7

A Partial Mechanization of Second-order Logic

J. L. Darlington
Gesellschaft für Mathematik und Datenverarbeitung
Bonn

Abstract

An investigation into the mechanization of logical inference in higher-order functional calculi has resulted in a theorem-proving program, written in SNOBOL-4 for the RCA Spectra 70/46 and the IBM 360/50, that performs many second-order inferences in addition to carrying out first-order 'resolutions' on Skolemized disjunctive formulae. The second-order aspect of the program is represented by an extended matching procedure, which operates in conjunction with rules for lambda abstraction and application, and for existential generalization and instantiation. These rules abstract properties from first-order formulae and apply axioms such as mathematical induction to these properties, thereby generating new second-order formulae as well as first-order formulae that could not have been produced by the resolution method alone. The mechanization of second-order logic is of potential usefulness in the areas of deductive question answering, illustrated by an example, and to the proving of formal properties of programs.

Among the more interesting recent developments in the theory and practice of automatic theorem proving are the incorporation of formal techniques such as J. A. Robinson's resolution method into the 'deductive sections' of question-answering and problem-solving systems (Chadwick *et al.* 1969, Green 1969), the solution of previously 'open' problems (Guard *et al.* 1969, Allen and Luckham 1970) and the initiation of research into higher- or omega-order theorem-proving algorithms (Gould 1966, Robinson 1969). The first two developments illustrate in different ways the actual and potential usefulness of a field of research pursued hitherto largely as an abstract exercise in artificial intelligence, whilst the third stems from the recognition that existing first-order theorem provers are not powerful enough for a significant portion of mathematics and that any effort toward strengthening

their inferential capacity will no doubt pay off in extending the scope and efficiency of question-answering and other practical systems having a deductive component. An investigation into the mechanization of second-order logic, as a step toward the complete mechanization of omega-order logic, was therefore started at RCA in 1969 and continued at the GMD in Bonn, and has resulted in an experimental theorem prover, written in SNOBOL-4 for the Spectra 70/46 and the IBM 360/50, that has proved a number of theorems requiring second-order deductions and equality substitutions in addition to basic first-order resolution-type inferences.

The language of first-order or 'restricted' predicate calculus with equality, in which variables range over individual constants such as integers, is capable of expressing in a formal and precise way many statements of mathematics and of ordinary discourse, such as the following:

(a) $Prime(7)$ '7 is a prime

(b) $(Ax)(L(1, x)$ *implies* $(Ey)(Prime(y)$ & $Divisor(y, x)))$
'every x (integer) greater than 1 has a prime divisor'

(c) $(Ax)(State(x)$ *implies* $(Ey)(Ez)(y \neq z$ & $Senator(y)$ & $Represent(y, x)$ & $Senator(z)$ & $Represent(z, x)$ & $(Aw)((Senator(w)$ & $Represent(w, x))$ *implies* $(w=y \vee w=z))))$
'every state is represented by exactly two Senators'

The next six propositions on the other hand all require second-order formulations, in which certain variables (in particular, 'f') are allowed to range over first-order properties:

(d) $(Ax)(Ay)((x=y)$ *implies* $(Af)(f(x)$ *implies* $f(y)))$
'if x equals y then y has all properties of x'

(e) $(Af)((f(0)$ & $Hered(f))$ *implies* $(Ax)(f(x)))$
'any hereditary property that applies to zero applies to all integers'

(f) $(Af)((Virtue(f)$ & $f(Father(David)))$ *implies not-*$f(David))$
'David has none of his father's virtues'

(g) $(Af)((Fault(f)$ & $f(Father(David)))$ *implies* $f(David))$
'David has all of his father's faults'

(h) $(Af)(Property\text{-}of\text{-}a\text{-}great\text{-}general(f)$ *implies* $f(Napoleon))$
'Napoleon had all the properties of a great general'

(i) $(Af)(Property\text{-}of\text{-}a\text{-}great\text{-}general(f)$ *if and only if*
$(Ax)(Great\text{-}general(x)$ *implies* $f(x)))$
'a property of a great general is a property which every great general has'

A principle such as (d) is used implicitly in mathematical proofs involving reduction of a complex algebraic expression to a simpler one, as in the reduction to 0 of the expression '$(a-a) \cdot b$' by means of the equations '$(x-x)=0$' and '$(0 \cdot x)=0$'. Proposition (e) is Peano's fifth postulate for arithmetic and is usually called 'mathematical induction'. The meaning of 'hereditary' is given by the equivalence:

Hered(*f*)≡(*Ax*)(*f*(*x*) *implies f*(*s*(*x*)))
'for all *x*, if *f* is true of *x* then *f* is true of the successor of *x*'

An analogous inductive postulate, called 'transfinite induction' or the 'least number principle', results from defining 'hereditary' as follows:

Hered'(*f*)≡(*Ax*)((*Ay*)(*Precede*(*y*, *x*) *implies f*(*y*)) *implies f*(*x*))
'for all *x*, if *f* is true of all *y* that precede *x* (according to a well-ordering relation) then *f* is true of *x*'

Examples (*f*) to (*i*), taken from a textbook of Copi (1958), illustrate how statements of ordinary discourse may require second-order formulations.

An earlier paper (Darlington 1968) described a method called '*f*-matching' which allowed second-order formulae such as (d) to be incorporated into resolution-type proofs, but statements such as (e) to (i) could not easily be accommodated since the method lacked an explicit representation for properties abstracted from first-order formulae. This limitation has been overcome in the current program by employing a 'lambda' representation similar to that of Landin (1964) and Robinson (1969) together with the necessary mechanism for property abstraction and application, and for the handling of 'choice' functions, which replace existentially quantified individual variables by functions of universally quantified property variables that precede them in the prenex quantifier list. The matching procedure for literals employed by the program is based on the familiar matching or 'unification' algorithm for first-order logic (Guard *et al.* 1969, Robinson 1965), but requires in addition that variables may take as values only variables or constants of the same logical type. Thus, the variables of type 0, i.e., '*x*', 'x_1', 'x_2', etc., may take as values only other x_i, or individual constants of type 0 such as '*David*', '*Napoleon*', '*a*', '*b*', '0', '1', '2', '*Choice*(*Lambda*(*a*)(*L*(*a*, 0)))', etc., or functions of these objects, such as '*Successor*(0)' or '*Father*(*David*)'; the variables of type 1, i.e., the property variables '*f*', 'f_1', 'f_2', etc., may take as values only other f_i, or properties of individuals such as '*Great-general*', '*Prime*', '*L*' ('less than') or '*Lambda*(*a*)(*L*(*a*, 0))'; the variables of type 2, i.e., '*g*', 'g_1', 'g_2', etc., may take as values only other g_i, or properties of properties of individuals such as '*Hereditary*', '*Virtue*' or '*Lambda*(*Prime*)(*Prime*(7))'; and so on, though the program has not yet been called upon to handle objects of type higher than 2. The main difference between matching in first-order logic, where variables take only individuals as values, and in second- and higher-order logic, where variables take predicates as values, is the following: that the latter may employ lambda abstraction to match two literals which fail to match in the conventional way, if the predicate of one is a constant, the predicate of the other is a variable of the same type, and the arguments of the second literal match any subset of the arguments of the first literal, permitting the abstraction of one or more properties based on the parts of the first literal that are so matched. Thus, the two literals '*L*(*a*, *x*)' and '*not-f*(*y*, *h*(*y*, *z*))' match in the conventional way, under the value assignments '*f*=*L*', '*y*=*a*' and '*x*=*h*(*y*, *z*)=*h*(*a*, *z*)', but the

property abstraction routine must be called into play if '$L(a, s(b))$' is to match '*not-f*(x)'. The argument 'x' of the second literal matches in turn 'a', '$s(b)$' and 'b', permitting '$L(a, s(b))$' to be translated in effect into three more elaborate formulae, i.e., '*Lambda*$(a)(L(a, s(b)))(a)$', '*Lambda*$(s(b))(L(a, s(b)))(s(b))$' and '*Lambda*$(b)(L(a, s(b)))(b)$', which state, respectively, 'a' has the property of being less than '$s(b)$', '$s(b)$' has the property that 'a' is less than it, and 'b' has the property that 'a' is less than its successor; these now match '*not-f*(x)' in the conventional way, under the assignments

$$f=Lambda(a)(L(a, s(b))),\ x=a$$
$$f=Lambda(s(b))(L(a, s(b))),\ x=s(b)$$
$$f=Lambda(b)(L(a, s(b))),\ x=b$$

The matching of two literals based on property abstraction does not require the literals initially to be of opposite sign. Thus, '$L(a, s(b))$' will also match '$f(x)$' in three ways, expressed by the assignments

$$f=Lambda(a)(not\text{-}L(a, s(b))),\ x=a$$
$$f=Lambda(s(b))(not\text{-}L(a, s(b))),\ x=s(b)$$
$$f=Lambda(b)(not\text{-}L(a, s(b))),\ x=b$$

In this case, the condition of opposite sign for matching is implicitly obeyed by matching '$f(x)$' in effect with the double negation of '$L(a, s(b))$'.

The property application routine comes into play during the generation of resolvents whenever a literal beginning with a lambda expression is produced. For example, the literal '*Lambda*$(a)(L(a, s(b)))(0)$', which applies the lambda expression to '0', is automatically converted into '$L(0, s(b))$' by the simple device of replacing 'a' by '0' in the 'lambda body' of the expression.

The general case of matching between '$f(S_1, \ldots, S_m)$' and '$C(T_1, \ldots, T_n)$' is dealt with in detail by Gould (1966), who also discusses the case in which both formulae begin with functional variables. Our program handles just the special case in which $m \leqslant n$, and is primarily geared for examples in which $m=1$. In any case, the general problem is to find every lambda expression that may be substituted for 'f' such that its application to '$S_1, \ldots, S_m$' results in a formula that matches '$C(T_1, \ldots, T_n)$', as in Gould's example of matching between '$f(x, B)$' and '$C(y)$' which may be accomplished in two ways, under the value assignments

(1) $f=Lambda(u, v)(u);\ x=C(y)$
(2) $f=Lambda(u, v)(C(g(u, v)));\ y=g(x, B)$

resulting in the common instances

(1′) $C(y)$
(2′) $C(g(x, B))$.

The procedure described may be illustrated by the following examples, all of which were solved by the program on the 360 in times ranging from a few seconds to under two minutes, though these times are extremely dependent on the number of garbage collections required, which depend in turn on the size of the 'batch partitions'.

Example 1

(1) *not-GGPROP(f)* ∨ *f(Nap)*
(2) *not-GGPROP(f)* ∨ *not-GG(x)* ∨ *f(x)*
(3) *GG(Choice(f))* ∨ *GGPROP(f)*
(4) *not-f(Choice(f))* ∨ *GGPROP(f)*
(5) *not-GG(Nap)*
(6) *not-GGPROP(GG)* (1) & (5)
(7) *GG(Choice(GG))* (3) & (6)
(8) *not-GG(Choice(GG))* (4) & (6)
(9) *contradiction* (7) & (8)

This example is a proof that Napoleon was a great general on the basis of (h) and (i), in which formula (1) is the Skolemized form of (h), (2) – (4) correspond to (i) and (5) is the negation of the conclusion. None of the matches involved require the property abstraction routine, so the property of being a great general is represented simply by '*GG*' instead of by the more elaborate lambda expression, '*Lambda(Nap)(GG(Nap))*'. In the actual operation of the program, the different choice expressions are automatically replaced by '*1', '*2', etc. as they are generated, resulting in a considerable reduction in the length of formulae where the properties involved are lambda expressions, as in the following examples.

Example 2

(1) *not-L(x, x)*
(2) *L(x, s(x))*
(3) *not-L(x, y)* ∨ *not-L(y, z)* ∨ *L(x, z)*
(4) *not-f(0)* ∨ *not-Hered(f)* ∨ *f(x)*
(5) *Hered(f)* ∨ *f(Choice(f))*
(6) *Hered(f)* ∨ *not-f(s(Choice(f)))*
(7) *L(a, 0)*
(8) *L(0, 0)* ∨ *not-Hered(Lambda(a)(not-L(a, 0)))* (4) & (7)
(9) *not-Hered(Lambda(a)(not-L(a, 0)))* (1) & (8)
(10) *not-L(*1, 0)* (5) & (9)
(11) *L(s(*1), 0)* (6) & (9)
(12) *not-L(*1, y)* ∨ *not-L(y, 0)* (3) & (10)
(13) *not-L(s(*1), 0)* (2) & (12)
(14) *contradiction* (11) & (13)

In this example, '*L*' is the relation of 'less than' and '*s*' stands for 'successor' Formula (4) is the Skolemized version of the inductive postulate (e), and (5) and (6) express part of the definition of 'hereditary'. The program automatically makes the substitution

Choice(Lambda(a)(not-L(a, 0))) = *1

in the generation of clauses (10) – (14).

The next example involves an equality substitution, and also shows how two or more properties may be abstracted from two or more separate clauses and disjunctively combined within the context of an inductive proof.

Example 3

(1) *not-L*(x, x)
(2) $L(x, s(x))$
(3) *not-L*$(x, y) \vee$ *not-L*$(y, z) \vee L(x, z)$
(4) $EQ(x, x)$
(5) *not-*$f_1(0) \vee$ *not-Hered*$(f_1, f_2) \vee f_1(x) \vee f_2(x)$
(6) *Hered*$(f_1, f_2) \vee f_1($*Choice*$(f_1, f_2)) \vee f_2($*Choice*$(f_1, f_2))$
(7) *Hered*$(f_1, f_2) \vee$ *not-*$f_1(s($*Choice*$(f_1, f_2)))$
(8) *Hered*$(f_1, f_2) \vee$ *not-*$f_2(s($*Choice*$(f_1, f_2)))$
(9) *not-L*$(0, b)$
(10) *not-EQ*$(0, b)$
(11) *not-EQ*$(0, 0) \vee$ *not-Hered*$($*Lambda*$(b)(EQ(0, b)), f_2) \vee f_2(b)$
(5) third literal & (10)
(12) *not-Hered*$($*Lambda*$(b)(EQ(0, b)), f_2) \vee f_2(b)$ (4) & (11)
(13) *not-Hered*$($*Lambda*$(b)(EQ(0, b)),$ *Lambda*$(b)(L(0, b)))$
(9) & (12) second literal
(14) $EQ(0, *2) \vee L(0, *2)$ (6) & (13)
(15) *not-EQ*$(0, s(*2))$ (7) & (13)
(16) *not-L*$(0, s(*2))$ (8) & (13)
(17) *not-L*$(0, s(0)) \vee L(0, *2)$ (14) & (16)
(18) *not-L*$(0, y) \vee$ *not-L*$(y, s(*2))$ (3) & (16)
(19) L$(0, *2)$ (2) & (17)
(20) *not-L*$(0, *2)$ (2) & (18)
(21) *contradiction* (19) & (20)

The expression '(f_1, f_2)' denotes the disjunctive property of being either f_1 or f_2. Formulae (5) – (8) above express the inductive postulate and the definition of 'hereditary' appropriate to '(f_1, f_2)'. Formula (5) is derived from (4) in the preceding example by replacing 'f' by '(f_1, f_2)', expanding '*not-*$(f_1, f_2)(0)$' into '*not-*$f_1(0)$ & *not-*$f_2(0)$' and '$(f_1, f_2)(x)$' into '$f_1(x) \vee f_2(x)$', and putting the result into conjunctive normal form thereby obtaining two clauses, the second of which, based on '*not-*$f_2(0)$', duplicates the sense of the first [i.e., (5)] and is not included in the axiom set. The program makes the substitution

Choice$($*Lambda*$(b)(EQ(0, b)),$ *Lambda*$(b)(L(0, b))) = *2$

in the generation of clauses (14) – (21). In the deduction of (17) from (14) and (16), the equality clause (14) is temporarily replaced by

(14′) *not-*$f(*2) \vee f(0) \vee L(0, *2)$

implicitly making use of the equality substitution axiom (d) stated earlier, and (14′) is then resolved with (16). This method treats an equality inference as just a special type of second-order inference, while at the same time avoiding the generation of large numbers of intermediate clauses that result from the unrestricted use of formula (d). It will be noted that (14′) is actually based on '$EQ(*2, 0)$' rather than '$EQ(0, *2)$'. This is because the program gives preference to reductive equality inferences, in the sense that it

tries to employ equality substitutions in such a way that formulae are reduced to simpler ones. This sort of switch is justified by the symmetry of the equality relation, which follows from the reflexive property and formula (d).

The next example is of a rather different sort from the preceding three, and shows the possibility of applying second-order logic to question-answering or problem-solving situations *via* an extension of Green's (1969) answer predicate ANS.

Example 4

(1) *not-Virtue*(f) ∨ *not-f*(*Father*(*David*)) ∨ *not-f*(*David*)
(2) *Kind*(*David*, *Animals*)
(3) *Virtue*(*Lambda*(x)(*Kind*(x, *Animals*)))
(4) *not-f*(*Father*(*David*)) ∨ ANS(f)
(5) *not-Virtue*(*Lambda*(*David*)(*Kind*(*David*, *Animals*))) ∨ *not-Kind*(*Father*(*David*), *Animals*) (1) & (2)
(6) *not-Kind*(*Father*(*David*), *Animals*) (3) & (5)
(7) ANS(*Lambda*(*Father*(*David*))(*not-Kind*(*Father*(*David*), *Animals*))) (4) & (6)

Clause (1) says [as does the earlier formula (f)] that David has none of his father's virtues, (2) that David is kind to animals, (3) that kindness to animals is a virtue, and (4) asks for the properties of David's father. The answer, clause (7), is the property of not being kind to animals.

The next example, taken from Robinson (1969), is the only theorem employing variables of type higher than 1 that the program has so far been applied to. The problem is to show that the two formulae '(*Af*)(*P*(*f*) *implies* *f*(*b*))' and '(*Ag*)(*g*(*P*) *implies* *g*(*Q*))' together entail '(*Af*)(*Q*(*f*) implies *f*(*b*))' where '*b*' is an individual constant, '*f*' is as before a variable of type 1, '*g*' is a variable of type 2, and '*P*' and '*Q*' are properties of type 2. The negation of the conclusion is the assumption that there is some property of type 1, e.g., '*R*', that has property '*Q*' but does not apply to '*b*'.

Example 5

(1) *not-P*(f) ∨ *f*(*b*)
(2) *not-g*(*P*) ∨ *g*(*Q*)
(3) *Q*(*R*)
(4) *not-R*(*b*)
(5) *not-P*(*R*) (1) & (4)
(6) *not-Q*(*R*) (2) & (5)
(7) *contradiction* (3) & (6)

In the deduction of (6) from (2) and (5), the argument '*P*' of the literal '*not-g*(*P*)' matches the predicate, rather than the argument, of the literal '*not-P*(*R*)', resulting in the assignment '*g*=*Lambda*(*P*)(*not-P*(*R*))', and this property is applied to '*Q*', according to the second literal of (2), producing '*not-Q*(*R*)'.

For the sake of simplicity, the property abstraction routine has so far been applied only to 'ground' literals, but this is not an essential restriction since properties may be abstracted not only from literals containing free variables but also from entire clauses containing several literals, as in the solution of Example 5 by 'forward deduction', which requires the abstraction of the property '*Lambda*(*P*)(*Af*)(*P*(*f*) *implies f*(*b*))' from the first premiss and its subsequent application to 'Q'. Furthermore, a whole new set of properties may be generated even from ground literals if one allows lambda abstraction to operate in conjunction with existential generalization. For example, in addition to the three properties abstracted earlier on from '*L*(*a*, *s*(*b*))', there are the three further properties '*Lambda*(*a*)(*Ex*)(*L*(*a*, *x*))', '*Lambda*(*s*(*b*))(*Ex*)(*L*(*x*, *s*(*b*)))' and '*Lambda*(*b*)(*Ex*)(*L*(*x*, *s*(*b*))', which apply respectively to '*a*', '*s*(*b*)' and '*b*', stating in effect that '*a*' has the property of being less than some integer, that '*s*(*b*)' has the property of having some integer less than it, and that '*b*' has the property of having some integer less than its successor. To test the consequences of lambda abstraction with existential generalization, clause (7) in Example 2 was replaced by the assumption that some integer is less than some other integer. Among the clauses generated from these premises was the 'theorem' that the property of being the smallest integer is not hereditary: this part of the deduction is given below.

Example 6

(1) *not-L*(*x*, *x*)
(2) *L*(*x*, *s*(*x*))
(3) *not-L*(*x*, *y*) ∨ *not-L*(*y*, *z*) ∨ *L*(*x*, *z*)
(4) *not-f*(0) ∨ *not-Hered*(*f*) ∨ *f*(*x*)
(5) *Hered*(*f*) ∨ *f*(*Choice*(*f*))
(6) *Hered*(*f*) ∨ *not-f*(*s*(*Choice*(*f*)))
(7) *L*(*a*, *b*)
(8) *L*(*1, 0) ∨ *not-Hered*(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (4) & (7)
(9) *L*(0, 0) ∨ *not-Hered*(*Lambda*(*1)(*not-L*(*1, 0))) ∨
not-Hered(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (4) & (8)
(10) *not-Hered*(*Lambda*(*1)(*not-L*(*1, 0))) ∨
not-Hered(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (1) & (9)
(11) *not-L*(*13, 0) ∨ *not-Hered*(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (5) & (10)
(12) *L*(*s*(*13), 0) ∨ *not-Hered*(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (6) & (10)
(13) *not-L*(*13, *x*) ∨ *not-L*(*x*, 0) ∨
not-Hered(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (3) & (11)
(14) *not-L*(*s*(*13), 0) ∨ *not-Hered*(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (2) & (13)
(15) *not-Hered*(*Lambda*(*b*)*not-*(*Ex*)(*L*(*x*, *b*))) (12) & (14)

In the deduction of (8) from (4) and (7), '$L(a, b)$' matches '$f(x)$' under the value assignments

$$f=Lambda(b)not\text{-}(Ex)(L(x, b)))$$
$$x=b$$

In other words, '$L(a, b)$' is interpreted in this case as denying to 'b' the property of having no integer less than itself. The first literal of (4) then denies this property to '0', that is, applies the property

$$not\text{-}f=Lambda(b)(Ex)(L(x, b))$$

to '0', and in the process instantiates 'x' by '*1', and the induction principle (4) is applied again to '$L(*1, 0)$', though this time without existential generalization. The rest of the deduction involves no further lambda abstractions.

It is obvious that a complete method for second-order logic, if such exists, would have to take into account more than the limited set of properties which the program is at present capable of abstracting, but it is questionable whether a program could satisfactorily cope with a total set of logically possible properties. In addition to the problem of completeness, there is that of the relation between higher-order logic and Amarel's (1968, 1971) work on problem representation and, in particular, the extent to which one can automate the process of finding or constructing a set of predicates of whatever type that lead to an optimum description and solution of a given problem.

Acknowledgement

The research presented in this paper was partially supported by the Air Force Office of Scientific Research of the Directorate of Information Sciences under contract No F44620-68-C-0012.

REFERENCES

Amarel, S. (1968) On representations of problems of reasoning about actions. *Machine Intelligence 3*, pp. 131–71 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Amarel, S. (1971) On representations of problems of program formation. *Machine Intelligence 6*, paper no. 25 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Allen, J. & Luckham, D. (1970) An interactive theorem-proving program. *Machine Intelligence 5*, pp. 321–36 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Chadwick, J.H., Coles, L.S., Jones, J.H., Raphael, B. & Whitby, O.W. (1969) Medical applications of remote electronic browsing. *SRI Project 7382.*

Copi, I.M. (1958) *Symbolic Logic.* New York: Macmillan.

Darlington, J.L. (1968) Automatic theorem-proving with equality substitutions and mathematical induction. *Machine Intelligence 3*, pp. 113–27 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Gould, W.E. (1966) A matching procedure for omega-order logic. *Sci. Rep. No. 4. AFRCL 66–781.* Princeton, New Jersey: Applied Logic Corporation.

Green, C.C. (1969) The application of theorem proving to question-answering systems. Ph.D. dissertation, Stanford University. *SAIP Memo AI–96.*

Guard, J.R., Oglesby, F.C., Bennett, J.H. & Settle, L.G. (1969) Semi-automated mathematics. *J. Ass. comput. Mach.* **16**, 49–62.

Landin, P.J. (1964) The mechanical evaluation of expressions. *Comput. J.* **6**, 308–20.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.* **12**, 23–41.

Robinson, J.A. (1969) Mechanizing higher-order logic. *Machine Intelligence 4*, pp. 151–70 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.