3
Beyond LOGLISP: combining functional and relational programming in a reduction setting

J. A. Robinson
School of Computer and Information Science, Syracuse University, USA

The initial plan for LOGLISP [1] was simply that it would offer, within LISP, a Horn-clause relational programming facility akin to PROLOG. This it does, but with some differences from PROLOG, notably the use of a breadth-first, rather than depth-first, elaboration of the underlying tree of alternative linear proofs, and the consequent avoidance of explicit backtracking as a control mechanism. It was because of these differences that the facility was called LOGIC rather than PROLOG, which would have been misleading. The name LOGLISP then refers to the combined system: LOGIC + LISP.

It soon became apparent, however, that the main interest of LOGLISP lay rather in its (relatively crude, but genuine) attempt to merge the functional programming style of LISP with the relational programming style of LOGIC and PROLOG. This was done by introducing the notion of 'LISP-transforms' into LOGIC.

The LISP-transform of a simple expression (atomic sentence or term) E of LOGIC is an expression which in many cases is the same as that obtained by applying LISP's EVAL function to E.

Thus, the LISP-transform of (PLUS 3 (TIMES 2 6)) is 15. However, the LISP-transform of (PLUS x (TIMES 2 6)) is (PLUS x 12), assuming that x is not defined ('has no value').

It is as though the LISP EVAL function had been modified to be more tolerant of undefined identifiers and to return the symbol or function call itself as its 'value' if it has no value in the usual sense.

Before seeking to unify a goal statement with the heads of appropriate Horn clauses, LOGIC first replaces it by its LISP-transform. This corresponds to PROLOG's concept of first executing the goal statement (if it is executable) but with the difference that it is then sent forward to the unifier for attempted resolutions, rather than being discarded as in PROLOG. Indeed, since real LISP has constructs whose evaluation causes side-effects, these can occur when LOGLISP computes a LISP-transform.

This step of replacing each selected goal by its LISP-transform and then

attempting to resolve away the transformed goal has far-reaching consequences.

An obvious and immediate consequence is to provide 'built-in' functions and predicates for LOGIC: any identifier with a LISP definition (whether a system or a user definition) will 'feel' that definition during the LISP-transformation process. In particular a LISP-defined goal sentence such as (LESSP 3 4) would be transformed, in this case to T(RUE), the goal sentence which in LOGIC is always unconditionally provable.

A less obvious consequence is that a goal sentence can contain calls on the LOGIC system itself (since they are LOGIC expressions as well as LISP expressions) such as calls on SETOF or ASSERT. These calls can be nested, so that one can compute, for example, the set of all Jim's cousins who have no sons by LISP-transforming the expression

(SETOF ALL x (COUSIN x JIM) and (NULL (SETOF ALL y (SON y x)))

so as to get (say)

(MARY BILL GEORGE).

This rather serendipitous feature of LOGLISP led to the realization that the SETOF construct is quite central in relational programming. Sets (represented in LOGLISP as lists) and relations (sets of tuples) are data objects constructed by deductively evaluating set descriptions and relation descriptions, using sets and relations defined by Horn clauses.

In constructing such sets and relations it is natural in LOGLISP to invoke functions defined in the usual LISP manner and to engineer the overall transaction as a mixture of LISP and LOGIC steps, but there is nothing in the constructions themselves which demands this distinction.

This suggests seeking a more complete merger of the relational and functional paradigms than LOGLISP provides. LOGLISP maintains 'separate but equal' environment management facilities, one for LOGIC variables, which deals with bindings made by unification, and another for LISP variables, which deals with bindings made by assignments and by function calls. The LISP-transformation process is distinguished from the LOGIC process of proving goals by resolution—indeed, these two processes are alternated in a two-phase cycle. The net overall effect is to implement, in this awkward way, a 'LOGLISP-reduction' process, but as a kind of antiphonal duet. It seems clear that there really ought to be only one process, rather than two. Definitions of functions and definitions of relations are not essentially different, and should be invoked in the same way. Variables are variables—there should be just one kind. There is nothing special about 'logical' variables, nor about unification. As will be seen, unification can be viewed as a kind of reduction process, and its steps can be treated in the same way as steps of reduction in general are treated.

1. LISP VIEWED AS A REDUCTION SYSTEM

The simplest view one can usefully take of LISP is that it offers the user two facilities: (a) a definition facility; and (b) a deduction facility.

The first of these allows one to define functions by, in effect, asserting new axioms. Each definition is essentially an equation

$$F = \text{lambda}\, X\, B$$

which associates with a symbol F a function described in the notation of the lambda calculus. Here, X is a list of distinct formal variables, while B is the 'body' of the description of the function. Thus one might assert the definition

$$! = \text{lambda}\, (n)\, (\text{if } n = 0 \text{ then } 1 \text{ else } n * (!\, (n - 1))$$

of the factorial function by introducing the symbol ! on the left-hand side and describing the function on the right-hand side by means of an expression in which ! occurs. Such recursive definitions are the very stuff of functional programs.

The second facility allows one, in effect, to pose certain kinds of deduction problem and have them solved. One might express these as:

> find the expression V in normal form for
> which the equation
>
> $E = V$
>
> is deducible from the (current set of) axioms.

In LISP one usually thinks of V as the result of applying EVAL to E, and calls it the 'value' of E. Thus if E is $(!\, 6)$ one would expect that V would be 720.

Not all expressions are in normal form. The general idea is that if an expression contains one or more subexpressions which can be rewritten in accordance with some definition then it is not yet in normal form and can be further 'reduced' by rewriting one of those subexpressions, or 'redexes', as Curry called them. This 'reduction' process can be kept up for as long as the expression contains such redexes, and in general the replacement of some redexes may well create new redexes, and so on. To be in normal form is, precisely, to contain no redexes.

This view of computation is the reduction point of view which comes with the classical lambda calculus. It automatically entails a 'no error stops' treatment of computation—an expression E always can be transformed, if it is not already in normal form. The point is that, for example, $(x + 4)$ is not an error if x is undefined. Instead, it is irreducible, and hence is in normal form. Thus, being in normal form is a relative notion. It depends on the set of definitions which is currently in

force. For example, if the definition

$$x = 5$$

is added then the expression $(x + 4)$ is no longer irreducible and reduces to the expression 9.

2. LOGIC VIEWED AS A REDUCTION SYSTEM

The simplest view one can take of LOGIC (or indeed of 'pure' PROLOG) is very similar to the above view of LISP. Again, two facilities are offered: (a) a definition facility; and (b) a deduction facility.

The first of these allows one to define relations by asserting axioms, called 'positive Horn clauses'. To define R one asserts, in effect, a single equation with a right-hand side which describes a relation by means of a disjunction of simple sentences each corresponding to such a clause:

$$R = \text{lambda } X \text{ (OR (for some } Y_1 : (X = T_1 \text{ and } B_1))$$
$$\vdots$$
$$\text{(for some } Y_n : (X = T_n \text{ and } B_n)))$$

where X is a list of distinct formal variables, T_i is the equally long list of terms which is the argument of R in the ith clause, Y_i is the (possibly empty) list of 'local' variables of the ith clause, and B_i is a conjunction of atomic sentences comprising the body of the ith clause.

In LOGIC and PROLOG one asserts the clauses separately for each i in the form

$$\text{for all } Y_i : R\ T_i \text{ if } B_i.$$

Provided (as Clark [2] argues and as is surely the case) one intends R to hold of a tuple ONLY IF one of these clauses applies, then it is straightforward to see that the conjunction of the separate clauses is equivalent to the single equation.

Another way of thinking about the definition of a relation R by a single equation is that it describes R as a union of relations, namely as

$$\text{(UNION lambda } X \text{ (for some } Y_1 : (X = T_1 \text{ and } B_1))$$
$$\vdots$$
$$\text{lambda } X \text{ (for some } Y_n : (X = T_n \text{ and } B_n))).$$

Although it is not the usual custom to do so, one can read 'lambda X' here as 'the set of all X such that', bearing in mind that, after all, a relation is usually construed as a set of tuples.

For example, one can define the APPEND relation in this manner as the

union of two relations:

APPEND = (UNION lambda (a, b, c) (for some x: $a = []$ and
$b = x$ and
$c = x$))
lambda (a, b, c) (for some x, y, z, w:
$a = [x \,.\, y]$ and
$b = z$ and
$c = [x \,.\, w]$ and
(APPEND $y\ z\ w$)))

the second of which refers recursively to the APPEND relation itself.

The deduction facility of PROLOG or LOGIC is best viewed as one for solving problems of the form:

find the expression V in normal form such that
the equation

lambda X (for some Y: A) = V

is provable from the (current set of) axioms.

In other words, find the (normal form description of) the set of all X such that A holds for some Y. It is assumed here that the normal form of a set description is one which lists the set's elements, as for example: $\{2, 4, 6\}$. One might note that this customary notation is insignificantly different from

lambda (x) ($x = 2$ or $x = 4$ or $x = 6$)

or from its description as the union of the singletons

(UNION (lambda (x) $x = 2$) (lambda (x) $x = 4$) (lambda (x) $x = 6$)).

LOGIC tries to provide such a 'set description' deduction facility with its SETOF function. Different versions of PROLOG vary on this point, but the practice is becoming more and more common to provide such a deductive construct in addition to the basic one, which in effect solves the problem:

find a term T such that the sentence

(lambda X (for some Y: A) T)

is provable from the (current set of) axioms.

The solutions T to this problem are in general not uniquely determined by the condition A, and so one speaks of the non-determinacy of the process of finding such a T. In fact, however, successive posings of the same problem are not independent, and run through the elements of the set:

lambda X (for some Y: A)

in some order. Thus, PROLOGS do find the set, but they do it one element at a time; and they do not always offer the service of representing the set by a single expression (or what is the same, as a data object).

So it can be seen that both LISP and LOGIC are essentially in the same business: of accepting definitions in the form of equations and of solving deduction problems by reduction of a given expression to its normal form. Why, then, should they be kept separate from each other? I believe that they need not and should not be.

A system being developed at Syracuse University will now be discussed in which both functional and relational programming merge into a single definition–deduction paradigm. The system is called SUPER (for Syracuse University Parallel Expression Reducer).

SUPER is a reduction system with a repertory of rewrite rules including all those one would expect in a (pure) LISP-like lambda calculus. In particular it will have the usual rule of beta reduction, which calls for the replacement of a redex of the form

(lambda X B) A

by the expression resulting from substituting A for (free) X throughout B. The constants

EVERY, SOME

are added to the language so as to provide the logical quantifiers via the constructions

for all X: A = (EVERY (lambda X A))

for some X: A = (SOME (lambda X A))

which go back to Church's language [3] for the simple theory of types. SUPER will be based on that system (which has been studied by Henkin [4] who gave a completeness result for it, and which recently was used by Andrews [5] as the formalism for a higher-order theorem proving system). Thus SUPER has only two syntactic constructs: application (of one expression as function, to another expression as argument) and abstraction. Its expressive power comes from a suitable collection of constants: TRUE, FALSE, NOT, AND, OR, IF, IFF, for the Boolean combinations of elementary logic; CAR, CDR, CONS, ATOM, EQ, COND, NIL, etc., for the LISP-like symbolic apparatus; PLUS, TIMES, etc., together with suitable numerals, for arithmetic. All of these constants will require appropriate reduction rules to give them operational meaning, and all of them are familiar and straightforward with the exception perhaps of those dealing with the SUPER version of unification. These will be discussed next.

Let us follow a few of the transformations necessary to compute the

normal form of the following set description:

$$\text{lambda } (p, q)\ (\text{APPEND } p\ q\ [1\,2\,3]) \tag{1}$$

which we can see, intuitively, is

$$\begin{array}{l}\{([], [1\,2\,3])\\ \ ([1], [2\,3])\\ \ ([1\,2], [3])\\ \ ([1\,2\,3], [])\}.\end{array} \tag{1'}$$

Here and henceforth the convention for list notation whereby, for example, the list $[1\,.\,[2\,.\,[3\,.\,[]]]]$ is written more readably as $[1\,2\,3]$ is used.

The expansion rule takes the definition of APPEND as the union of two simple relations and rewrites (1) as the union of two relations

$$\begin{array}{l}(\text{UNION lambda } (p, q)\ (\text{for some } x{:}\ p = [] \text{ and}\\ \qquad\qquad\qquad\qquad\qquad\qquad q = x \text{ and}\\ \qquad\qquad\qquad\qquad\qquad [1\,2\,3] = x)\\ \qquad \text{lambda } (p, q)\ (\text{for some } x,y,z,w{:}\ p = [x\,.\,y] \text{ and}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad q = z \quad \text{and}\\ \qquad\qquad\qquad\qquad\qquad\qquad [1\,2\,3] = [x\,.\,w] \text{ and}\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (\text{APPEND } y\ z\ w)))\end{array} \tag{2}$$

(In applying the expansion rule it is necessary to pay attention to the question of clashes of bound variables and to take precautions of the usual kind, namely, to change the local variables of the set descriptions, if necessary, before replacing the formal variables by actual terms of the redex.)

In the second component of the union expression there is an expansion redex, but there are now two redexes of another kind, one in the first and one in the second component of the union. These call for applications of the contraction rule. According to this rule a local variable can be dropped from the quantifier prefix of the body of a simple relation expression provided that one of the conjuncts in the body is an equation between that variable and some term which does not contain it. In the first component above the variable is x and the term is $[1\,2\,3]$. In the second component the variable is z and the term is q. In addition to dropping the variable from the prefix all of its occurrences in the body must be replaced by the term. So in the above x is replaced throughout the body of the first component by $[1\,2\,3]$, and z throughout the body of the second component by q. Finally, the trivial equation thus created is dropped. In the first component this is the equation $[1\,2\,3] = [1\,2\,3]$. In

the second component it is $q = q$. The resulting expression is

$$\begin{aligned}&(\text{UNION lambda } (p, q)\ (p = [] \text{ and } q = [1\,2\,3])\\ &\qquad \text{lambda } (p, q)\ (\text{for some } x, y, w: p = [x\,.\,y] \text{ and}\\ &\qquad\qquad [1\ 2\ 3] = [x\,.\,w] \text{ and}\\ &\qquad\qquad (\text{APPEND } y\ q\ w))))\end{aligned} \tag{3}$$

It should be noted that the contraction rule applies only to redexes of the particular kind described. The form in general is

lambda A (for some B: C_1 and $\cdots$ and C_m and $(V = E)$ and D_1 and $\cdots$ and D_n)

where the list B of local variables contains the variable V and E is a term not containing V. (The equation can also be $E = V$.) The redex is replaced by

lambda A (for some B': $(C_1$ and $\cdots$ and C_m and D_1 and $\cdots$ and $D_n)\{E/V\}$)

where B' is the list B with V omitted. That is, we replace V by E throughout the conjunction, and drop the trivial equation $E = E$ thus created.

Notice that in this example contraction is applied simultaneously to two different redexes. This is a small example of the way in which reduction can be done in parallel.

The next step illustrates the decomposition rule. This states that a redex of the form

lambda A (for some B: $(C$ and $[P\,.\,Q] = [R\,.\,S]$ and $D)$)

where C and D are both simple conjunctions (possible empty), may be replaced by

lambda A (for some B: $(C$ and $P = R$ and $Q = S$ and $D)$).

All that this rule is saying is that an equation between two dotted pairs is equivalent to two equations between their respective heads and tails. Applying decomposition to (3) yields

$$\begin{aligned}&(\text{UNION lambda } (p, q)\ (p = [] \text{ and } q = [1\,2\,3])\\ &\qquad \text{lambda } (p, q)\ (\text{for some } x,y,w: p = [x\,.\,y]\\ &\qquad\qquad 1 = x\\ &\qquad\qquad [2\,3] = w \text{ and}\\ &\qquad\qquad (\text{APPEND } y\ q\ w)))\end{aligned} \tag{4}$$

thus creating two new equations which permit contraction to be applied

twice more, after which we have

$$\begin{aligned}&(\text{UNION lambda } (p, q)\ (p = [] \text{ and } q = [1\,2\,3])\\ &\qquad \text{lambda } (p, q)\ (\text{for some } y{:}\ p = [1\,.\,y] \qquad (5)\\ &\qquad\qquad (\text{APPEND } y\ q\ [2\,3])))\end{aligned}$$

and we must take stock of what has been happening to the original set description (1) as it is step-by-step being transformed into the description (1′).

In (5) already a singleton set containing the first couple has emerged. The other three elements have yet to emerge from the description of the rest of the set. However, this description has been partially developed, and now intuitively reads 'the set of all couples of lists, the first of which starts with 1 and has a tail which, when appended to the second, yields the list [2 3]'. We can intuitively see that this is the set

$$\{([1], [2\,3]), ([1\,2], [3]), ([1\,2\,3], [])\}.$$

Further applications of the three rules expansion (using the definition of APPEND), contraction and decomposition will carry (5) step-by-step nearer to the union of singleton sets which is the required normal form representing (1′). Only one further rule is required to complete the overall transformation. This is the failure rule which states that a redex of the form

$$(\text{UNION } S_1 \cdots S_m\ (\text{lambda } A\ (\text{for some } B{:}\ C))\ T_1 \cdots T_n)$$

where C is a conjunction of simple sentences one of which is obviously false, may be replaced by the expression

$$(\text{UNION } S_1 \cdots S_m\ T_1 \cdots T_n).$$

The intuitive justification of the rule is that the deleted component describes the empty set.

Some examples of 'obviously false' simple sentences are:

$$\text{FALSE}, [] = [x\,.\,y],\ 2 = 3.$$

The failure rule is so named because it corresponds to the occasions in the unification process when an attempt to unify two expressions fails.

Let us skip forward to the point where the failure rule comes into play in our example. The following description is reached after applying an

expansion:

(UNION lambda (p, q) $p = []$ and $q = [1\,2\,3]$
lambda (p, q) $p = [1]$ and $q = [2\,3]$
lambda (p, q) $p = [1\,2]$ and $q = [3]$
lambda (p, q) $p = [1\,2\,3]$ and $q = []$
lambda (p, q) (for some x, y, z, w:
$p = [1\,2\,3\,.\,y]$ and
$y = [x\,.\,z]$ and
$[] = [x\,.\,w]$ and
(APPEND $y\ q\ w$)))

whose final component contains the impossible equation $[] = [x\,.\,w]$. It also contains a further recursive call on the APPEND definition, and hence is an expansion redex. In addition, by virtue of containing the equation for the local variable y it is a contraction redex. However, the rewriting of the entire UNION expression by the failure rule simply drops these redexes and thus would be the most advantageous choice.

The final expression is then (ensugaring the singletons):

(UNION {([], [1 2 3])}
{([1], [2 3])}
{([1 2], [3])}
{([1 2 3], [])})

which can be further ensugared to the form of (1′) if desired.

The overall computation sketched in this example corresponds, in a reduction setting, to the complete exploration of the tree of alternative Horn-clause resolution deductions which PROLOG or LOGIC would perform in response to a request to find the set of all (p, q) such that (APPEND $p\ q$ [1 2 3]). The expansions correspond to the invocations of the two clauses of the APPEND definition; the contractions to the successive bindings made by the unification process in attempting to unify a selected goal with a clause head; the decompositions to the recursive calls to the unification algorithm when two dotted pairs are to be unified; and the failures to the moments when the unification algorithm encounters an impossible combination.

3. SCOPE OF SUPER

Although the underlying language of SUPER is the simple theory of types, sometimes also called the predicate calculus of order omega, at present the experiment has not gone far enough to know whether the repertory of rules can be usefully extended to cover higher-order unification. Huet [6] and Pietrzykowski and Jensen [7] have given unification algorithms for

the typed lambda calculus, but the computational problems are much more complex than in the first order case.

The objectives so far have been limited to reorganizing the present stock of ideas about first-order relational programming so that at least the same capability one has in LOGIC can be reproduced in the reduction setting. The next goal is to investigate the feasibility of implementing SUPER, as it presently exists, in a parallel reduction architecture. Klaus Berkling is currently designing a second version of his well-known GMD Reduction Machine [8] which will embody the expansion, contraction, decomposition and failure rules as well as a full set of rules suitable for a LISP-like functional language. This machine will have a multiprocessor architecture and its design is at present under way.

4. FUTURE WORK

It would, of course, be very interesting to extend these rules beyond the first-order unification level. Another line of investigation is to see how far, if at all, one can push the combinator approach which in recent years has been so well exploited by Turner [9]. The main problem seems to be that transformation of a lambda abstraction to pure combinator form disarranges the syntactic structure of the original so much that the unification analysis cannot be carried out. In the present system this comes out in the contraction rule: in order to apply the rule one has to identify the term which will be substituted for the variable being eliminated. This is easy enough in the original expression, but after transformation into pure combinator form there are no longer any variables and in particular no equations between variables and terms. There seems therefore to be no way to identify the term. So the question is: what corresponds to the unification process after all variables have been transformed away?

It would be useful to know more about the role played by the typing of expressions in the SUPER language. Huet's higher-order unification algorithm makes crucial use of the types of expressions at certain stages, but none of the rules considered in this discussion do, as witnessed by the fact that types have scarcely been mentioned.

Finally, it should be said that aspirations do not presently extend to building a complete proof procedure for SUPER, although this is not, in view of Henkin's result cited earlier [4], out of the question. Rather something like a higher-order Horn clause resolution theorem prover is sought in which it will be possible to do logic computations à la Kowalski [10] but without the restriction to first order. Related work is in progress at Imperial College by Darlington and his group [11] and at Cambridge University by Paulson [12]. The point is to retain, if possible, the directed

purposefulness of a computation process, and not to slide back into the world of mere searching.

REFERENCES

1. Robinson, J. A. and Sibert, E. E. (1982) LOGLISP: an alternative to PROLOG. In *Machine intelligence 10* (eds J. E. Hayes, D. Michie, and Y.-H. Pao) pp. 399–419. Ellis Horwood, Chichester and New York, Halsted Press.
2. Clark, K. L. Negation as failure. In *Logic and databases* (eds H. Gallaire and J. Minker) pp. 293–294. Plenum Press, New York.
3. Church, A. (1940) A formulation of the simple theory of types. *J. Symbolic Logic* **5**, 56–68.
4. Henkin, L. (1980) Completeness in the theory of types. *J. Symbolic Logic* **15**, 81–91.
5. Andrews, P. B., Miller, D. A., Cohen, E. L., and Pfenning, F. (1984) Automating higher order logic. In *Automated theorem proving: after 25 years* (eds W. W. Bledsoe and D. Loveland) *Contemporary mathematics*, Vol. 29, pp. 169–92. American Mathematical Society.
6. Huet, G. (1975) A unification algorithm for typed lambda calculus. *Theoretical Computer Science* **1**, 27–57.
7. Pietrzykowski, T. and Jensen, D. A complete mechanization of omega-order type theory . *ACM National Conference* 1972, Vol. 1, pp. 82–92.
8. Berkling, K. J. (1976) Reduction languages for reduction machines. *Gesellschaft für Mathematik und Datenverarbeitung* **957**, Bonn.
9. Turner, D. A. (1979) A new implementation technique for applicative languages. *Software Practice and Experience* **9**, 31–49.
10. Kowalski, R. A. (1979) *Logic for problem solving*. North Holland, Amsterdam.
11. Darlington, J., Field, A. J., and Pull, H. (1985) The unification of functional and logic languages. Report DOC 85/3. Imperial College, London.
12. Paulson, L. C. (1985) Natural deduction theorem proving via higher order resolution. Technical report No. 67. Computer Laboratory, Cambridge University.