
ABSYS: THE FIRST LOGIC PROGRAMMING LANGUAGE —A RETROSPECTIVE AND A COMMENTARY

E. W. ELCOCK

- ▷ In the research literature, logic programming, as a procedural interpretation of SLD resolution, has largely been associated with developments arising from the interaction of Colmerauer and Kowalski and their colleagues in the early seventies. Around 1967 the Group for Computing Research at the University of Aberdeen designed and implemented a programming system called Absys. It should be interesting to the logic programming community that Absys was a logic programming language in the full current sense of that descriptor, and the first such programming language. This claim is not intended to be aggressive or territorial (indeed, the current PROLOG “phenomenon” is certainly not of our causing and not something to which we would lay claim). Rather, it is hoped that logic programmers might be interested to hear how subsequent developments in what is now called equational programming, and alternative presentations of the unification algorithm, allow Absys to be recognized for what it was. ◁
-

1. PREAMBLE

“Mine is a long and sad tale!” said the Mouse, turning to Alice, and sighing.

“It is a long tail, certainly” said Alice, looking down with wonder at the Mouse’s tail; “but why do you call it sad?” and she kept on puzzling about it while the Mouse was speaking....

The descriptor *logic programming* is usually identified with Kowalski’s [17] procedural interpretation of SLD resolution [21], a refutation procedure for definite clauses, in turn based on the resolution refutation procedure for sets of sentences in clausal form [25, 26]. It is not a coincidence that *logic programming* is¹ a translation of *programmation en logique*, which in turn led to the acronym *Prolog* (which is certainly a better acronym than *Logpro*).

Address correspondence to E. W. Elcock, Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7

Received December 1987; accepted August 1988.

¹Folklore?

The roots of *programmation en logique* drew their nourishment from Colmerauer's interest in natural language systems [2]. The first implementation of a version of PROLOG would seem to have been developed in Marseilles around 1972 [2], to be followed in 1973 by a more efficient implementation [27]. The developments in the early seventies are presented in a recent paper by Kowalski [18] as the story of a fascinating interaction between research groups in Edinburgh and Marseilles. Towards the end of this account Kowalski talks about the relationship between the emerging concept of logic programming and the work of Hayes [12] on the notion of *computation as controlled deduction*. He refers to Hayes's work as being

influenced by Absys,² a declarative programming language developed at the University of Aberdeen and reported in a number of papers in the *Machine Intelligence* series [6, 10, 11]. Absys anticipated a number of Prolog features such as "invertability", "negation by failure", "aggregation operators" and the central role of backtracking.

Certainly, like Hayes, my colleagues and I were interested in the eventual possibility of separate declarative and control language components. However, in the context of this paper, the important phrase in the quotation from Kowalski is the remark that "Absys anticipated a number of Prolog features". This turned out to be hardly surprising. The precise relation between Absys and PROLOG has tantalized the author for many years. It seemed that anything expressible in PROLOG could be transparently mapped into Absys. Happily, recent work on unification as equation solving [20, 22], together with the use of this framework in discussing resolution strategies [30], now makes it clear that a "pure" subset of Absys did not just "anticipate" a "number of features of" but was indeed an implementation of pure (completed) PROLOG. Indeed as far as the author is aware, the programming system Absys, essentially completed in 1967 [11, 7], was the first design and implementation of a logic programming language in the sense identified in the first paragraph above.

When published, Absys was referred to as a *system for processing assertions*.³ We shall see that Absys was a language firmly rooted in logic and anticipated many of the concepts of logic programming rediscovered several years later. The following list identifies the more important of these:

SLD resolution: SLD resolution, on which most (and certainly the early) PROLOG interpreters were based, is SL resolution [16] restricted to Horn clause logic programs [17]. Kowalski's formulation of the concept of SLD resolution dates from the early seventies. Soundness and completeness results were given by Hill [13]. The Absys (circa 1967) refutation procedure can certainly be regarded as an implementation of SLD resolution. Indeed, it would seem reasonable to claim that SLD refutations were first demonstrated in Absys.

Solving sets of equations. Absys implemented SLD resolution in the framework of solving systems of equations [22, 30, 20]. This framework has provided a powerful conceptual tool for work on extending syntactic to semantic unification (see for example [15]). It is also worth remarking that this is essentially

²An acronym for *Aberdeen System*.

³The descriptor *logic programming* had not been coined.

the route taken by Colmerauer in the late seventies in his development of PROLOG II [3, 4] considered as a system for solving sets of equations over the domain of infinite rational trees.

Less central but also worthy of remark are the following:

The computation rule. The concept of fairness [19] was recognized in Absys, and Absys used a fair computation rule.

Negation. Negation was handled in Absys by augmenting SLD resolution with a negation-as-failure rule [1]. Absys used program completions [1].

Delay mechanisms. Both primitive and user declared delay constructs were used in Absys to handle the same kind of problems which led to their introduction in PROLOG contexts by Colmerauer [4] and Naish [24].

Aggregation operators. A *set-of* aggregation operator was implemented in Absys-4 (circa 1968).

Constraint solving. Although simplistic compared to current work [15], Absys was created with the constraint programming paradigm in mind, and its design shows clear evidence of this. Certainly what is referred to in [15] as “solving constraints by local propagation” [28] and “runtime rearrangement of equations” was anticipated in the Absys work (see for example [8]), where the concept was included under the broader notion of “data directed control”.

The paper continues with a general introduction which is intended to set the stage for a “modern” redescription of Absys in the framework of solved forms of sets of equations, enabling a revealing comparison with PROLOG and with PROLOG II. This redescription and comparison follows. Finally a detailed discussion of the treatment of equality in Absys is given, fully justifying the claim made in the title of the paper.

In order to make the present paper easily readable, some slight liberties have been taken with the original syntax of the Absys text. It should also be stressed that what is referred to here as “Absys” is a subset of the language Absys-4, circa 1968. A description of the full language, which contained higher order features and simple control mechanisms, is to be found in [9]. An attempt has been made to give enough, but just enough, background to make the paper stand alone.

2. GENERAL INTRODUCTION

Standard PROLOG is usually described as follows. A PROLOG program P is a sequence of positive definite Horn clauses. We are interested in SLD refutations of $P \cup \{G\}$, where G is a negative clause, the “goal” clause. The PROLOG interpreter explores the space of SLD derivations using a “leftmost goal literal” computation rule [21] and a search rule based on the textual ordering of the sequence of program clauses. Of particular computational interest are the compared answer substitutions [21] associated with found refutations.

A central computational notion of a PROLOG program is that of a “procedure”. A PROLOG procedure is just the subsequence of program clauses with the same predicate letter for the positive atom. This set of clauses is said to *define* the procedure. This notion is at the heart of the motivation of the procedural semantics

for PROLOG refutations and is the notion which forms the link with more conventional programming paradigms.

Absys is most easily and naturally described (in the sense of closeness to its computational paradigm) within the framework of what is now referred to as *equational programming*. In order to facilitate the comparison with PROLOG, we will redescribe PROLOG within just such a framework. To allow this redescription to stand alone, we will make brief digressions—to describe the “homogeneous form” of a PROLOG program and the so-called “general procedure” [29], and to describe the relationship between unification, resolution strategies, and solving sets of equations [20, 30].

2.1. Unification and Equation Solving

The following is largely taken from [20]. An equation set E is in *solved form* if it has the form $\{v_1 = t_1, \dots, v_n = t_n\}$ where the variables v_i , $1 \leq i \leq n$, do not appear in the right hand sides of any equation. The variables v_1, \dots, v_n are the *dependent* (“eliminable”) variables. The remaining variables are the *independent* variables or “parameters”. If E is in solved form, then a (ground) solution for E is specified by substituting any ground terms for the parameters, and conversely.

Equation sets E and E' are said to be *equivalent* if they have the same set of solutions. The following *solved form algorithm* (based on Herbrand’s original unification algorithm: see [20]) transforms a soluble set of equations E into an equivalent set E' in solved form.⁴

Given an equation set E , nondeterministically choose an equation and obey an applicable rule from the list:

1. If the equation is of the form $f(t_1, \dots, t_n) = f(s_1, \dots, s_n)$, replace it by the equations $(t_1 = s_1, \dots, t_n = s_n)$.
2. If the equation is of the form $f(t_1, \dots, t_n) = g(s_1, \dots, s_m)$ where $f \neq g$, halt with failure.
3. If the equation is of the form $x = x$, delete the equation.
4. If the equation is of the form $t = x$, where t is not a variable, replace it by $x = t$.
5. If the equation is of the form $x = t$ where $t \neq x$ and x has another occurrence in the set of equations:
if x appears in t then halt with failure;
otherwise replace x by t in every other equation.

The algorithm terminates when no step is applicable or when failure has been returned.

As an example of an application of the algorithm consider the equation set

$$f(h(x), z) = f(y, g(y)), \quad z = g(h(x)).$$

Choosing the last equation and using rule 5, we get

$$f(h(x), g(h(x))) = f(y, g(y)), \quad z = g(h(x)).$$

⁴The algorithm is also presented in [22] as a starting point for the development of an efficient unification algorithm.

Choosing the first equation and using rule 1, we get

$$h(x) = y, \quad g(h(x)) = g(y), \quad z = g(h(x)).$$

Choosing the first equation and using rule 4, we get

$$y = h(x), \quad g(h(x)) = g(y), \quad z = g(h(x)).$$

Choosing the second equation and using rule 1, we get

$$y = h(x), \quad h(x) = y, \quad z = g(h(x)).$$

Choosing the first equation and using rule 5, we get

$$y = h(x), \quad h(x) = h(x), \quad z = g(h(x)).$$

Finally, choosing the second equation and using rules 1 and 3, we get

$$y = h(x), \quad z = g(h(x)).$$

No more rules apply, and the set of equations is in solved form with parameter x and eliminable variables y, z .

The following result (analogous to Robinson's unification theorem [25]) is proved in [20].

Theorem 2.1. The solved form algorithm applied to a set of equations E will return an equivalent set of equations E' in solved form if E is solvable. It will return failure otherwise.

There is an obvious mapping $\{v_1 = t_1, \dots, v_n = t_n\} \rightarrow \{v_1/t_1, \dots, v_n/t_n\}$ from solved forms to idempotent substitutions. It is shown in [20] that:

Statement 2.1. The problem of finding an idempotent substitution which is a most general unifier (mgu) of two atoms $P(t_1, \dots, t_n)$ and $P(s_1, \dots, s_n)$ becomes just the problem of finding the substitution derived from a solved form of the equation set $E = \{t_1 = s_1, \dots, t_n = s_n\}$.

It is convenient to regard this last equation set as derived from the equation $P(t_1, \dots, t_n) = P(s_1, \dots, s_n)$ by an obvious extension of rule 1 of the solved form algorithm. Thus if the two atoms are $P(f(h(x), z), z)$ and $P(f(y, g(y)), g(h(x)))$, then the equation set to be solved is that of the example above. An mgu obtained from the solved form derived above is $\{y/h(x), z/g(h(x))\}$.

2.2. Resolution Strategies and Equation Solving

Finally we need a link between equation solving and resolution derivations. We take what we need from the work of Wolfram et al. [30].

Let P , G , and R be a program, goal, and computation rule respectively. An SLD derivation of $P \cup \{G\}$ using R is a sequence of goals $G = G_0, G_1, \dots$, such that there exist variants C_1, C_2, \dots of program clauses of P and a sequence $\theta_1, \theta_2, \dots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using R as follows.

Let G_i be $\leftarrow A_1, \dots, A_m, \dots, A_k$, and C_{i+1} be $A \leftarrow B_1, \dots, B_q$. Let A_m be the goal atom selected by R . A derivation of G_{i+1} from G_i and C_{i+1} is possible if A and A_m

are unifiable with mgu θ_{i+1} when

$$G_{i+1} \equiv \leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k) \theta_{i+1}.$$

An SLD refutation of length n is a finite derivation of length n for which G_n is the empty goal. The answer substitution is obtained from the composition of mgu $\theta_1 \theta_2 \dots \theta_n$.

There are several points of interest. We have noted that the mgu of A and A_m can be derived from a solution of the equation $A = A_m$. In the definition of an SLD refutation above, this mgu would be applied immediately to G_i to derive G_{i+1} before going on to the next deduction step. Wolfram et al. [30] have pointed out that this “immediacy” can be regarded as an incidental feature of the resolution strategy. Each SLD refutation has its associated *set* of equations—one for each unification in the refutation. This set can be obtained by replacing the standard formulation of SLD refutation with one in which unifiers appear only implicitly as the yet unprocessed equations.⁵ We redescribe an SLD refutation as follows.

A goal is a pair $\langle G, E \rangle$ where G is a set of atoms and E is a set of equations. The initial goal is $\langle G, \emptyset \rangle$. The derivation step

$$G_{i+1} \equiv \leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k) \theta_{i+1}$$

in the standard formulation, where θ_{i+1} is an mgu of A_m and A , is replaced by the step

$$\langle G_{i+1}, E_{i+1} \rangle \equiv \left\langle \leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k), E_i \cup \{A = A_m\} \right\rangle$$

where no substitution is applied to the set of atoms. A derivation is a refutation of length n if it terminates with G_n empty and E_n a soluble set of equations. The set E_n is the associated set of equations. The answer substitution for the SLD refutation can now be obtained simply as the solution of the associated set of equations E . The solution process, as we have seen, proceeds by selecting an equation at each step to which a rule of the solved form algorithm applies and using that rule to generate an equivalent set of equations. The solved form obtained as the final equation set is independent of the particular equation selected at any solution step. Wolfram et al. use this result to bypass the complexities of standard proofs of the “independence of computation rule” in the standard theory of SLD refutations. The result however is of particular interest here because, as we shall see below, although Absys normally processes “unification equations” as they arise, occasionally completion of this processing is, for good reasons, deferred. The results above justify this.

2.3. The Homogeneous Form and General Procedure

Given a PROLOG program clause c :

$$P(t_1, \dots, t_n) \leftarrow B,$$

its homogeneous form⁶ is c' :

$$P(x_1, \dots, x_n) \leftarrow x_1 = t_1, \dots, x_n = t_n, B,$$

where the variables x_1, \dots, x_n do not appear in the original clause c .

⁵A treatment of this notion in the context of general linear resolution schemes is given by Cox [5]. In particular, the set of equations associated with the SLD derivation is an instance of what Cox calls the constraint set associated with a deduction plan.

⁶Called the “general form” by Clark [1].

The homogeneous form of a PROLOG program P is the set of clauses $P' = \{c' : c \in P\}$. We have the result [29]:

$$P \cup \{G\} \text{ is unsatisfiable} \quad \text{iff} \quad P' \cup \{G\} \cup \{x = x\} \text{ is unsatisfiable.}$$

That is, $P \cup \{G\}$ is unsatisfiable iff $P' \cup \{G\}$ is unsatisfiable in the context of the equality theory $\forall x \, x = x$.

Showing that $P' \cup \{G\} \cup \{x = x\}$ is unsatisfiable may be reduced to equation solving by using the general procedure [29]. One simply uses SLD resolution and a computation rule which selects a goal literal for elaboration until a goal without literals is reached. The goal is now a set of equations E of the form $s_i = t_i$, $1 \leq i \leq n$, where s_i and t_i are terms. We now "solve" the equation set by selecting equations for unification with the clause $\leftarrow x = x$ of the associate equality theory until the empty goal is reached.

2.4. PROLOG—an Alternative Description

It should be clear from the discussion above that the unification of the equations $t_i = s_i$ of E resulting from the general procedure, each with the equation $x = x$ of the associated equality theory, is equivalent to finding the solved form of the equation set $\{\dots, x_i = t_i, x_i = s_i, \dots\}$. Using rule 5 of the solved form algorithm, this is equivalent to finding the solved form of $\{\dots, x_i = t_i, t_i = s_i, \dots\}$.

Further, since the variants of x introduced by repeated unifications with $x = x$ are of no interest in any answer substitution, it follows that in the general procedure discussed above, we can remove all operational reference to the equality theory $\{x = x \leftarrow\}$ and simply use the solved form algorithm to find the solved form of the equation set resulting from the general procedure.

This is essentially what happens in the Absys system.

Note that we could have reached this last equational view of PROLOG by *direct* appeal to the solved form algorithm as a unification algorithm and to the view of SLD resolution presented in Section 2.2. The reference to the homogeneous form and to the general procedure is not essential. The introduction of the homogeneous form however certainly acts as a nice bridge between the (nonequational) syntax of standard PROLOG and the (equational) syntax of Absys used below.

2.4.1. A (Classical) PROLOG Example. Consider the PROLOG program $P = \langle c_1, c_2 \rangle$, where

$$c_1: \text{app}([], L, L).$$

$$c_2: \text{app}([X|L], M, [X|N]) \leftarrow \text{app}(L, M, N).$$

The homogeneous form is $P' = \langle c'_1, c'_2 \rangle$, where

$$c'_1: \text{app}(R, S, T) \leftarrow R = [], S = L, T = L.$$

$$c'_2: \text{app}(R, S, T) \leftarrow R = [X|L], S = M, T = [X|N], \text{app}(L, M, N).$$

The following is a refutation of the goal $\text{app}([1], U, [1, 2])$ using the method described above. We first use the general procedure to obtain the set of equations.

Resolving the goal atom against a variant of c'_2 , we get

$[1] = [X|L]$, $U = M$, $[1, 2] = [X|N]$, $app(L, M, N)$; on resolving the remaining atom against a variant of c'_1 , we get

$[1] = [X|L]$, $U = M$, $[1, 2] = [X|N]$, $L = []$, $M = L_1$, $N = L_1$.

We now use the solved form algorithm to solve this set of equations.

A sequence of solution steps is:

$1 = X$, $[] = L$, $U = M$, $[1, 2] = [X|N]$, $L = []$, $M = L_1$, $N = L_1$ by rule 1;

$X = 1$, $L = []$, $U = M$, $[1, 2] = [X|N]$, $L = []$, $M = L_1$, $N = L_1$ by rule 4;

$X = 1$, $L = []$, $U = M$, $[1, 2] = [1|N]$, $[] = []$, $M = L_1$, $N = L_1$ by rule 5;

$X = 1$, $L = []$, $U = M$, $N = [2]$, $M = L_1$, $N = L_1$ by rule 1;

$X = 1$, $L = []$, $U = [2]$, $N = [2]$, $M = [2]$, $L_1 = [2]$ by assorted rules.

The last goal is now in solved form and contains the computed answer substitution $U/[2]$.

3. ABSYS: A SIMPLIFIED ACCOUNT

Absys text consists of a conjunction of assertions. The assertions may be either procedure definitions or literals to be satisfied: the intended interpretation is that the conjunction of the asserted literals is a logical consequence of the conjunction of asserted procedure definitions. In current usage, the conjunction of procedure definitions would be called the “program”, and the conjunction of literals the “goal”.

The syntax of procedure definitions is borrowed from the lambda calculus. For example, “append” might typically be defined in Absys by the following procedure:⁷

$app = \text{lambda } R, S, T$
 $\{ R = [], S = T \}$ or
 $\{ R = [X|L], T = [X|N], app(L, S, N) \}$

The body of the lambda expression is in disjunctive normal form. The bound variables R, S, T of the expression are to be taken as universally quantified, and the local variables X, L , and N of the second disjunction of the body of the lambda expression as existentially quantified. Indeed, the definition of app could be given the reading.

$\forall R, S, T \text{ } app(R, S, T) \leftrightarrow$
 $\{ R = [], S = T \}$ or
 $\exists X, L, N \text{ s.t. } \{ R = [X|L], T = [X|N], app(L, S, N) \}$

Variables in a goal literal are read as existentially quantified. A goal literal such as

$app([1], U, [1, 2])$

⁷“ app ” could have been defined with the additional equations to make it mimic the PROLOG homogeneous form completely—but this would be patently silly.

is treated as an application of the lambda expression (procedure) *app* and leads to the goal literal being replaced by

$$\{[1] = [], U = [1, 2]\} \text{ or } \\ \{[1] = [X_1|L_1], [T_1 = [X_1|N_1]], \text{app}(L_1, U, N_1)\}$$

The *or* is distributed through the goal conjunct in the computation in which the application takes place, essentially creating two new goals which can notionally be regarded as initiating parallel derivations. In the example, the first conjunct is unsatisfiable and any derivation (computation) containing it will fail (abort).

The operational semantics of *or* makes this definition of *app* have the effect of the homogeneous form of the two clause definition of *append* in standard PROLOG given above. The example refutation below should make this quite clear.

It should however be noted here that the Absys definition, with its associated *iff* reading, is just what Clark [1] calls the *disjunctive definition*⁸ of *append*, used by Clark in his notion of program completion as a theoretical framework for justifying negation as failure. This point will be elaborated in the section on negation in Absys below.

3.1. A (Classical) Absys Example

As mentioned, with a minor difference discussed below, the solved form algorithm is used for equation solving in Absys. However, rather than elaborating all goal atoms before beginning equation solving (as in the general procedure used in the "PROLOG" example above), Absys interleaves equation solving and procedure application. Further, the ongoing equation solving establishes bindings which are immediately used.

With *append* defined as above, and ignoring failed derivations as in the PROLOG example, elaboration of the goal atom *app*([1], *U*, [1, 2]), regarded as an application of the lambda expression, might⁹ give the following refutation:

Applying *app*, we get

$$[1] = [X_1|L_1], [1, 2] = [X_1|N_1], \quad \text{app}(L_1, U, N_1).$$

The first equation gives

$$1 = X_1, [] = L_1, \quad [1, 2] = [X_1|N_1], \quad \text{app}(L_1, U, N_1) \text{ by rule 1.}$$

The first two equations give rise to the bindings $X_1/1, L_1/[]$, and the third equation now gives (rule 1) the new goal

$$1 = 1, \quad [2] = N_1, \quad \text{app}(L_1, U, N_1).$$

The first equation is deleted by rule 1, and the second adds a binding $N_1/[2]$, giving a binding environment

$$X_1/1, \quad L_1/[], \quad N_1/[2]$$

and the goal *app*([], *U*, [2]).

⁸Called the *completed definition* in [21].

⁹Absys does not guarantee a particular order of processing of conjuncts.

Elaboration of the literal by application of the *app* procedure gives

$$[] = [], \quad U = [2].$$

The first equation is deleted by rule 1, and the second adds the binding $U/[2]$ to give the binding environment

$$X_1/1, \quad L_1/[], \quad N_1/[2], \quad U/[2],$$

i.e., the substitution derived from the complete solved form and containing the computed answer substitution $U/[2]$.

3.2. A Simplified Conclusion

Finally, to close this simple account it remains to say that, as illustrated in the example Absys refutation given above, a conjunct of equations over lists in Absys is entailed by the “program” just in case a solved form can be constructed.¹⁰ The answer substitution provided by the solved form can of course be queried.¹¹

From what has been said earlier, we see that Absys provided an implementation of SLD resolution. A minor difference between Absys and PROLOG is that the only functor in Absys is the list constructor: it was not possible in Absys to introduce general terms, with the result that the Herbrand universe was simply a set of lists.¹² A noteworthy difference is that the implementation of Absys used a “fair” [19] search rule, in that *all* derivations continue to be incrementally elaborated until found unsatisfiable.

4. SOME DETAILS AND A CAVEAT

Discussions of the implementation of Absys are given in [8, 9]. For our present purpose—the claim that Absys was a logic programming language—the essential notion is the Absys treatment of equality.

It is important to understand the concept of an Absys identifier. When created, an identifier is associated with a reference to a data structure which is typed as an uninstantiated variable. This structure is designed to hold a list of references to suspended processes which in turn reference the identifier, and typically are such that they require the identified variable to be instantiated for completion of the process.

Equality is implemented as a process which takes two references as arguments.

If the references are to lists, then the process in effect spawns equality processes between the heads and tails of the lists—cf. rule 1 of the solved form algorithm.

If the references are to the same constant or to the same variable, then the process simply succeeds—cf. rules 1 and 3 of the solved form algorithm.

If the references are to a list and a constant, or to different constants, the process fails and terminates the computation of which it is a constituent—cf. rule 2.

¹⁰See, however, the caveat discussed below.

¹¹Thus, the example refutation would cause the output “assertion” *tout*(“ U ”, U) to produce “ $U = [2]$ ”.

¹²The difference is called “minor” because one can always use the simple device of writing terms as lists by writing $[f, t_1, \dots, t_n]$ instead of $f(t_1, \dots, t_n)$ —see for example [23].

If the references are to uninstantiated variables x and y where $x \neq y$, then the process is “suspended” and associated with both x and y using the data structures they reference, as indicated above. This constitutes a conceptual variant, discussed below, of rule 5 of the solved form algorithm.

If the references are to an uninstantiated variable x and a term (list or constant) t , then the structure for x is retyped “instantiated”, the binding to t is recorded in the structure, and any suspended equality processes associated with x are added to the goal of the current computation. The reference associated with x , however, is now a reference to t , implementing rules 4 and 5 of the solved form algorithm. In the case where t is a list, i.e. not a constant, t is *not* checked for occurrences of x —i.e., Absys like the standard PROLOG interpreter, has no “occur check.”

Claim 4.1. This treatment of equality, apart from the lack of an occur check, captures the solved form algorithm and so, from what has been outlined earlier, completely justifies the claim that Absys was an (SLD) resolution based logic programming language, and to the author’s knowledge the first such logic programming language designed and implemented.

There is one caveat. It is possible for a computation which is a refutation to terminate¹³ with variables uninstantiated in a way that does not directly capture the full solved form. Because of the treatment of equality between uninstantiated variables, the total set of bindings is “incomplete”. More precisely, the binding environment is constrained by possibly “suspended” equations between variables. For example, a refutation might terminate with a binding environment such as

$$Z/[X, 2, [3]], \quad W/[X, 5, Y]$$

and with the equation $X = Y$ associated with uninstantiated variables X and Y . Such a constraint on the computed answer substitution can of course be queried in a natural way by querying either X or Y , and would in any case be activated if any additional goal assertion instantiating X or Y were added to the Absys text.¹⁴

A simple illustration may be helpful.

4.1. A Simple Example of Suspended Equations

We define procedures for reversing a list as follows:

```

rev1 = lambda R, S, T
      { R = [], S = T } or
      { R = [X|L], rev1(L, [X|S], T) }
rev  = lambda R, S
      { rev1(R, [], S) }

```

¹³“Hibernate” is a better word, since the computational process associated with the derivation can be reactivated—see below.

¹⁴Absys was an interactive, incremental system, and was designed with this in mind.

Ignoring failed derivations, the goal atom $rev([1, 2, U], [V|L])$ is elaborated as follows:

From the definition of rev we get

$$rev1([1, 2, U], [], [V|L]).$$

The definition of $rev1$ gives

$$[1, 2, U] = [X_1|L_1], \quad rev1(L_1, [X_1], [V|L]).$$

Processing the equation (cf. rules 1 and 4 of the solved form algorithm) gives the binding environment

$$X_1/1, \quad L_1/[2, U].$$

Elaboration of the goal atom $rev1([2, U], [1], [V|L])$ gives

$$[2, U] = [X_2|L_2], \quad rev1(L_2[X_2, 1], [V|L]).$$

Processing the equation augments the binding environment to

$$X_1/1, \quad L_1/[2, U], \quad X_2/2, \quad L_2/[U].$$

Elaboration of the goal atom $rev1([U], [2, 1], [V|L])$ gives

$$[U] = [X_3|L_3], \quad rev1(L_3, [X_3, 2, 1], [V|L]).$$

Processing the equation augments the binding environment by $L_3/[]$ and associates the equation $U = X_3$ with U and X_3 .

Elaborating the goal atom for the last time gives

$$[] = [], \quad [X_3, 2, 1] = [V|L].$$

Processing of the first equation terminates without further action (cf. rule 1 of the solved form algorithm), whilst processing of the second equation replaces it by (cf. rule 1)

$$X_3 = V, \quad [2, 1] = L.$$

Processing of these equations finally leads to the derivation terminating ("hibernating") with the binding environment

$$X_1/1, \quad L_1/[2, U], \quad X_2/2, \quad L_2/[U], \quad L_3/[], \quad L/[2, 1]$$

and the with the suspended equations

$$U = X_3 \quad \text{and} \quad X_3 = V.$$

The binding environment exhibits a "partially" solved form constrained by the suspended equations. If we query the computed answer substitution for U , we will be told that $U = X_3$, whereupon querying X_3 tells us that $X_3 = V$, i.e. that the computed answer substitution for U is constrained to be U/V .

The suspended equations represent a constraint on the *parameters* of the partially solved form. Since the constraint is a simple set of equalities between parameters, the constraint is always satisfiable and the partially solved form can be completed.

As mentioned earlier, Absys was an incremental system in that the text could be augmented by new assertions. This, together with the progressive development of

the binding environment (substitution) associated with a solved form, made it sensible and simpler to leave equations between variables unprocessed until one of the variables becomes instantiated by a term. Thus, in the example above, if we now add to the Absys text and augment the goal by typing the conjunct $rev([5],[V])$, say, then V will become bound to 5. The equation $X_3 = V$ associated with V is now reprocessed, binding X_3 to 5. This in turn causes the equation $U = X_3$ associated with X_3 to be reprocessed, binding U to 5.

The total binding environment now is

$$X_1/1, \quad L_1/[2,5], X_2/2, L_2/[5], L_3/[], L/[2,1],$$

$$X_4/5, \quad L_4/[], V/5, X_3/5, U/5,$$

where X_4 and L_4 are the variables introduced in processing $rev([5],[V])$. This is the substitution equivalent to the full solved form, and contains the answer substitution $L/[2,1], U/5, V/5$ for the now augmented goal.

5. NEGATION IN PROLOG AND IN ABSYS

5.1. Negation in PROLOG

The expressiveness of PROLOG is increased by allowing program clauses to contain negative literals in their bodies. The mechanism for handling the negative subgoals generated in a refutation is to augment SLD resolution with the negation-as-failure rule [1] (SLDNF resolution). The following is largely taken from Lloyd [21].

A *general program clause* is a clause of the form

$$p(t_1, \dots, t_n) \leftarrow L_1, \dots, L_m$$

where the L 's are literals.

A *general program* is a finite set of general program clauses.

A *general goal* is a clause of the form

$$\leftarrow L_1, \dots, L_m$$

where the L 's are literals.

Let P' be the homogeneous form of a general program P . Let a given predicate p in P' be defined by the k clauses

$$\begin{aligned} p(X_1, \dots, X_n) &\leftarrow E_1, \\ &\vdots \\ p(X_1, \dots, X_n) &\leftarrow E_k, \end{aligned}$$

in which each E_i has the reading

$$\exists Y_1, \dots, Y_r (X_1 = t_{i1}, \dots, X_n = t_{in}, L_1, \dots, L_{m_i}),$$

where Y_1 to Y_r are the variables of t_{i1} to L_{m_i} . The *completed definition* of p is the formula

$$\forall X_1, \dots, X_n \quad p(X_1, \dots, X_n) \leftrightarrow E_1 \vee \dots \vee E_k.$$

The equality theory for the predicate “=” introduced in the completion¹⁵ essentially constrains the predicate to be interpreted as the identity relation on the Herbrand universe of P .

Finally, let P be a general program.¹⁶ The *completion* of P is the collection of completed definitions for each predicate in P together with the equality theory. In PROLOG, the notion that we are working with a completion is essential for expressing the soundness [1] and completeness [14] results for the negation-as-failure rule.

In practice, in standard PROLOG the programmer gives the system the general program. The “understanding” would have to be that the “system” completes the general program. In addition, as Clark [1] points out, the system has to include the equality theory mentioned above.

5.2. Negation in Absys

Consider the PROLOG definition of *append*:

$$\begin{aligned} &app([], L, L) \\ &app([X|L], M, [X|N]) \leftarrow app(L, M, N) \end{aligned}$$

used in the example of Section 2.2.2. Its completion can be written

$$\begin{aligned} &\forall R, S, T (app(R, S, T) \leftrightarrow \\ &\quad \exists L (R = [], S = L, T = L) \vee \\ &\quad \exists X, L, M, N (R = [X|L], S = M, T = [X|N], app(L, S, N))). \end{aligned}$$

This can be simplified to

$$\begin{aligned} &\forall R, S, T (app(R, S, T) \leftrightarrow \\ &\quad (R = [], S = T) \vee \\ &\quad \exists X, L, N (R = [X|L], T = [X|N], app(L, S, N))) \end{aligned}$$

which is the reading of the Absys definition

$$\begin{aligned} &app = \text{lambda } R, S, T \\ &\quad \{ R = [], S = T \} \text{ or} \\ &\quad \{ R = [X|L], T = [X|N], app(L, S, N) \} \end{aligned}$$

given in Section 3. As mentioned there, Absys definitions of predicates are completed definitions. Consequently Absys programs are completions of PROLOG programs.

Absys allowed for general programs and goals using the operator *not*. The operator *not* distributes with respect to *or* and the implicit *and* in the expected way. The operator acts like a degenerate *or* in that during goal evaluation it initiates an independent computational branch, but one in which the satisfiability criteria for termination are reversed.

¹⁵Given in full in [1, 21].

¹⁶Assumed not to contain any undefined predicates.

We note that the solved form algorithm used to implement Absys equality captures the requirements of the equality theory for negation by failure¹⁷ listed by Clark. It follows that Absys implemented negation by failure in the required context of completions of general programs.¹⁸

A simple illustrative example from Clark [1] is the completion of his student “microdatabase”

```

student(brown) ←
student(smith) ←

takes(brown, c101) ←
takes(smith, c101) ←
takes(smith, c301) ←

math-course(c101) ←
math-course(c301) ←

non-math-major(X) ← math-course(Y), not(takes(X, Y))

```

The completion of this program is rendered in Absys as¹⁹

```

student = lambda X
    { X = 'BROWN' } or { X = 'SMITH' }

math-course = lambda C
    { C = 'C101' } or { C = 'C301' }

takes = lambda X, C
    { X = 'BROWN', C = 'C101' } or
    { X = 'SMITH', C = 'C101' } or
    { X = 'SMITH', C = 'C301' }

non-maths-major = lambda X
    { maths-course(C), not(takes(X, C)) }

```

Finally, a nice example taken from a demonstration of Absys-4 circa 1968, using the Absys primitive aggregation operator *set* (implemented in an analogous way to the PROLOG operator introduced some years later), is

```

setdiff = lambda S1, S2, S
    { set(X), { mem(X, S1), not(mem(X, S2)) }, S }

```

More complex examples can be found in [7].

¹⁷Again with the exception of an “occur check”.

¹⁸As implementations both Absys and PROLOG have the shortcoming that they do not use a *safe* computation rule, and so soundness and completeness are not guaranteed in either.

¹⁹With apologies for the flip in syntax.

6. EPILOGUE

This paper was written to fulfill several goals.

- (1) It is presented as a contribution to the history of what might well turn out to be a computational revolution.
- (2) It is intended as a “gift” to all engaged on the Absys project,²⁰ but particularly to Michael Foster, who most clearly recognized that, expertise in logic apart, one was unlikely to get the wrong result for the right reason (with apologies to T. S. Eliot).
- (3) It is unashamedly intended to be a sixtieth-birthday present to myself and to allay forever ghosts and goblins of fact and ethics, past, present, and future, by removing misunderstandings of the nature and achievements of Absys, which can in retrospect be seen for what it was—the first implemented logic programming language based on Horn logic with equational unification and SLD resolution as its refutation procedure. Needless to say, I take full responsibility if any of my rewriting of history is unacceptable.

I would like to thank Alan Robinson and Jean-Louis Lassez for their encouragement and for the many helpful suggestions about what I really meant to say. Of course, I do not hold them responsible if I did not say it.

The work was supported by the National Science and Engineering Research Council under operating grant A9123.

REFERENCES

1. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.) *Logic and Databases*, Plenum, New York, 1978.
2. Colmerauer, A., Les Systems-Q ou un Formalisme pour Analyser et Synthetiser des Phrases sur Ordinateur, Publication Interne No. 43, Department d'Informatique, Univ. de Montreal, 1973.
3. Colmerauer, A., *Prolog II Reference Manual and Theoretical Model*, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille, 1982.
4. Colmerauer, A., Prolog and Infinite Trees, in: K. L. Clark and S. A. Tarnlund (eds.), *Logic Programming*, Academic, New York, 1982.
5. Cox, P. T., Deduction Plans: A Graphical Proof Procedure for First Order Predicate Calculus, Ph.D. Thesis, Univ. of Waterloo, 1977.
6. Elcock, E. W., Descriptions, in: D. Michie (ed.), *Machine Intelligence 3*, Edinburgh U.P., 1968.
7. Elcock, E. W., Problem Solving Compilers, in: N. Findler (ed.), *Artificial Intelligence and Heuristic Programming*, Edinburgh U.P., 1971.
8. Elcock, E. W., McGregor, J. J., and Murray, A. M., Data Directed Control and Operating Systems, *British Comput. J.* 15(2):125–129 (1972).
9. Elcock, E. W., and Gray, P. M. D., Absys, Equation Solving and Logic Programming, TR 213, Dept. of Computer Science, Univ. of Western Ontario, 1988.

²⁰E. W. Elcock, J. M. Foster, P. M. D. Gray, J. J. McGregor, and A. M. Murray, all at that time members of the Science Research Council Group for Computer Research, the University of Aberdeen, Scotland.

10. Foster, J. M., Assertions: Programs Written without Specifying Unnecessary Order, in: D. Michie, (ed.), *Machine Intelligence 3*, Edinburgh U.P., 1968.
11. Foster, J. M., and Elcock, E. W., Absys 1: An Incremental Compiler for Assertions—an Introduction, in: D. Michie (ed.), *Machine Intelligence 4*, Edinburgh U.P., 1969.
12. Hayes, P. J., Computation and Deduction, in: *Proceedings of the 2nd IJCAI-1*, Czechoslovak Academy of Sciences, 1973, pp. 105–118.
13. Hill, R., LUSH-resolution and its completeness, DCL Memo. 78, Dept. of Computational Logic, Univ. of Edinburgh, 1974.
14. Jaffar, J., Lassez, J.-L., and Lloyd, J. W., Completeness of the Negation by Failure Rule, in: *IJCAI-83*, Karlsruhe, 1983, pp. 500–506.
15. Jaffar, J. and Lassez, J.-L., Constraint Logic Programming, in: *Proceeding of the 14th ACM POPL Conference*, Munich, 1987.
16. Kowalski, R. A. and Kuehner, D., Linear Resolution with Selection Function, *J. Artificial Intelligence* 2:227–260 (1971).
17. Kowalski, R. A., Predicate Logic as a Programming Language, in: *Proceedings of IFIP 74*, North Holland, Amsterdam, 1974, pp. 569–574.
18. Kowalski, R. A., The Early Years of Logic Programming, *Comm. ACM* 31(1):38–44 (1988).
19. Lassez, J.-L. and Maher, M. J., Closures and Fairness in the Semantics of Programming Logic, *Theoret. Comput. Sci.* 29:167–184 (1984).
20. Lassez, J.-L., Maher, M. J., and Marriott, K. G., Unification Revisited, RC 12394 (No. 55630), IBM–T. J. Watson Research Center, 1986.
21. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.
22. Martelli, A. and Montanari, U., An Efficient Unification Algorithm, *ACM TOPLAS* 4(2):258–282 (1982).
23. McCabe, F. G., *Micro-Prolog—Programmer's Reference Manual*, Logic Programming Associates, 1980.
24. Naish, L., An Introduction to MU-PROLOG, TR 82/2, Dept. of Computer Science, Univ. of Melbourne, 1982.
25. Robinson, J. A., A Machine Oriented Logic Based on the Resolution Principle, *J. Assoc. Comput. Mach.* 12(1):23–41 (1965).
26. Robinson, J. A., *Logic: Form and Function*, Elsevier North Holland, New York, 1979.
27. Roussel, P., Publication Interne, Group d'Intelligence Artificielle, Univ. d'Aix-Marseille Lumini, 1975.
28. Steele, G. L. (Jr.) and Sussman, G. J., Constraints, *Proceedings of APL '79*, *APL Quote Quad* 9(4):662–642 (1979).
29. van Emden, M. H. and Lloyd, J. W., A Logical Reconstruction of Prolog II, *J. Logic Programming* 1:143–149 (1984).
30. Wolfram, D. A., Maher, M. J., and Lassez J.-L., A Unified Treatment of Resolution Strategies for Logic Programs, TR 83/12, Dept. of Computer Science, Univ. of Melbourne, 1983.