp $w</td><td>re=</td></tr>
<tr><td>13 :</td><td>rimp($w Imp($h Imp $w))=$h Imp $w</td><td>Unfolding{rimp} 12</td></tr>
<tr><td>14 :</td><td>rimp(con(K $w $h))=$h Imp $w</td><td>Unfolding{con’0} 13</td></tr>
<tr><td>15 :</td><td>con(MP(Hyp $w)(K $w $h))=$h Imp $w</td><td>Unfolding{con’6, Unfolding{con’0}} 14</td></tr>
<tr><td>16 :</td><td>con(MP(Hyp $w)(K(con(Hyp $w))$h))=$h Imp con(Hyp $w)</td><td>Unfolding{con’3} 15</td></tr>
<tr><td>17 :</td><td>con(ded $h(Hyp $w))=$h Imp con(Hyp $w)</td><td>Unfolding{ded’2, indep’5} 16</td></tr>
<tr><td>18 :</td><td>isImp((con(Hyp $w))Imp($h Imp(con(Hyp $w))))</td><td>isImp’0</td></tr>
<tr><td>19 :</td><td>isImp(con(K(con(Hyp $w))$h))</td><td>Unfolding{con’0} 18</td></tr>
<tr><td>20 :</td><td>con(Hyp $w)=con(Hyp $w)</td><td>re=</td></tr>
<tr><td>21 :</td><td>limp((con(Hyp $w))Imp($h Imp(con(Hyp $w))))=con(Hyp $w)</td><td>Unfolding{limp} 20</td></tr>
<tr><td>22 :</td><td>limp(con(K(con(Hyp $w))$h))=con(Hyp $w)</td><td>Unfolding{con’0} 21</td></tr>
<tr><td>23 :</td><td>gp(Hyp $w)</td><td>gp’3</td></tr>
<tr><td>24 :</td><td>gp(K(con(Hyp $w))$h)</td><td>gp’0</td></tr>
<tr><td>25 :</td><td>gp(MP(Hyp $w)(K(con(Hyp $w))$h))</td><td>gp’6 19,22,23,24</td></tr>
<tr><td>26 :</td><td>gp(ded $h(Hyp $w))</td><td>Unfolding{ded’2, indep’5} 25</td></tr>
<tr><td>27 :</td><td>con(ded $h(Hyp $w))=$h Imp con(Hyp $w)/\gp(ded $h(Hyp $w))</td><td>|-/\ 17,26</td></tr>
<tr><td>28 :</td><td>con(ded $h(Hyp $w))=$h Imp con(Hyp $w)/\gp(ded $h(Hyp $w))</td><td>eqcases 2-10,11-27</td></tr>
<tr><td>29 :</td><td>gp(Hyp $w)=>(con(ded $h(Hyp $w))=$h Imp con(Hyp $w)/\gp(ded $h(Hyp $w)))</td><td>|-=> 1-28</td></tr>
</table>

Jape proof of dedLemma’5.

Jape proof of dedLemmas'6.

1 :	**gp p1=>(con(ded $h p1)=$h Imp con p1/\gp(ded $h p1)), gp p2=>(con(ded $h p2)=$h Imp con p2/\gp(ded $h p2))**	assumptions
2 :	**gp(MP p1 p2)**	assumption
3 :	**gp p1，gp p2，isImp(con p2)，limp(con p2)=con p1**	assumptions
4 :	**con(ded $h p1)=$h Imp con p1/\gp(ded $h p1)**	assumption
5 :	**con(ded $h p2)=$h Imp con p2/\gp(ded $h p2)**	assumption
6 :	**con(ded $h p1)=$h Imp con p1，gp(ded $h p1)**	assumptions
7 :	**con(ded $h p2)=$h Imp con p2，gp(ded $h p2)**	assumptions
8 :	**$h Imp(rimp(con p2))=$h Imp rimp(con p2)**	re=
9 :	**rimp(($h Imp(con p1))Imp($h Imp(rimp(con p2))))=$h Imp rimp(con p2)**	Unfolding{rimp} 8
10 :	**rimp(rimp(con(S $h(con p1)(rimp(con p2)))))=$h Imp rimp(con p2)**	Unfolding(rimp(con(S $h(con p1)(rimp(con p2))))){Conjecture conLemmas'3} 9
11 :	**rimp(con(MP(ded $h p2)(S $h(con p1)(rimp(con p2)))))=$h Imp rimp(con p2)**	Unfolding{Conjecture dedconLemma'1} 10
12 :	**con(MP(ded $h p1)(MP(ded $h p2)(S $h(con p1)(rimp(con p2)))))=$h Imp rimp(con p2)**	Unfolding{con'6, leibnizhyp con(ded $h p1)} 11
13 :	**con(ded $h(MP p1 p2))=$h Imp rimp(con p2)**	Unfolding{ded'0} 12
14 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)**	Unfolding{con'6, leibnizhyp limp(con p2)} 13
15 :	**isImp(($h Imp(con p1))Imp($h Imp(rimp(con p2))))**	isImp'0
16 :	**isImp(rimp(con(S $h(con p1)(rimp(con p2)))))**	Unfolding(rimp(con(S $h(con p1)(rimp(con p2))))){Conjecture conLemmas'3} 15
17 :	**isImp(con(MP(ded $h p2)(S $h(con p1)(rimp(con p2)))))**	Unfolding{Conjecture dedconLemma'1} 16
18 :	**$h Imp(con p1)=$h Imp con p1**	re=
19 :	**limp(($h Imp(con p1))Imp($h Imp(rimp(con p2))))=$h Imp con p1**	Unfolding{limp} 18
20 :	**limp(rimp(con(S $h(con p1)(rimp(con p2)))))=$h Imp con p1**	Unfolding(rimp(con(S $h(con p1)(rimp(con p2))))){Conjecture conLemmas'3} 19
21 :	**limp(con(MP(ded $h p2)(S $h(con p1)(rimp(con p2)))))=$h Imp con p1**	Unfolding{con'6, leibnizhyp con(ded $h p2)} 20
22 :	**limp(con(MP(ded $h p2)(S $h(con p1)(rimp(con p2)))))=con(ded $h p1)**	leibnizhyp con(ded $h p1) 21,6.1
23 :	**isImp(con(S $h(con p1)(rimp(con p2))))**	Conjecture conLemmas'1
24 :	**$h Imp con p2=$h Imp((con p1)Imp(rimp(con p2)))**	Conjecture dedconLemma'0 3.3,3.4
25 :	**$h Imp((con p1)Imp(rimp(con p2)))=$h Imp con p2**	sy= 24
26 :	**$h Imp((con p1)Imp(rimp(con p2)))=con(ded $h p2)**	leibnizhyp con(ded $h p2) 25,7.1
27 :	**limp(con(S $h(con p1)(rimp(con p2))))=con(ded $h p2)**	Unfolding{Conjecture conLemmas'2} 26
28 :	**gp(S $h(con p1)(rimp(con p2)))**	gp'1
29 :	**gp(MP(ded $h p2)(S $h(con p1)(rimp(con p2))))**	gp'6 23,27,7.2,28
30 :	**gp(MP(ded $h p1)(MP(ded $h p2)(S $h(con p1)(rimp(con p2)))))**	gp'6 17,22,6.2,29
31 :	**gp(ded $h(MP p1 p2))**	Unfolding{ded'0} 30
32 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2))**	|-/\ 14,31
33 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2))**	/\\|- 7-32,5
34 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2))**	/\\|- 6-33,4
35 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2))**	=>|- 5-34,3.2,1.2
36 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2))**	=>|- 4-35,3.1,1.1
37 :	**con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2))**	gp'7 3-36,2
38 :	**gp(MP p1 p2)=>(con(ded $h(MP p1 p2))=$h Imp con(MP p1 p2)/\gp(ded $h(MP p1 p2)))**	|-=> 2-37