A Language for Implementing Arbitrary Logics

Mark Tarver
mark@uk.ac.leeds.scs,
Division of Artificial Intelligence,
Centre for Theoretical Computer Science,
University of Leeds, Leeds, LS2 9JT, UK.

Abstract

SEQUEL is a new-generation functional programming language, which allows the specification of types in a notation based on the sequent calculus. The sequent calculus notation suffices for the construction of types for type checking and for the specification of arbitrary logics. Compilation techniques derived from both functional and logic programming are used to derive high-performance ATPs from these specifications.

1 Introduction

Many of the seminal advances in the field of automated deduction are encapsulated in certain landmark implementations; Bledsoe's IMPLY (Cohen & Feigenbaum (1982)), the Boyer-Moore Theorem Prover (Boyer & Moore 1979), and NuPrL (Constable et al. 1986). In each of the cited cases, the implementation of these systems required thousands of lines of code and many man months or years of effort before systems of an acceptable degree of reliability could be constructed. The sheer effort in implementing a good ATP[1] or PA[2] can be gauged by sampling some of the better-known implementations.

Figure 1. Code Profile of Some Leading ATPs.

System	Lines	Based On	Language
LLAMA	4K	Many Sorted Logic	Lisp
ITP	60K	First Order Logic	Pascal
Boyer-Moore	100K	Recursive Functions	Lisp
NuPrL	114K	Martin-Lof	Lisp/ML
Oyster	4K	Martin-Lof	Prolog

The recognition that much effort was required to implement any significant proof-assistant or theorem-prover has led computer scientists to take an interest in generic logics that are capable of representing a wide number of formal systems. It would be more sensible to implement one generic logic within which all other logics could be expressed than to continually reimplement logics from scratch.

The problem is that scientists and logicians are in disagreement about what that generic logic should be; whereas Kowalski (1980) speaks for an older tradition in advocating first-order logic, more recent developments covered in Thompson (1991) point towards some form of constructive type theory. However there are many different systems of constructive type theory and no single system has yet emerged as predominant. The upshot is that although scientists agree that traditional ways of building proof-assistants and theorem-provers is costly, and that better ways should be found, no single logic commands sufficient support to be recognised as a generic logic suitable for all purposes.

A more promising approach is to try to isolate what all logics have in common and then build a shell. This approach is encapsulated by Abrial's B-Tool (Abrial (1986)). Logics are built in the B-Tool by entering the appropriate axioms into the machine. This is indeed a welcome alternative to typing in thousands of lines of code. But since the B-Tool is not itself a programming language, but a tool written in Pascal it lacks the flexibility of control that a programming language such as SML would provide. A better solution would be to integrate the solution of the specification of an arbitrary logic into a functional programming language of a new generation that would absorb the lessons to be learnt from Abrial's B-Tool and languages like Standard ML. This is a good description of the *SEQUEL* programming language.

2 The Structure and Philosophy of SEQUEL

SEQUEL fairly neatly separates into two parts;

1. A core functional language that is highly suited to the symbolic manipulation 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

[1] Automated theorem prover.
2Proof Assistant.

* 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.

3 Core SEQUEL

Core *SEQUEL* runs on top of Common Lisp for both portability and the richness that the Lisp environment has to offer. The core language is a *conditional priority rewrite language* modelled after Prolog syntax with the novel feature of allowing a version of backtracking called *backtracking on demand.*

A rewrite language is a functional language in which functions are defined by means of rewrite rules. A *SEQUEL* rewrite rule takes the form $i1,\ldots,i_n \rightarrow o$ which meets the variable condition that all variables in o also appear in $ii,..,i_n$- A rewrite rule is triggered by a series of inputs matching $i1,\ldots,i_n$ and the appropriate variable instantiations are made in evaluating the output. 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 solution to the problem of conflict is to order the rewrite rules so that when two conflict, only one is allowed to fire. Such a rewrite system is called a *priority* rewrite system in Baeten et al. (1987) where it is shown that priority rewrite systems have an expressive capacity in excess of rewrite systems that lack priority. As an example, the following pair of rewrite rules defines the equality relation only with the proviso that the first has priority over the second. In a non-priority rewrite system this relation cannot be defined.

```
II -> true
XY -> falsa
```

By organising such rules into function definitions with the capacity to call other functions and letting the order of appearance determine priority, a language very like core *SEQUEL* is created. A rewrite rule R_{i+1} appearing immediately after another rewrite rule R_i in a function definition is the successor rule to R_i.

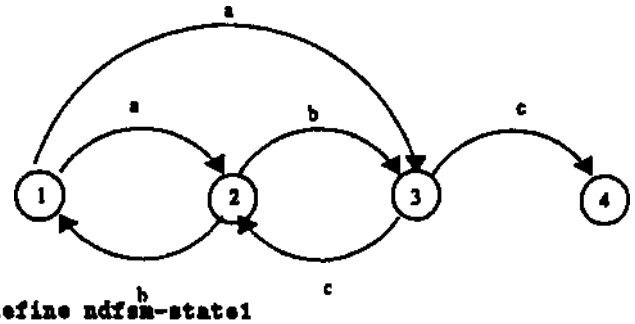
In backtracking on demand a *failure object* U is defined which can be returned by any rewrite rule. When U is returned, *SEQUEL* unwinds the current environment E back to a previous environment E' such that:-

1. E' a s the environment in force when a rewrite rule R_i was applied,
2. R_i was signalled in the SEQUEL program as a choice point rewrite rule*
3. R_i was the last such choice point rewrite rule applied.

Such choice point rewrite rules are signalled by the use of <- instead of -> within a function definition. If no such R_i can be found, then U is returned, else the rule R_{i+1} becomes the highest priority rule to be applied in environment E'.

Backtracking on demand is a very powerful feature of core *SEQUEL* which gives *SEQUEL* the kind of facility with non-deterministic algorithms that Prolog possesses, but with far more control. Backtracking rewrite rules can be selected to deal with the choice points in the program, leaving the deterministic parts of the program free from the unnecessary backtracking that forces Prolog programmers to resort to the use of the cut. A nice illustration of core *SEQUEL's* power is shown in this short program which simulates a non-deterministic finite state machine (NDFSM).

Figure 2. An NDFSM




```
(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, backtracking on demand has a clean semantics since all functions that use <- can be statically transformed into extensionally equivalent functions that do not use <-.

Core *SEQUEL* is compiled into Common Lisp via an abstract machine *SLAM*[3] that produces Lisp code of a size and efficiency comparable to that of hand-written code. The best testimony to the efficiency of a language is that the author should choose to implement the language in itself. Though the *SEQUEL* system consists of 22,000 lines of Common Lisp, only 600 lines of it were written in Common Lisp. The remainder was generated from 4,000 lines of *SEQ UEL.*

4 Flexible Typing

SEQUEL is currently the unique inhabitant of the class of flexibly-typed languages. A flexibly-typed language is one in which:-

[3]SequeL Abstract Machine

- Static and strong type-checking is optional
- The type-discipline is visible, declarative and logic-driven rather than procedurally encoded and invisible from the user.

The type discipline of *SEQUEL*, XTT[4], describes the types of 200 Common Lisp functions and is encoded in Horn clause logic. Since SLAM is an abstract machine for the compilation of Horn clauses, these Horn clauses are reduced to efficient Lisp using techniques derived from WAM (Maier & Warren (1986)). Efficient occurs-check unification is required for type-checking and this is enabled using compilation techniques pioneered by Stickel (1986) and Plaisted (1984). With type-checking switched on, *SEQUEL* has the capabilities of a statically typed functional language like SML. However since *SEQUEL* exploits conceptual relations between sequent calculus and Horn clause logic to allow the specification of types in sequent calculus notation, *SEQUEL'S* ability to rapidly prototype ATPs and PAs exceeds that of SML by at least an order of magnitude.

5 Specifying Types in *SEQUEL*

Types are specified in SEQUEL in terms of *type theories* where a type theory is a set of sequent calculus axioms that describe what it is for an object to inhabit a type. 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 is a partial function on the set of all Σs that returns a tuple of sequents $<...,\Sigma_n>$ ($n \geq 0$) together with a necessity condition N that states whether $<...,\Sigma_n>$ is merely sufficient to prove Σ_0 or necessary and sufficient. Since N is two-valued, sufficiency or necessity plus sufficiency can be indicated by 0 and 1 respectively. The statement that [1 | X] inhabits the type *binary* if and only if X inhabits the type *binary* is then associated with a function that receives any pair $<\Delta,$'[1 | X] : binary'$>$ and returns the pair $<<\Delta,$'X : binary'$>,1>$.

SEQUEL receives sequent calculus axioms of this form and generates the associated functions from these axioms which drive the process of type checking. This process is referred to as *animation*. Instead of associating a numerical value to indicate the status of N, *SEQUEL* uses either straightforward rewriting (where N = 1) or horn-clause reasoning with backtracking (where N = 0) since there may be several sufficient conditions of inhabitation. The specification of *binary* in *SEQUEL* is as follows[5]:-

```
(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)])
```

6 Logics as Types

SEQUEL is founded on the philosophy that logics can be specified as types. The most significant difference between the specification of a logic and that of a type like *binary* is that proving an expression inhabits a type defined according to the proof theory of a logic is generally more complex. In the case of *binary*, *SEQUEL* animates that type by a decision procedure that determines whether an expression inhabits that type. But in the case of more complex types, like the type of all first-order theorems, the control strategy for animation has to be laid down by the user.

Logics are described to *SEQUEL* using the same sequent calculus notation that is used to define other types. An important difference is that such theories are generally declared to be *interactive* i.e. the control strategy for animation will be fixed by the user. Such control strategies are defined in terms of a structure called a *framework* within which the logic is defined.

Frameworks are founded on the idea that a logic is a structured object with discernable components.

1. *A logic has a syntax.* There are rules of formation that determine unambiguously when an expression counts as a formula of that logic and when it does not. For example, thinking of arithmetic as a 'logic', the expression 1 + 2 - (4 * 5) is a formula; but * 6 * -78 + + - 7 is not. Formulae of a logic that meet the requirements of its syntax are called *well-formed formulae* or *wffs* for short.
2. *A logic has a proof theory.* A proof theory gives rules for deriving wffs from wffs. For example, in arithmetic we may have the proof rule a+b = c implies a = c-b.

[4] eXtended Type Theory

[5] <A> abbreviates any arbitrary set of assumptions; *iff* indicates that two problems are essentially equivalent. |- is the logical turnstile; and * is to be read as "is of the type".

3. *A logic has a semantic* or model theory.* A semantics gives rules for assigning interpretations to wffs. For example, in arithmetic we may interpret a, b, c as denoting 1,2 and 3 respectively, but not Tom,Dick and Harry. The semantics of the language of arithmetic forbids this sort of interpretation.
4. *A logic has a pragmatics;* or a set of principles to help people (or computers) reason with it. For example, in algebra we have a principle that tells us to isolate all occurences of an unknown on one side of an equation. This is part of the pragmatics of algebra.

In *SEQUEL* there is a structure called a *framework* that maps onto the computationally relevant parts of a logic. Specifically a framework is a triple F = <S,P,T> where:-

1. S is a syntax.
2. P is a proof theory which consists of a series of sequent calculus axioms and a set of rewrite rules.
3. T is a pragmatics which consists of a series of *tactics* for solving problems couched in F.

The syntax of a logic L is laid down by axiomatising what counts as a T-expr (typed expression). A T-expr is of the form p * r where p is a wff and r is a type. The syntax is determined by axiomatically defining the types *type* and *wff as* type theories in the form of sequent calculus axioms in exactly the same way as we have seen for *binary.* Once the syntax has been defined in this way, it is animated by *SEQUEL* into a decision procedure for checking that all expressions submitted as T-exprs are in fact T-exprs. In particular, no proof-theoretic axiom or rewrite rule of L can be defined that produces non-T-exprs as outputs.

The proof theory itself is written in a mixture of *SEQUEL* and sequent calculus. For instance, the axiom called BRANCH in the framework TABLEAU for first-order logic is characterised by the following axiom.

```
:name branch
X * thm, <A> |- Z * thm
Y * thm, <A> |- Z * thm
thus
(X V Y) * thm, <A> |- Z * thm
```

SEQUEL translates this axiom into an equivalent Horn clause procedure that if written in Prolog might appear thus:-

```
branch([[DELTA,>>,[R,*,thm]] | SEQUENTS],
       [[DELTA1,>>,[R,*,thm]],
        [DELTA2,>>,[R,*,thm]] | SEQUENTS])
 :- member([[P,v,Q],*,thm],DELTA),
    !,
    remove([[P,v,Q],*,thm],DELTA,DELTA_AUX),
    add([P,*,thm],DELTA_AUX,DELTA1),
    add([Q,*,thm],DELTA_AUX,DELTA2).

branch(SEQUENTS,SEQUENTS).

member(X,[X | _]).
member(X,[_ | Y]) :- member(X,Y).

remove(_,[],[]).
remove(X,[X | Y],Z) :- !, remove(X,Y,Z).
remove(X,[Y | Z],[Y | W]) :- remove(X,Z,W).

add(X,Y,[X | Y]).
```

Though Prolog is generally rated as a particularly powerful declarative language, especially when dealing in logic-based formalisms, the simplicity of the *SEQUEL* specification as compared to the equivalent Prolog is a good demonstration of the advantages of specifying logics in *SEQUEL*.[6]

Axioms like branch are characterised by functions that have the type *proof* → *proof*; i.e. the application of an axiom α to a developing proof produces something that, if completed, functions as a proof of the input to α. In cases where α is inapplicable, α functions as an identity function as the Prolog above shows. The application of an axiom to a developing proof is called a *refinement* and these applications are driven by the function *refine* of type *symbol integer proof* → *proof* - the symbol being a theory within which the axiom is found, the integer n denoting the nth axiom of the theory.

Experience has shown that not all axioms are best represented in sequent calculus. Certain forms of reasoning lend themselves best to rewrite rules and so rewrite rules are also admitted to the proof theory of a framework. Such rules have the type *T-expr* → *T-expr*; an example of which being:-

```
(defrew double-negation
  {T-expr -> T-expr}
  [[" [" X]] * thm] -> [X * thm])
```

Rewrite rules are invoked by the inbuilt tactical[7] *rewrite* of type *(T-expr → T-expr) integer proof → proof.* Rewrite applies the chosen rewrite rule to the nth T-expr in the sequent as denoted by the choice of n for the integer position.

7 Side Conditions

SEQUEL has a powerful and flexible treatment of side conditions to axioms. A side-condition in SEQUEL is a boolean-valued expression that if it evaluates to false causes the axiom to fail in its intended application and behave as the identity function. Side-conditions can also be used to explicate complex expressions by binding variables to the result of evaluating functional expressions. All side conditions are type checked.

For example, suppose that we wish to introduce the rule of beta reduction in the lambda calculus into a *SEQUEL* framework. Given an expression ((lambda x (f

[6] The example is more advantageous to Prolog than it should be. Since most Prologs do not include occurs-check unification, this sort of code would, in general, be inadequate for the task of implementing TABLEAU. A proper implementation would replace an implicit call to Prolog's unification by a call to an explicit unification algorithm written by the user. The resulting code would be more complex, and less efficient, than the Prolog cited here.

[7] A higher-order function that takes tactics as arguments

x)) a) we want to β-reduce this to (f a) by replacing bound variable x by a. Using the :side-condition command all this can be parcelled up into one axiom.

```
:name beta-reduce
:side-condition (bind BETA-REDUCE?
                      (subst-for Z X Y))
<A> |- BETA-REDUCE? * TYPE?
thus
<A> |- ((lambda X Y) Z) * TYPE?
```

The bind function has the effect of binding the free variable BETA-REDUCE? to the result of the beta-reduction as a side-effect and returns true as its result. The function bind has the type $\alpha\ \alpha \rightarrow bool$; hence the expression to which BETA-REDUCE? is bound has to be of the same type as the expression returned by (subst-for Z X Y). In this way *SEQUEL* maintains the syntactic integrity of the logic under the use of side-conditions.

8 Tactic = Sequent Calculus + Control

It is possible to construct any proof in *SEQUEL* simply by persisting with refinement and rewriting. But this is a very tedious way to do proofs. There are three ways in which *SEQUEL* lightens the production of proofs.

- Through the use of tactics.
- Through derived rules.
- Through a machine-generated graphical interface.

The use of tactics to automate computer-assisted reasoning has been a feature of automated reasoning since Gordon, Milner and Wadsworth's work on the LCF in 1977. The essential idea is that tactics are functions built out of primitive functions that are guaranteed to be validity-preserving using composition operators that are guaranteed to produce new functions that are also validity-preserving. The LCF system was geared towards proofs in the PP-λ calculus and the primitive functions were specific to that system and no other. The primitives in *SEQUEL* are generic to all logics. There are 8 such operations in all.

1. REFINE
2. REWRITE
3. XTT The command XTT invokes XTT over the problem. XTT is the resident type system of *SEQUEL*. If a conclusion is unifiable with an assumption in the context, XTT will solve it. However where there is a choice of where to perform the unification (with the third or fifth formula for instance) XTT will always pick the earlier one in the list of assumptions. XTT has the type *proof* $\rightarrow$ *proof*.
4. THIN The command THIN followed by an integer *n* removes the *n*th assumption from the context. Its type is *integer proof* $\rightarrow$ *proof*.
5. SWAP The command SWAP followed by integers *m* and *n* exchanges the *m*th and *n*th T-exprs in the context with each other. The type of SWAP is *integer integer proof* $\rightarrow$ *proof*.
6. ROTATE The command ROTATE followed by integers *m* and n exchanges the mth and nth sequents in the stack of remaining unsolved sequents with each other. The type of ROTATE is *integer integer proof* $\rightarrow$ *proof*.
7. LEMMA If the command LEMMA followed by a T-expr *t* is typed to a sequent $\Delta \vdash A$, then *SEQUEL* poses the problem $\Delta \vdash t$. Upon solving this, the original sequent $\Delta \vdash A$ becomes replaced by *t*, $\Delta \vdash A$. The type of lemma is *T-expr proof* $\rightarrow$ *proof*.
8. INST - short for instantiate - allows the instantiation of a variable by a value. INST is a 3 place function of type α *term proof* $\rightarrow$ *proof*. Given a variable as its first argument, INST replaces it by the term in the second argument throughout the developing proof.[8]

The execution of one of these primitives is called a *tactical inference* and the performance of ATPs built under *SEQUEL* is measured in TIPS or *tactical inferences per second*. Due to the efficient compilation techniques from rewrite rules and sequent calculus to Lisp and finally to machine code, SEQUEL-generated ATPs can run over 2,000 TIPS on a SPARC II.

Any tactic constructed out of these 8 primitives that can be verified as having the type .... *proof- proof* will be admitted as a sound or *type 2 tactic.SEQUEL* also recognises *type 0* and *type 1* tactics[9] according to the following classification.

Type 0: these are tactics which consist of raw unchecked code of no established signature. Type 0 tactics can be declared to the system and used as such, but *SEQUEL* will warn the user that the tactics are type 0 before admitting them into its library.

Type 1: these are tactics which are established as outputting a syntactically well-formed stack of sequents as an output if supplied with a syntactically well-formed stack of sequents as an input. Type 1 tactics are also referred to in *SEQUEL* as *syntactically correct* tactics.

Type 2: these are tactics which are established as mapping proofs to proofs. This means that given a proof P as an input, type 2 tactics will output a proof P' as an output where if P' can be proved then so can P. Type 2 tactics correspond to safe or validity-preserving moves in a proof.

[8] The type *term* is undefined in XTT which means that it is the responsibility of the user to define this type in suck a way that INST preserves its signature. In effect, the result of replacing a variable in a T-expr by a term must be to creates another T-expr. INST is the only primitive which makes this sort of requirement. If the first argument is not a variable then INST acts as the identity function.

[9] In an early report, type 3 tactics were recognised. These were tactics that not only mapped proofs to proofs, hut were guaranteed to produce an output that was provable if *and only if t*he input was. In Paulson (1986) type 3 tactics are referred to as *conservative* tactics. *SEQUEL* does not acknowledge type 3 tactics, although it would be possible to effectively recognise some kinds of type 3 tactics.

SEQUEL tactics, properly constructed, display a clean division between *declaretive* clarity, donated from the sequent calculus, and *procedural* control, donated from the atomic operations allowed on them.

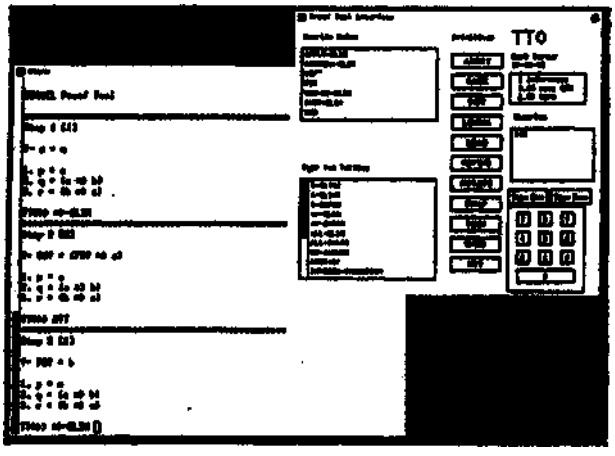
9 Derived Rules

Once a proof has been secured then the theorem can be turned into a derived rule. The non-logical constants in the theorem are uniformly replaced by variables and the theorem becomes a sequent calculus rule. The non-logical constants are specified by the user. The facility is modelled after the Edinburgh Interactive Proof Editor (Lindsay etal. (1986)).

10 Graphical Interface

ATPs and proof assistants are much improved by a graphical interface and the ability to drive proof using a mouse. *SEQUEL* includes a C-based graphical interface based on the highly portable ATHENA Widgets library designed for X-Windows by M.I.T. Any framework designed under *SEQUEL* will generate the appropriate graphical interface with all the rewrite rules and tactics displayed and proofs are driven by 'point and click'. Facility is included whereby the user can attach help to tactics by associating information-holding strings with the names of tactics and rewrite rules and this information is fed into pop-up help windows which are summoned by using the mouse.

Figure 3. A Graphical Interface for the Logic TTO



11 Productivity and Applications

About half-a-dosen ATPs and proof assistants have now been built under *SEQUEL* including theorem-provers for first-order logic without equality/ simple algebraic set theory, a calculus for spatial relations, and experimental systems for partial evaluation and constructive type theory - all with graphical interfaces and mouse-controlled on-line help. Most of these systems are available under ftp.

In a conventional language this output would represent the combined efforts of a research team of three people working solidly for at least a year. In fact the sum of all these systems in *SEQUEL* represented an investment of 6 working weeks for the author.

12 Obtaining SEQUEL

SEQUEL is free for all educational activities that do not involve commercial profit. *SEQUEL* currently runs under Lucid Common Lisp version 4.01 and Kyoto Common Lisp. *SEQUEL* is available with a basic manual by ftp from the University of Leeds. To obtain *SEQUEL,* ftp to the address agora.leeds.ac.uk, cd to scs/logic, type 'binary[9] and then 'mget *' and finally 'quit'. The file README contains instructions on what to do with the software. There is a *SEQUEL* interest group that receives e-mail on new implementations of and in *SEQUEL;* interested parties should e-mail the author at Leeds.

Acknowledgements

I should like to acknowledge the support given by Peter Dew, David Hogg and Tony Cohn to my work. Thanks also to Jeremy Littler who managed the intricacies of the ATHENA Widgets library and the challenges of interfacing C to Common Lisp.

References

[Abrial, 1986] J.R. Abrial *B User Manual,* Oxford, 1986.

[Baeten and Bergstra, 1987] J. Baeten and J. Bergstra, *Term Rewriting Systems with Priorities,* Rewriting Techniques and Applications, LNCS 256,1986.

[Boyer and Moore, 1979] Boyer R.S. and Moore J.S. *A Computational Logic,* Academic Press, 1979.

[Cohen and Feigenbaum, 1982] P.R. Cohen and E.A. Feigenbaum *The Handbook of Artificial Intelligence* Pitman, 1982, p95-101.

[Constable et al., 1986] R.L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System,* Prentice-Hall, 1986.

[Kowalski, 1986] R. Kowalski, SIGART Newsletter, Special Issue on Knowledge Representation 1986 p44.

[Lindsay et al., 1988] P.A. Lindsay, R.C. Moore, B. Ritchie *Review of Existing Theorem Provers,* IPSE 2.5 report, Dept. of Comp. Science, University of Manchester 1988.

[Miller, 1990] D. Miller *Abstractions in Logic Programs,* Odifreddi (ed)., *Logic and Computer Science,* Academic Press, 1990.

[Milner, Gordon and Wadsworth, 1979] R. Milner, M. Gordon and C. Wadsworth *Edinburgh LCF,* Springer-Verlag, LNCS, 1979.

[Paulson, 1987] *Logic and Computation: Interactive Proof with the Cambridge LCF,* CUP, 1987.

[Thompson, 1991] *Type Theory and Functional Programming,* Addison-Wesley, 1991.