A Typed Functional Language Based on Sequent Calculus

Mark Tarver,
Artificial Intelligence Division,
University of Leeds.

May 23, 1994

Abstract

SEQUEL is a new functional language that allows the specification of types within a notation derived from the sequent calculus. Using a synthesis of techniques drawn from both logic and functional programming; types specified in sequent calculus notation are compiled to programs that enable efficient type-checking of SEQUEL programs. The power of the SEQUEL's type notation enables types to be defined which define theoremhood for arbitrary logics and for which there is no decision procedure. To support the process of type-checking for such types, SEQUEL has a sophisticated logic-independent theory of tactics and a graphically-driven interactive proof interface. SEQUEL extends the idea of type-checking beyond what many current functional languages can easily represent and has comparisons with generic theorem-provers like Isabelle and NuPrL as well as languages like Miranda and ML.

Introduction

Current research is tending to the conclusion that there is a deep connection between type theory and logic. The foundations for this conclusion are present in systems like NuPrL (Constable et al. (1986)) in which the process of proof in an arbitrary logic is reduced to the process of proving that a type is inhabited. Although there are now many implementations similar to the NuPrL system, the platform of these implementations is generally a functional language different in no important respects from Edinburgh ML that was the platform of the Edinburgh LCF (Milner,Gordon,Wadsworth(1978)). In particular, no explicit support is given within languages like ML to the idea that, if proof and type-checking are really the same, then the mechanism by which logics and types should be defined should in each case also be the same. ML offers as its provision for defining types, a form of BNF grammar notation that ensures that the user is kept firmly within the bounds of Milner's $\mathcal{W}$ algorithm. The arguments for this arrangement is that decidability is a good thing and programmers should not be encouraged to stray. The arguments against this arrangement is that it obscures the important conceptual connection we have observed and makes much harder, the construction of systems like NuPrL that have a much richer idea of type-checking. SEQUEL, (SEQUEnt processing Language), was constructed with these latter arguments in mind.

The Logics as Types Principle

SEQUEL fairly neatly separates into two parts;

1. A core functional language that is highly suited to the symbolic manipulat ion of logical formulae.
2. An approach to type theory via the sequent calculus that allows the high-level specification of arbitrary logics through sequent calculus notation.

The two are tied together by a philosophy that insists that the specification of logics can be reduced to the specification of types, and that proof within a given logic is equivalent to proving that a type is inhabited. The process of proof in a logic is thus nothing more than a form of type-checking, different from that found in languages like SML because the control mechanisms for driving a proof in a logic are more complex, but alike in kind. One single notation, derived from the sequent calculus, suffices to specify logics and to define the types against which *SEQUEL* programs can be verified as well-typed.

These ideas find their most succinct expression in the Logics as Types Principle; which asserts the continuity of type-checking with deductive reasoning in general.

The Logics-as-Types Principle
The only difference between conventional 'type-checking' and 'logical proof' is that logical proof is harder and often undecidable. But the means by which the proof theory should be specified is the same.

When dealing with more advanced types, it is no longer reasonable to expect the computer to manage the process of type-checking without some guidance from a human being. This guidance may be fully interactive, with the human guiding every step of the proof, or the human may decide to transfer some of her expertise into the machine in the form of a program. The mechanism for representing this exertise in a secure manner is described later in this paper.

This paper has three parts; first the core programming component of *SEQUEL* is described and its compilation into Lisp; second the type discipline of the language and how types are specified in sequent calculus notation with a sketch of how this calculus is compiled into efficient code for type checking; third, a description of how logics are specified in the same sequent calculus notation.

1 SEQUEL as a Rewrite Language

Core SEQUEL is a *backtracking priority rewrite language.* A rewrite language is a functional language in which functions are defined by means of rewrite rules. A *SEQUEL* rewrite rule takes the form $i_1,\ldots,i_n \rightarrow o$ which meets the variable condition that all variables in o also appear in $i_1,\ldots,i_n$. A rewrite rule is triggered by a series of inputs matching $i_1,\ldots,i_n$ and the appropriate variable instantiations are made in evaluating the output. The pattern-matching notation is derived from the Edinburgh Prolog conventions.

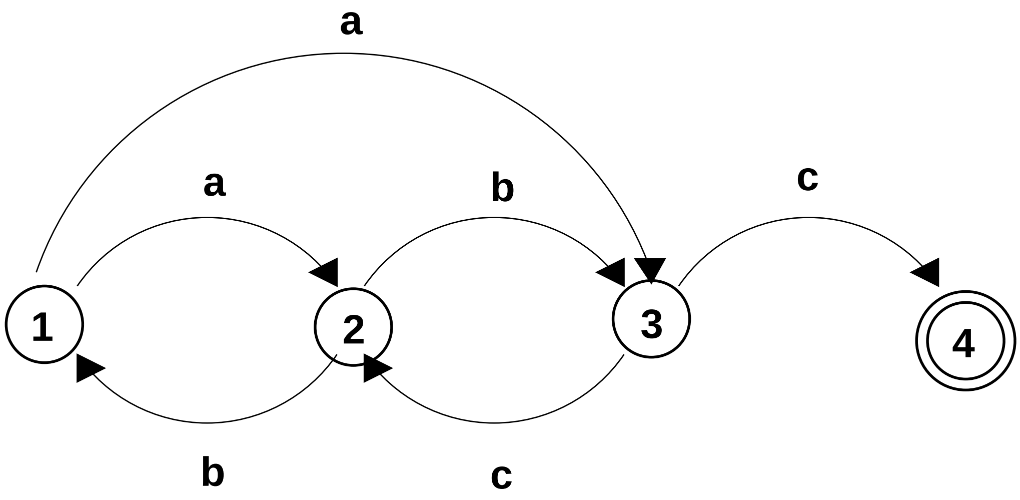
Two rewrite rules are said to *conflict* on an input when they both match to an input, but the results of applying the two rewrite rules are different. A rewrite system is a priority rewrite system when there is a ordering $<$ on the set of all rewrite rules such that if two rewrite rules R1 and R2 conflict on an input then either R1 $<$ R2 or R2 $<$ R1 but not both. Given such an ordering, a priority rewrite system enforces uniqueness of rewriting by requiring that R1 be applied before R2. In *SEQUEL*, rewrite rules are organised sequentially within function definitions and potentially conflicting rewrite rules are ordered by the order of their appearance in the definition. Rules appearing earlier in the definition take precedence over those that occur later. A rewrite rule Ri+1 appearing immediately after another rewrite rule Ri in a function definition is called the successor rule to Ri.

SEQUEL also supports a form of backtracking called *backtracking on demand.* In backtracking on demand a *failure object* $\mho$ is defined, which if returned as the result of a computation causes the computation to return to the last choice point signalled in the program and to try another rewrite rule. More exactly backtracking on demand requires a priority ordering on the set of all rewrite rules and a subset of these rewrite rules are marked as *backtracking rewrite rules.* If the effect of rewriting an input is to produce $\mho$, then *SEQUEL* unwinds the computation back to the last backtracking rewrite rule Ri invoked and the rule Ri+1 is then tried. If the successor of Ri is not defined then an error is signalled. If there is no backtracking rewrite rule to be found then $\mho$ is returned.

Example: Implementing an Non-Deterministic Finite State Machine

A non-deterministic finite state machine (NDFSM) is a machine that reads as an input a series of symbols and decides whether to accept or reject that series. An NDFSM has a number of states and upon receipt of a symbol will change from being in one state to being in another. There is a privileged set of states called *final states* and if the automaton is in one of these states when it finishes reading a series then it accepts that series otherwise it rejects it.

An NDFSM may have the option of jumping to any one of a number of states on receipt of a symbol; but it has the magic property that if there is one state that has to be chosen in order for the whole list to be accepted then the NDFSM will choose that state and not one that will cause the list to be rejected. The operation of an NDFSM can be represented by a labelled directed graph in which the nodes of the graph represent the states of the NDFSM and the labels on the arcs are the inputs required to get the NDFSM to jump from one state to the next. Below is an labelled directed graph illustrating an NDFSM. There is one final state in this NDFSM, state 4. Initially the NDFSM is in state 1. Using semantic backtracking,



it is easy to set up a program to simulate this NDFSM. Rewrite rules that are designed to mark choice points are signalled by the use of <- instead of -> within a function definition and the failure object is designated by the expression `(fail)`.

```
(define ndfsm-state1
  [a | X] <- (ndfsm-state2 X)
  [a | X] -> (ndfsm-state3 X)
  _ -> (fail))

(define ndfsm-state2
  [b | X] <- (ndfsm-state3 X)
  [b | X] -> (ndfsm-state1 X)
  _ -> (fail))

(define ndfsm-state3
  [c | X] <- (ndfsm-state4 X)
  [c | X] -> (ndfsm-state2 X)
  _ -> (fail))

(define ndfsm-state4
  [] -> accept
  _ -> (fail))
```

Unlike the cut in Prolog, semantic backtracking is a semantically clean construction since we can eliminate all places where <- appears through the following transformation.[1]

[1]This transformation is not efficient, since E_2 may be computed twice if `(not (equal` E_2 `(fail)))` is true. The actual compilation technique used by *SEQUEL* does not use this equation.

E_1 <- E_2 $=_{df}$ E_1 (not (equal E_2 (fail))) -> E_2

Another way of invoking semantic backtracking is through the function λxy.(fail-if x y). This is a higher-order function that takes a 1-place function f and evaluates the result of applying the function f to its second argument. The result is a boolean which if `t` forces backtracking if fail-if is placed placed after a <- but which if `nil` allows the second argument to be returned. Effectively λxy.(fail-if x y) allows us to choose what will count as a failure object.

To illustrate, suppose we have written two programs to solve problems in abstract algebra; the first by using intelligent techniques that mimic the way a mathematician might think; the second by using brute force and computer speed. In either case the sign of success is that the machine reduce the equation to one of self-identity (so the problem $(x \circ x^{-1}) = e$ can be reduced to $e = e$). We observe that some problems that evade the first approach fall to the second and so we resolve to combine the two programs, attempting to solve by using intelligent techniques if possible and if not, backtracking to the beginning and using brute force.

In this case, the sign of failure on the part of our 'intelligent' program is that it fail to return a self-identity. It would be useful to be able to change the failure object so that a non-self-identity object would cause failure and backtracking. λxy.(fail-if x y) supplies this need. The top level of our program looks might look like this:-[2]

```
(define solve-abstract-algebra
  EQUATION? <- (fail-if (/. (e) (not (self-identity e)))
                  (try-intelligent-techniques EQUATION?))
  EQUATION? -> (try-brute-force EQUATION?))

(define self-identity
  [X = X] -> t
  _ -> nil)
```

λxy.(fail-if x y) is a powerful means of controlling backtracking which allows the selection of a possibly infinite class of failure objects to control backtracking - including or excluding the failure object itself. For instance λx.(ndfsm-state1 X) can also be defined.

```
(define ndfsm-state1
  [a | X] <- (fail-if (/. (e) (equal e (fail))) (ndfsm-state2 X))
  [a | X] -> (ndfsm-state3 X)
  _ -> (fail))
```

[2]The /. is our representation of the Greek λ and so the anonymous function (/. (e) (not (self-identity e))) is in conventional notation λx.(not (self-identity x)).

1.1 Implementation

All function definitions and type theories are compiled into efficient Common Lisp code via an instruction set of an abstract machine SLAM (SequeL Abstract Machine). In the case of type theories specified in SEQUEL's sequent calculus notation, the constituent axioms of these theories are first reduced to pure Horn clauses (i.e. to Horn clauses that do not use negation-as-failure) and from Horn clauses to SLAM instructions and from SLAM to Lisp. The mechanism by which an axiom works can be fairly easily modelled through a horn clause interpreter which uses occurs-check unification. Thus a sequent becomes a horn-clause goal and a sequent transformation rule becomes a means of mapping goals-as-sequents to new goals-as-sequents.

Since unification is but a extension of pattern-matching, a reinterpretation of the instructions of a virtual machine suitable for compiling a Horn-clause will Yield code suitable for interpreting a pattern-matching functional rewrite language. By choosing a pattern-matching rewrite language as the basis of SEQUEL, considerable economies were made in the implementation of SEQUEL, since one virtual machine would now suffice for the implementation of both the logical and functional components of the language.

1.1.1 The Compilation of Core SEQUEL

The rewrite rules that constitute core SEQUEL are compiled into the virtual instructions of a machine SLAM (SequeL Abstract Machine) and from SLAM to Common Lisp code. Each SLAM instruction represents an encoding of some term found on the LHS of a rewrite rule and each SLAM instruction is a triple <T,X,D> where

1. T indicates the sort of the term. A term can either be:-
   (a) A closed list of fixed length, like [a b X] which matches to only 3 element lists.
   (b) An open list which contains the |, like [a b | X] which matches to all lists beginning a b.
   (c) A constant, for instance henry is a constant.
   (d) A term-variable which is free; in [a b X X] the first occurrence of X is free.
   (e) A term-variable which is bound; in [a b X X] the second occurrence of X is bound.
   (f) A tail-variable which is free; in [a b | X] X is free and a tail variable.
   (g) A tail-variable which is bound; in [[a b | X] [d | X]] the second occurrence of X is bound tail variable.
2. A list of integers that indicates the position of the term. So [1 1] would indicate a term in the first position of the first argument.

3. A flag for dereferencing which is used only when SLAM is used for unification. (see later)

The instruction codes for each of these types is:-

Closed Lists cll
Open Lists .... opl
Constant .... name
Free term variable .. fv
Bound term variable . bv
Free tail variable .. ftv
Bound tail variable .. btv

Example: the SLAM instruction code for the rewrite rule

```
[[a b | X] [d | X]] -> X
```

with the associated Lisp code is:-

```
SLAM Instructions                       Lisp Code

[[cll [[a b | X] [d | X]] [1] no]  (and (consp fp1) (= (length fp1) 2))
[opl [a b | X] [1 1] no]           (and (consp (car fp1))
                                        (> (length (car fp1)) 1))
[name a [1 1 1] no]                (eq 'a (caar fp1))
[name b [2 1 1] yes]                (eq 'b (cadar fp1))
[ftv X [3 1 1] yes]                 <no instruction>
[opl [d | X] [2 1] yes]             (consp (cadr fp1))
[name d [1 2 1] yes]                (eq 'd (caadr fp1))
[btv X [2 2 1]] yes]                (equal (cdadr fp1) (cddar fp1))
```

2 SEQUEL as a Statically Typed Language

SEQUEL is a functional language that marries the technology of logic and functional programming to support the identification of proof with type-checking. Unlike ML, types are specified not in a BNF grammar notation but in terms of *axioms* written in sequent calculus notation. A series of axioms designed to explicate the formal properties of a single type is called a *type theory*. Techniques borrowed from both logic and functional programming are used to derive efficient type-checking programs from type theories.

Thus suppose that the type *binary* of all binary numbers is to be defined such that any list of 1s and 0s inhabits this type. To define this in *SEQUEL* a theory *binary* is constructed which axiomatises the relevant properties. In order to state the axiomatic properties of *binary*, we have to state the conditions under which it can be inferred from a set of assumptions Δ that an object X inhabits the type $binary$. In many cases, proving X inhabits the type $binary$ from Δ will be conditional on proving something else. For instance, proving [1 | X] inhabits the type $binary$ from a set of assumptions Δ is possible if and only if Δ can be used to prove X inhabits the type $binary$.

A *sequent* Σ is a pair $<\Delta$,A$>$ where A is a formula and Δ is a set of formulae from which A is taken to follow. A sequent calculus axiom states the conditions under which any one of a class S of sequents Σ can be solved. In SEQUEL these conditions may be sufficient (signalled by the use of `thus` or necessary and sufficient (signalled by the use of `iff`). In the case of necessary and sufficient conditions $SEQUEL$ uses rewriting or horn-clause reasoning with backtracking for merely sufficient conditions since there may be several sufficient conditions of a sequent being provable. The conditions for Σ to be true (whether necessary and sufficient or merely sufficient) is a n-tuple (n $\geq$ 0)$<$...,$\Sigma_n$$>$ of sequents which are solved in their order of appearance. If n $=$ 0, then the axiom states that any sequent of the class S can be solved. In SEQUEL, sequent calculus axioms are written as schemata and the class S is defined through pattern-matching or unification with the schematic axiom. The schematum that represents the class S of sequents is referred to as the *input schematum* and the schemata that represent the sequents that may be used to solve sequents that match the input schemeta are called the *output schemata*.

In accordance with the Logics as Types Principle, types for the purposes of programming (soft types) are declared through sequent calculus notation. Here is the $SEQUEL$ specification of the type of all binary numbers, where a binary number is a list of 0s and 1s. |- is the logical turnstile and * means 'is of the type'.

```
(theory binary

  thus <A> |- [] * binary

  <A> |- X * binary
  iff
  <A> |- [1 | X] * binary

  <A> |- X * binary
  iff
  <A> |- [0 | X] * binary)
```

Such theories as $binary$ are animated in $SEQUEL$ into effectively computable functions which can be used to type-check functions such as this one:-

```
(define invert
  {binary -> binary}
  [] -> []
  [0 | X] -> [1 | (invert X)]
  [1 | X] -> [0 | (invert X)])
```

2.1 The Type Discipline of SEQUEL

SEQUEL is written on top of Common Lisp and inherits the system functions of Common Lisp, some 769 in number according to Steele (1989)). Currently SEQUEL holds signatures for 340 of these functions and recognises as base types symbols, booleans, strings, integers, characters, floating point and rational numbers. SEQUEL also recognises and will type-check code involving destructive operations involving simple assignments, access and modification of hash-tables, n-dimensional arrays and property lists. The type information on Common Lisp is held within a subsystem XTT (eXtended Type Theory) as a Horn clause program of 1,600 lines. XTT is easily extensible and can be extended by the user to incorporate and type-check code from Lisp libraries. All signatures determined by XTT are fed to the Lisp compiler and are used to generate `declare` and `the` compiler optimisations (see Steele (1988) p450) that use type information to optimise the binary code.

2.2 Compiling Sequent Calculus

Every sequent calculus axiom has either the form

```
<Sequents>                  <Sequents>
iff                          thus
<Sequent>                   <Sequent>
```

'Iff' axioms are compiled using into rewrite rules. 'Thus' axioms are compiled into Horn clauses. To do this we divide the axiom into two parts corresponding to the head and body of the nascent Horn clause.

The head of the Horn clause corresponds to the conclusion of the sequent found in the input schematum. The body corresponds to what occurs before the turnstile of the input schematum and also includes the output schemata. For example the following sequent calculus axiom:- axiom for & introduction.

```
:name &-intro
X * theorem, Y * theorem, <A> |- Z * theorem
thus
(X & Y) * theorem, <A> |- Z * theorem
```

is translated into an Horn clause procedure that if written in Prolog might appear thus:-

```
&-intro([DELTA,>>,[R,*,theorem]] | SEQUENTS],
       [[DELTA_1,>>,[R,*,theorem]] | SEQUENTS])
:-
% corresponds to the (X & Y) * theorem in the axiom
member([X,&,Y],*,theorem],DELTA),
```

```
% corresponds to fact that the axiom shows that (X & Y) * theorem
% is not in the output sequent
remove([X,&,Y],*,theorem],DELTA,DELTA_0),
% corresponds to the prescence of P * theorem, Q * theorem in the output sequent
append([X,*,theorem],[Q,*,theorem]],DELTA_0,DELTA_1).

% if the axiom fails in application then it behaves as an identity function
% in respect of the proof

&-intro(SEQUENTS,SEQUENTS).
```

This Horn clause is compiled into SLAM instructions, the head of the Horn clause taking the role of the LHS of a rewrite rule. The SLAM instructions are the same, however they are interpreted into Lisp instructions for unification.

2.2.1 Optimising Dereferencing

It is well-known that type-checking requires occurs-check unification and that Prolog implementations, with rare exceptions, avoid the occurs-check on the grounds that it depreciates the performance of the inference engine. The need for an occurs-check required a design for SLAM different from that of the accepted model for Prolog compilation which is the Warren Abstract Machine (Warren (1983)) or WAM. Proofs of well-typedness can be very long and the overheads of the occurs-check and dereferencing are heavy so producing an efficient compiler for occurs-checking Horn clause engine to do type-checking is a non-trivial task. Robinson's unification algoritheorem depreciates in exponential time against the complexity of the expressions checked.

Workers in the field of high-performance theorem-proving (Plaisted (1984), Stickel (1986)) have tackled this problem with some success. In particular Plaisted has shown that Horn clauses can be statically analysed to determine where occurs-checking needs to be done, thus eliminating many of the unnecessary occurs-checks that the naive algoritheorem imposes. To these techniques SEQUEL adds a dynamic run-time optimisation that sets a flag at the beginning of every unification process to 0. The flag is set to 1 just when a variable in the goal assumes a binding. If the flag is still set to 0 at the end of the unification process then no problem variable has been bound and effectively only pattern-matching and not unification has been performed. The setting of the flag determines the way in which dereferencing is performed. In pattern-matching, dereferencing is performed in constant time (simply follow a pointer) whereas dereferencing for occurs-check unification is quadratic (Maier & Warren (1988)). By using pattern-matching dereferencing whenever it was flagged as safe to do so, SEQUEL quadruples the performance of the generated code.

3 Representing Logics in SEQUEL

Types for which there exists no decision procedure for determining for any arbitrary object whether it inhabits such a type or not, are here referred to as *hard types*. The restrictions imposed by the type definition mechanisms of ML and related languages ensure that hard types cannot be introduced through these definition mechanisms; however the sequent calculus notation of SEQUEL allows hard types to be defined (e.g. the type of all first-order theorems). Hard types are usually introduced into SEQUEL in the course of defining formal systems of which first-order logic is only one example. Consequently the discussion of hard types can be usefully submerged in the discussion of how SEQUEL treats the representation of logics.

If hard types/logics are to be catered for within SEQUEL then the user must be given the power to orchestrate the process of type-checking or animation. Instead of a fixed control strategy determined by rewriting to a canonical form and LUSH resolution, more sophisticated and possibly interactive processes must be used which must, at the same time, guarantee the soundness of the final result. In order to achieve these ends SEQUEL regiments the process of animation for hard types by reference to a formal model of what a logic is. This formal model is called a *framework*. A *framework* maps onto the computationally relevant parts of a logic. Specifically a framework consists of:-

1. A syntax.
2. A proof theory which consists of a series of sequent calculus axioms and a set of rewrite rules.
3. A pragmatics which consists of a series of *tactics* for solving problems couched in this framework.

The syntax of a framework is laid down by defining the type of all *T-expressions* (T-exprs). A T-expr is a typed expression of the form e : τ where e is a well-formed formula (wff) and τ is a type. By defining the types *wff* and *type* in the form of type theories the SEQUEL type checker will certify the syntax of all user inputs.

The axioms of a framework are laid out using the same sequent calculus notation as before, except that the theory is declared to be interactive; and rewrite rules are laid down as SEQUEL functions of type T-expr $\rightarrow$ T-expr. Once a framework has been defined, the tactics that consitute the pragmatics of the framework are constructed out of operations on axioms and rewrite rules and these tactics can be verified as sound by the resident type-checker. The permitted operations are:-

- **Refine** given a sequent calculus axiom and a given sequent, refining that sequent in respect of that axiom is an atomic operation.
- **XTT** given any sequent, solving it by XTT using the `XTT` command is an atomic operation.

- **Rewrite** given any formula, rewriting it to another formula according to a designated rewrite rule is an atomic operation.
- **Thin** given any sequent, removing one of the assumptions is an atomic operation.
- **Swap** given any sequent, exchanging the position of any two assumptions is an atomic operation.
- **Rotate** given a list of sequents to solve, exchanging the position of any two sequents is an atomic operation.
- **Lemma** to introduce a lemma at any point in the proof is an atomic operation. The lemma becomes a new goal. If solved, it is inherited of every context of every sequent in the proof-tree below the sequent in which it is introduced.

The execution of one of these primitive operations is called a *tactical inference* and the speed of execution of theorem-provers designed in *SEQUEL* is measured in TIPS or *tactical inferences per second.* A graphical interface that runs under X-windows is generated automatically for the implementation in which the tactics can be accessed through a mouse.

Example: propositional calculus

The following theory defines the type *theorem* for propositional calculus if some basic rewrite rules are added. The theory illusrates the device of naming axioms as an alternative to invoking them through REFINE (so CONTRADICTION is short for REFINE PROP 2).

```
(theory prop :interactive yes

   :name indirect-proof
   (~ X) * theorem, <A> |- false * theorem
   thus
   <A> |- X * theorem

   :name contradiction
   thus X * theorem, (~ X) * theorem, <A> |- false * theorem

   :name &-left
   X * theorem, Y * theorem, <A> |- false * theorem
   thus
   (X & Y) * theorem, <A> |- false * theorem

   :name V-left
   X * theorem, <A> |- false * theorem
```

```
   Y * theorem, <A> |- false * theorem
   thus
   (X V Y) * theorem, <A> |- false * theorem)
```

The added rewrite rules are:-

```
(defrew dneg
  [[~ [~ X]] * theorem] -> [X * theorem])

(defrew deMorgan
  [[~ [X V Y]] * theorem] -> [[[~ X] & [~ Y]] * theorem]
  [[~ [X & Y]] * theorem] -> [[[~ X] V [~ Y]] * theorem])

(defrew negequiv
  [[~ [X <=> Y]] * theorem] -> [[[X & [~ Y]] V [[~ X] & Y]] * theorem])

(defrew negimp
  [[~ [X => Y]] * theorem] -> [[X & [~ Y]] * theorem])

(defrew equiv-elim
  [[X <=> Y] * theorem] -> [[[X => Y] & [Y => X]] * theorem])

(defrew imp-elim
  [[X => Y] * theorem] -> [[[~ X] V Y] * theorem])
```

The following short program with comments after ;s gives a semantic tableau theorem prover for PC which is sound and complete. It embodies the following procedure.

1. Every proof begins with a step of indirect-proof and this is the only place where this axiom is invoked.
2. All the rewrite rules of PC are applied in a normalising fashion to the point where they can no longer be applied to produce a change.
3. Sequents containing contradictions should be eliminated by the contradiction axiom.
4. In cases where the formulae have been normalised and no contradiction has been found, but there is a disjunction ($P \vee Q$) in the assumption list of the sequent, the branch axiom should be used.
5. The preceding rules should be applied until either (a) the prooftool returns yes or (b) none of the above is applicable. If (b) the original formula is not a theorem of PC.

```
(deftactic zap
```

```
  ; apply indirect proof and repeat steps 2-4 until
  ; they can no longer be applied to produce change
  {proof -> proof}
  PROOF? ->  (fixpoint PC-solve (indirect-proof PROOF?)))

(define fixpoint
  ; find a fixpoint of a function
  F? X -> (fixpoint1 F? X (funcall F? X)))

(define fixpoint1
  _ X X -> X
  F? X Y -> (fixpoint1 F? Y (funcall F? Y)))

(define PC-solve
  ; apply the CONTRADICTION axiom
  ; then the &-left axioms
  ; then perform rewriting
  ; then branch on any disjuncts found
  {proof -> proof}
  PROOF? -> (branch (do-rewrites (&-left (contradiction PROOF?)))))

(define do-rewrites
  {proof -> proof}
  PROOF? -> (do-rewrites1 (assumptions# PROOF?) PROOF?))

(define assumptions#
  ; find the number of assumptions in the context of the
  ; current sequent.  head has the type proof -> sequent
  ; and allows us to interrogate the first sequent of the
  ; proof to find out what is going on
  {proof -> integer}
  PROOF? -> (length (context-in (head PROOF?))))

(define context-in
  ; takes a sequent and returns the hypothesis list
  {sequent -> (list T-expr)}
  [<A>? |- _] -> <A>?)

(define do-rewrites1
  ; works through each assumption applying rewriting
  ; to it
  {integer proof -> proof}
   0 PROOF? -> PROOF?
   N? PROOF? -> (do-rewrites1 (1- N?)
                             (do-rewrite N? PROOF?)))

(define do-rewrite
```

```
; applies each rewrite rule in turn to the chosen
; T-expr
{integer proof -> proof}
N? PROOF? -> (rewrite deMorgan N?
                (rewrite dneg N?
                  (rewrite negequiv N?
                     (rewrite equiv-elim N?
                        (rewrite imp-elim N? PROOF?))))))
```

Accessing SEQUEL

SEQUEL is public domain software and can be accessed through anonymous ftp. The address is agora.leeds.ac.uk and the files are held in scs/logic.

References

1. Constable et al. **Implementing Mathematics with the NuPrl Proof Development System**, Prentice-Hall, (1986).
2. Maier D. & Warren D. S. **Computing with Logic**, MIT Press, (1988).
3. Milner R, Gordon M, Wadsworth C.P. **Edinburgh LCF**, Springer- Verlag, LNCS, (1979).
4. Paulson L. C. **Logic and Computation - Interactive Proof with the Cambridge LCF**, Cambridge University Press, (1987).
5. Plaisted D. *The Occur-Check Problem in Prolog*, in **New Generation Computing**, (1984).
6. Steele G. L. **Common Lisp; the language**, Digital Press, 1988.
7. Stickel M. *A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler*, **JAR** (1988).
8. Tarver M. *A Language for Implementing Arbitrary Logics*, IJCAI vol II (1993).
9. Warren D.H.D. *An Abstract PROLOG Instruction Set* Technical Report, SRI, Menlo Park, USA, (1983).