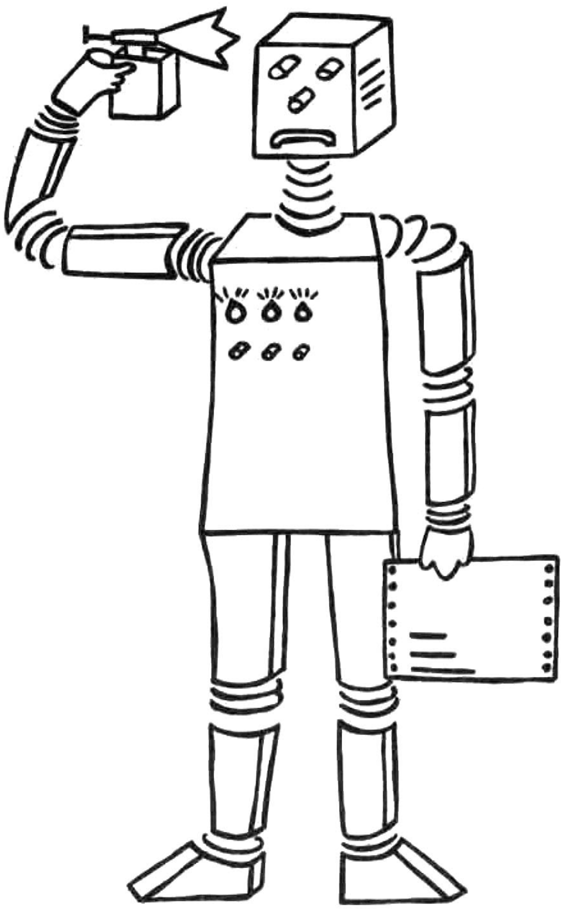
SEKI-REPORT

Artificial
Intelligence
Laboratories

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



RELFUN: A Relational/Functional Integration with Valued Clauses

Harold Boley

May 1986 SEKI-REPORT SR-86-04

RELFUN: A RELATIONAL/FUNCTIONAL INTEGRATION WITH VALUED CLAUSES ^

Harold Boley, Fachbereich Informatik, Universitaet Kaiserslautern
Postfach 3049, D-6750 Kaiserslautern, W. Germany

Abstract:

The RELFUN programming language is introduced as an attempt to integrate the capabilities of the relational and functional styles. Definitions of functions and relations are specified uniformly by valued Horn clauses, where rules return the value of their right-most premise. Functional nestings are flattened to relational conjunctions, using a purified version of PROLOG's is-primitive. RELFUN functions may have non-ground arguments, like relations, and/or return non-ground values; their input and output arguments can be inverted like those of relations. Higher-order functions are definable as (function-)valued clauses, with funarg problems being avoided by the standard renaming of clause variables. RELFUN's operational semantics is specified as an abstract machine, which also yields its first (e-mailable) FRANZ LISP implementation.

1. Introduction

The various proposals for combining logic (relational) and applicative (functional) languages differ in the degree to which they merge the two original components. For instance, many of the present LISP/PROLOG combinations, tabulated from HCPRVR to LISPLOG in [Boley & Kammermeier et al. 1985], consist of largely independent LISP and PROLOG parts. The separate identities of a functional and a relational component are also found in some less LISP/PROLOG-oriented proposals, though not often as easily as in RF-Maple [Voda & Yu 1984]. Other, more theoretically motivated approaches focus, however, on a unifying concept, such as equations in Eqlog [Goguen & Meseguer 1984], in an attempt to closely unite the functional and relational styles. Let us refer metaphorically to the two ends of this spectrum as the (function and relation) 'addition' vs. 'integration' paradigm.

Some advantages of the language addition paradigm accrue from the ease with which the two ingredients can be developed, tuned, maintained, and presented independently. Certain (casual, novice) users may even attempt to ignore one of the ingredients and run, for instance, their old LISP or PROLOG programs with minimal modification. More importantly, the functional and relational styles in isolation are better understood both in programming theory and practice than any of their various 'holistic' integrations.

Some advantages of the language integration paradigm are the greater simplicity, clarity, regularity, parsimony, and power of a single, unified language. In the long run, only a language more

^ This research was supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314, "Kuenstliche Intelligenz - Wissensbasierte Systeme".

powerful than, let us say, LISP and PROLOG together, whose size is less than the sum of their sizes, will attract users of these wide-spread languages. An underlying reason for the interest recently aroused by the integration paradigm might have been the following: The kinship between the functional and relational programming methods is intuitively so obvious that one would expect a common root, which, once unearthed, may also shed some new light on these 'twin disciplines' individually.

This paper discusses the RELFUN (RELational/FUNctional language) integration attempt, whose unifying concept is resolution/flattening computation with 'valued clauses'. RELFUN's operational semantics and implementation were based on a version of Kenneth Kahn's "Pure PROLOG in Pure LISP" interpreter [Kahn 1983/84], which was also the basis of our LISPLOG development [Boley & Kammermeier et al. 1985] within the LISP/PROLOG addition paradigm. Both RELFUN and LISPLOG draw on experience with FIT [Boley 1983], a somewhat different 'integration' language in which fitting plays the dominant role usually played by unification.

Although in the short term the pragmatically more straightforward addition paradigm may be superior to the theoretically more satisfactory integration paradigm, it is our belief that in the long term the latter will supersede the former. A similar conviction seems to underlie the research of Alan Robinson's group, which is directed toward a successor to the well-known LOGLISP [Robinson & Sibert 1982] language.

At present, however, while realizations (mostly in LISP) are available for many LISP/PROLOG additions, (transparent) implementations have not been published for many of the proposed functional/relational integrations. In this respect the RELFUN integration profits from the specification of its operational semantics as an abstract machine: notated in a simple functional subset of FRANZ LISP, this specification also provides us with an initial interpreter implementation acting as a 'rapid prototype'. The listing of 'pure RELFUN' in the appendix is easily input into one's local FRANZ LISP (compiler) or transcribed to other LISP dialects; our full RELFUN implementation augments this chiefly by impure primitives and by an improved interface and efficiency.

2. Valued resolution

Within the integration paradigm there exist several quite different approaches, which may be grouped by their principal unifying concept. Three typical integration concepts are the following: Equations can be used to define functions, but are themselves still relations (involving the binary relation symbol "="); they are applied, e.g., in William Kornfeld's PROLOG extension [Kornfeld 1983], in Eqlog [Goguen & Meseguer 1984], and in [Fribourg 1984]. Descriptions can be used for accessing relations as if they were (non-deterministic) functions; they are implemented, e.g., as QUTE's epsilon expressions [Sato & Sakurai 1983] and FIT's LOCAL expressions [Boley 1983]. Lazy evaluation can be used as the functional correlate to relational non-determinism (backtracking); it is utilized, e.g., in Funlog [Subrahmanyam & You 1984] and in RF-Maple [Voda & Yu 1984].

The integration approach taken in RELFUN synthesizes functions and relations right at the level of their definitions through recursion equations and Horn clauses. Thus, instead of distinguishing functional and relational definition methods, as by using Eqlog's "=" and ":-", respectively, only one definition method is used. All RELFUN definitions are generalized Horn clauses (facts and rules), called valued clauses, of the form (conclusion premise1 ... premiseN). The generalization consists of permitting arbitrary terms, not only goals, as the premises of valued rules and assigning a value to each resolution of a goal with a clause as follows ('valued resolution'):

Goal/fact resolution (N=0): 'true'.
Goal/rule resolution (N>0): the value of premiseN.

A right-most rule premise which is again a goal is employed recursively for valued resolution. Ultimately, if premiseN resolves with a fact, the valuation process ends with 'true'; if premiseN is a 'final' term, namely an arbitrary (LISP) atom, (logical) variable, or (instantiation) pattern, it ends with the term's [ultimate] instantiation result in the current binding environment (atoms and unbound variables instantiate to themselves). In RELFUN, variables are marked by the prefix "_"; patterns are submitted to what is called '[recursive] realization' in [Robinson & Sibert 1982] by prefixing them with an explicit instantiation operator, "`", which denotes the important mapping from the relational 'binding world' to the functional 'value world'.

If used in the ordinary way, for relation definitions, valued clauses behave as in relational languages, except that on success they return the value 'true' in addition to binding possible request variables; on fail they yield the value 'unknown'.

To reformulate a well-known relational example, using the database

```
((likes mary wine))                 ; N=0
((likes john _x) (likes _x wine))  ; N=1
```

the goal (likes john _whom) value-resolves with the only rule by binding the variable _whom to the variable _x and yielding the value of its single premise, (likes _x wine); this goal, again, value-resolves with the only fact by returning the value 'true' and binding the variable _x, hence _whom, to mary.

Assigning explicit truth-values to every relation call is akin to the classical view of relations (predicates) as characteristic or boolean functions, also maintained in RF-Maple; but note that we are not using a functional definition method to define relations, but are endowing a relational definition method (i.e., Horn clauses) with truth-values.

If used in the new way, for function definitions, valued clauses behave similarly to directed conditional equations or conditional term-rewriting rules, where valued rules with N>1 premises can utilize their N-1 first premises not only as conditions but also for accumulating partial results, and then return the value of their Nth premise as the final result.

To reformulate a well-known functional example, using the database

```
((fac 0) 1)                                         ; N=1
((fac _x) (is _y (fac (sub1 _x))) (times _x _y)) ; N=2
```

along with sub1 (predecessor) and times (multiplication) definitions, the goal (fac 3) value-resolves with the second rule by returning the value of its right-most premise, (times _x _y), in an environment where _x is bound to 3 and _y is bound to the recursive fac value, 2, whose computation is terminated by the first rule.

Such functions, like relations, may bind request variables in addition to returning a value. However, the value returned can be an arbitrary term, not just a truth-value. Indeed, in RELFUN this is the <u>only</u> remaining difference between functions and relations, because, once again, functions, like relations, may evaluate non-deterministically. For instance, the valued clauses ((pet-f mary) canary) and ((pet-f mary) pony) define a non-deterministic function pet-f, corresponding to the definition of a relation pet-r by the valued clauses ((pet-r mary canary)) and ((pet-r mary pony)). Now, (pet-f mary) non-deterministically returns canary or pony, just as (pet-r mary _res) non-deterministically binds _res to canary or pony; (pet-f _who), like (pet-r _who _res), additionally binds the request variable _who to mary.

<u>3. Flattening and is-terms</u>

The computation rule of resolution, even in its 'valued' form, can only be applied to goals, not to (nested) terms that are to be evaluated in the usual call-by-value manner. Hence, RELFUN's abstract machine is complemented by another central computation rule, called 'flattening'. The <u>(term) flattening</u> process recursively replaces a main term's non-data subterms by unique variables, generating 'is-terms', which assign the subterms to these variables, and conjoining them to the left of the main term. For example, while the term (g `(h _x) b) is already 'flat' because the subterm `(h _x) is data-passivated by an instantiation prefix, the term (g (h _x) b) is said to be 'nested' because it contains the subterm (h _x) without a "`"-prefix; while the former is usable directly as a goal, the latter must first be flattened to the conjunction (is _1:1 (h _x)) (g _1:1 b), where _1:1 is a unique variable. (In RELFUN, terms are conjoined simply by juxtaposition, and variables are made unique by a 'level' index printed after a ":".)

To illustrate, with g=likes, h=fatherof, and b=mary, the request (likes `(fatherof _x) mary) asks whether some individual <u>denoted</u> by `(fatherof _x), a description of the father of any individual _x, likes mary (fatherof acts as a function in the predicate-calculus sense); if the database contains ((likes (fatherof sue) mary)) as a monolithic fact, this request returns 'true' and binds _x to sue. On the other hand, the request (likes (fatherof _x) mary) asks whether some individual <u>computed</u> as the value of (fatheof _x), an application returning the father of any individual _x, likes mary (fatherof acts as a function in a generalized lambda-calculus sense); if the database contains the rule ((fatherof sue) john) and the fact ((likes john mary)), this request, via flattening to (is _1:1 (fatherof _x)) (likes _1:1 mary), internally binds _1:1 to john and again returns 'true' and binds _x to sue.

Besides such dynamic flattening (at 'run time'), a semantically equivalent but more efficient static flattening (at 'compile time') is also supported by our RELFUN implementation.

RELFUN's is-terms can be viewed as a generalized and purified version of PROLOG's is-primitive or as a 'consistent-assignment' version of the single-assignment statements in functional languages. They may be generated during term flattening (cf. the fatherof example) or may be directly written by the user to name partial results (cf. the fac example). As in many LISP/PROLOG integrations, the right-hand is-sides in RELFUN may be arbitrary terms, not just PROLOG's numerical ones; however, these terms are not evaluated by different machinery (e.g., LISP's eval) but by the normal RELFUN evaluator. For this reason, resolution is prepared to cope with is-goals (is-terms having flat right-hand sides) and flattening is prepared to cope with is-terms still having nested right-hand sides. Since is-goals, like ordinary ones, may resolve with several clauses, non-determinism in functional nestings is thus reduced to non-determinism in relational conjunctions.

For example, if the previous fatherof definition is extended by the rule ((fatherof pat) jack), the nesting (likes (fatherof _x) mary) should non-deterministically evaluate to (likes john mary) or to (likes jack mary); the flattened conjunction (is _1:1 (fatherof _x)) (likes _1:1 mary) realizes this by non-deterministic resolution of the is-goal with the fatherof rules, evaluating to (is _1:1 john) (likes _1:1 mary) or to (is _1:1 jack) (likes _1:1 mary).

The generalized is-primitive can be seen to embody a mapping from the functional 'value world' to the relational 'binding world' that, in a sense, is inverse to instantiation.

After resolution and flattening, a third important computation rule is (value) consumption, which deals with the values returned by functions, e.g., by unifying left-hand is-sides with the values returned by their right-hand function applications. For instance, in the previous example the values john or jack returned by (fatherof _x) are consumed by assigning them to the variable _1:1.

4. Relations, functions, and their integration

A major benefit of an integration approach like the one pursued with RELFUN is the fact that it combines the following -- in our opinion fundamental -- properties of functional and relational programming:

1. Since function calls return values they may be nested: The 'output' of an inner call is directly consumable as one of the 'inputs' of an outer call.

2. Since relation calls bind possible request variables they may be inverted: The 'inputs' and 'outputs' of a call are interchangeable. (For limitations of invertibility see, e.g., [Boley 1983].)

That is, RELFUN calls both return values and bind possible request variables, hence can be nested and inverted. Moreover, in RELFUN a function call may return everything that a relation call may bind and vice versa. In particular, since PROLOG's relation calls may bind a variable to another variable or even to a structure containing variables, RELFUN's function calls may return such non-ground terms (patterns).

As a first example, if not only canary and pony (of section 2) but everything is considered as a pet by mary, we can express this, relationally, with the fact ((pet-r mary _x)) and, functionally, with the rule ((pet-f mary) _x); the latter defines pet-f for the argument mary as what might be called a 'universally non-deterministic' function. Now, (pet-r mary _res) binds _res to (a unique renaming of) the variable _x and (pet-f mary) returns this (renamed) variable. Hence, adding the fact ((likes john pony)), the request "Does john like any pet of mary?" succeeds both as a relational conjunction, (pet-r mary _aux) (likes john _aux), and as a more concise functional nesting, (likes john (pet-f mary)). While the relation pet-r must bind a request variable, _aux, for transporting its internal variable _x:1 to the likes goal, the pet-f function can return its _x:2 to this goal (implemented in RELFUN via a flattening-generated is-variable _1:1).

As a second example, RELFUN can simulate PROLOG's append relation and LISP's append function individually:

```
((append-r nil _s _s)) ; double parens since facts have no premises
((append-r (_h . _t) _s (_h . _r)) (append-r _t _s _r))

((append-f nil _s) _s)
((append-f (_h . _t) _s) (cons-f _h (append-f _t _s)))
((cons-f _h _t) `(_h . _t)) ; the "`" permits free cons-f arguments
```

It can also combine these to an append 'relfuntion' that both binds its last argument to the catenation result and returns this result:

```
((append-rf nil _s _s) _s)                ; equivalent definition:
((append-rf (_h . _t) _s (_h . _r))       ; ((append-rf _u _v _w)
 (append-rf _t _s _r)                     ;  (append-r _u _v _w)
 `(_h . _r))                              ;  _w)
```

This definition can be viewed as an extension of append-r, where the additional premise yields the instantiation of the third argument (the former append-r fact thus becomes an append-rf rule); alternatively, it can be viewed as an extension of append-f, where the additional argument permits a flattening of the cons-f/append-f nesting into an append-rf/"`" conjunction ("`" is also usable directly in append-f, obtaining the flat clause ((append-f (_h . _t) _s) (is _r (append-f _t _s)) `(_h . _r)), with the initial is-premise creating a let-like context for the auxiliary variable _r; cf. _y in fac of section 2).

For the following note that for ground terms the instantiation operator, "`", acts like LISP's quote, "'", and that "id" ('non-terminal identifier') corresponds to PROLOG's 'anonymous variable'. Now, append-rf calls can be both nested (with request-variable binding effects eliminated), like append-f calls, as in [(multiple) arrows indicate (non-deterministic) values]

```
(append-rf `(0) (append-rf `(1 2) `(3) id) id) => (0 1 2 3)
```

and inverted, like append-r calls, as in [commas precede bindings]

```
(append-rf _x _y '(1 2 3)) => (1 2 3), _x = nil,       _y = (1 2 3)
                           => (1 2 3), _x = (1),       _y = (2 3)
                              . . .
                           => (1 2 3), _x = (1 2 3), _y = nil
```

Naturally, (free) logical variables and patterns are also allowed as actual function arguments. Then, just as the append-r call

```
(append-r _x '(_x 2) _o) => true, _x = nil,      _o = (nil 2)
                         => true, _x = (_h:1), _o = (_h:1 (_h:1) 2)
                                  . . .
```

successively <u>binds</u> the request variable _o to the patterns (nil 2), (_h:1 (_h:1) 2), etc., the append-f call

```
(append-f _x '(_x 2)) => (nil 2),               _x = nil
                      => (_h:4 (_h:4) 2), _x = (_h:4)
                         . .
```

non-deterministically <u>returns</u> equivalent patterns, instantiatable by subsequent calls, as in (is (1 . _r) (append-f _x '(_x 2))).

The previous append-r call is equivalent to the conjunction (append-rf _x '(_x 2) _o) true, and the append-f call is equivalent to (append-rf _x '(_x 2) id). In general, append-rf can replace all uses of append-r and of append-f, i.e. for any terms u, v, and w the following equivalences hold:

```
(append-r u v w)  =  (append-rf u v w) true   ; success value:'true'
(append-f u v)    =  (append-rf u v id)       ; return argument: id
```

5. <u>Higher-order functions</u>

Perhaps the most important current issue in relational/functional integration is higher-order functions (and relations), not directly available in extant relational languages, most notably in PROLOG, though playing an increasing role in recent functional languages, most notably in FP. Even in the core field of logic programming, at least in the 'pure' approach represented by Robert Kowalski, the availability of higher-order functions is now viewed as an advantage of functional programming. The present research is based on the conviction that an integrated language should inherit this central functional characteristic [Boley 1983].

In RELFUN, 'elementary' function-forming operators such as composition can be simply defined by valued clauses like

```
(((compose _f _g) _a) (_f (_g _a)))  ; ('(compose list pet-f) mary)
```

New higher-order functions such as twice can be defined from old ones by using the instantiation operator to return functions, as in

```
((twice _f) '(compose _f _f))          ; ((twice fatherof) sue)
```

No 'upward funarg' problem arises because RELFUN, on resolution of a goal like (twice cdr) with (function-valued) clauses like the above, uniquely renames their variables, here _f to _f:1, binds these to the actual arguments, here _f:1 to cdr, and keeps such bindings till the end of the entire computation; thus, the functional value, here `(compose _f:1 _f:1), when applied, will find the (right) values for its variables. The 'downward funarg' problem is banished by the lexical scoping that results from the same unique renaming of variables; for example, the RELFUN version of LISP's mapcar higher-order function

```
((mapper _f nil) nil)
((mapper _f (_h . _t)) (cons-f (_f _h) (mapper _f _t)))
```

and its 'twice mapper' derivate

```
((maptwice _f _a) (mapper `(compose _f _f) _a))
```

correctly evaluate (maptwice sub1 `(3 4 5)) to (1 2 3), because the maptwice call binds its renamed _f variable, _f:1, to sub1 and the mapper call binds its differently renamed _f variable, _f:2, to the functional argument `(compose _f:1 _f:1).

While relational languages could be criticized for using variable renaming even for first-order clauses, their renaming mechanism is also usable for higher-order clauses (see above), thus directly exploiting the alpha-conversion rule of lambda calculus instead of employing additional devices like closures, which dominated functional languages before the advent of combinators. Of course, the 'lambda-variable-eliminating' work of John Backus on combining forms and of David Turner on combinators, when transferred to relational programming, could be interpreted as suggesting that one should get rid of logical variables too. Indeed, variable-free definitions of new higher-order functions from old ones are also possible with RELFUN's valued clauses, as exemplified by the following definition of fourtimes on the basis of twice:

```
((fourtimes) (twice twice))
```

However, the general problem of devising a completely variable-free version of resolution in spite of ordinary unification's dependence on 'consistent-assignment' variables appears to be quite difficult; Alan Robinson's LOGLISP successor project is currently attacking this challenge in relational/functional integration.

6. Conclusion

RELFUN's operational semantics was specified as an abstract machine in pure LISP (incl. I/O statements), on the basis of the PROLOG specification given in [Kahn 1983/84]. The appendix defines the pure RELFUN kernel in a pure subset of FRANZ LISP. This first RELFUN interpreter was augmented by three-valued constants ('true', 'unknown', and 'false'), impure RELFUN primitives (namely, 'call' and 'initial cut'), LISP function access (only after flattening), a refined user interface (incl. a 'list-of-terms tracer'), and efficiency enhancements (mainly the 'static-flattener'). The resulting prototype implementation of RELFUN is running in compiled FRANZ LISP with sufficient efficiency for exploring the new relational/functional programming technique. Since our LISPLOG implementation project also started from Kahn's interpreter [Boley & Kammermeier et al. 1985], RELFUN's efficiency can be further enhanced by adapting the techniques used there (e.g., clause indexing, structure sharing, recursion removal, and compilation).

This is of course mostly standard PROLOG efficiency technology, much of which, however, also applies to the functional features of RELFUN, because its flattening and is-resolution computation rules in a sense reduce functional to relational semantics. Since static flattening does not incur a run-time overhead, the remaining efficiency penality for supporting both functional and relational programming can probably be kept small.

Non-standard (in particular, parallel) methods for enhancing the efficiency of relational and functional languages may also be combined in RELFUN. For example, since RELFUN reduces the evaluation of function arguments to that of conjunctions of is-terms, parallel evaluation of (side-effect-free) arguments may be implemented by (sharing-free) and-parallelism for these conjunctions.

7. References

[Boley 1983] H. Boley: FIT - PROLOG: A Functional/Relational Language Comparison. Universitaet Kaiserslautern, FB Informatik, Interner Bericht 95/83, MEMO SEKI-83-14, Dec. 1983

[Boley & Kammermeier et al. 1985] H. Boley, F. Kammermeier u. die LISPLOG-Gruppe: LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung. Universitaet Kaiserslautern, FB Informatik, MEMO SEKI-85-03, August 1985. Short version in: B. Nebel (Ed.): Papiere zum Workshop Logisches Programmieren und Lisp. TU Berlin, FB Informatik, KIT-REPORT 31, Dec. 1985, pp. 36-53

[Fribourg 1984] L. Fribourg: Oriented Equational Clauses as a Programming Language. J. Logic Programming, 1984:2, pp. 165-177

[Goguen & Meseguer 1984] J. Goguen, J. Meseguer: Equality, Types, Modules, and Generics for Logic Programming. Stanford University, Center for the Study of Language and Information, Report No. CSLI-84-5, March 1984. Also in: J. Logic Programming, 1984:2, pp. 179-210

[Kahn 1983/84] K. M. Kahn: Pure PROLOG in Pure LISP. Logic Programming Newsletter 5, Winter 83/84, pp. 3-4

[Kornfeld 1983] W. Kornfeld: Equality for Prolog. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, pp. 514-519

[Robinson & Sibert 1982] J. Robinson, E. Sibert: LOGLISP: Motivation, Design and Implementation. In: K. Clark, S. A. Taernlund (Eds.): Logic Programming. Academic Press, London, 1982, pp. 299-313

[Sato & Sakurai 1983] M. Sato, T. Sakurai: Qute: A Prolog/Lisp Type Language for Logic Programming. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, pp. 507-513

[Subrahmanyam 1984] P. A. Subrahmanyam, J.-H. You: Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. 1984 International Symposium on Logic Programming, Feb. 1984, Atlantic City, New Jersey, IEEE Computer Society Press, pp. 144-153

[Voda & Yu 1984] P. J. Voda, B. Yu: RF-Maple: A Logic Programming Language with Functions, Types, and Concurrency. Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT 1984, pp. 341-347

APPENDIX: PURE RELFUN IN PURE LISP (e-mail source requests: boley@uklirb.uucp)

The listing below defines pure RELFUN in a pure subset of FRANZ LISP with I/O being effected essentially by lineread/pp-form (read/print macros for transformations like _x <-> (vari x) and '(b b) <-> (inst (b b)) may be added). Each relfun call fixes a database as a list of valued clauses. Inside its (infinite) top-loop the user may type in term conjunctions or a single interactive command, 'more', for requesting additional non-deterministic results. The central and-process function returns the (lineread-listified) next userline -- if non-'more' -- as a success signal, and returns nil -- possibly caused by 'more' -- as a failure signal. 'More' (? _x '(a _x c)) in (((? _h (_h . _t)) '(_h . _t)) ((? _h (id . _t)) (? _h _t)))! For functions like dynamic-flattener and de-inst the term argument may be an arbitrary (RELFUN) s-expression, sometimes equivalent to a (LISP) list; e.g., within (relfun '(((list . _r) _r))) both (list a . (is (_x _x) '(b b))) and (list a is (_x _x) '(b b)) via (is (_x _x) '(b b)) (list a . '(_x _x)) yield (a b b), _x = b.

```
(def relfun (lambda (database) (top-loop (readl) database)))  ; ALL defS "BY VALUE"
(def top-loop
  (lambda (userline database)
    (top-loop (or (and (neq (car userline) 'more)
                       (and-process userline '((bottom)) database 1))
                  (prog2 (pretty-print 'unknown) (readl)))
              database)))

(def and-process
  (lambda (list-of-terms environment database level)
    (let ((term (cond (list-of-terms (car list-of-terms)) (t 'true))))
         (cond ((or (null list-of-terms)
                    (and (null (cdr list-of-terms)) (final-p term)))
                (pretty-print (ultimate-instant (un-inst term) environment))
                (print-bindings environment environment)
                (more-p))
               ((final-p term)     ; NON-is-CONSUMPTION: ABSORB NON-RIGHT-MOST VALUE
                (and-process (cdr list-of-terms) environment database level))
               ((final-is-p term)      ; is-CONSUMPTION: unify THE TWO is-SIDES
                (let ((is-environment
                       (unify (s-patt-is term)
                              (un-inst (s-expr-is term))
                              environment)))
                     (and is-environment
                          (and-process (or (cdr list-of-terms)
                                           (list (list 'inst (s-patt-is term))))
                                       is-environment
                                       database
                                       level))))
               (t (let ((conjunction-goal (dynamic-flattener (un-is term) 1 level)))
                       (cond ((car conjunction-goal) ; FLATTENING: EXTRACT SUBTERMS
                              (and-process (append (car conjunction-goal)
                                                   (cons (gen-is term
                                                                 (cadr
                                                                  conjunction-goal))
                                                         (cdr
                                                          list-of-terms)))
                                           environment
                                           database
                                           (add1 level)))
                             (t (or-process database           ; RESOLUTION: APPLY CLAUSES
                                            database
                                            (cdr list-of-terms)
                                            term
                                            environment
                                            level)))))))))
```

```
(def dynamic-flattener
  (lambda (term varnum level) ; term WAS un-is-ED IN and-process BUT MAY STILL BE A
    (cond ((final-p term) (list nil term)) ; RIGHT-HAND-SIDE is OR ONE NESTED AFTER
          ((is-t term) (list (list term) (list 'inst (s-patt-is term)))) ; MAIN DOT
          ((final-p (car term))
           (let ((conjunction-goal
                  (dynamic-flattener (cdr term) varnum level)))
                (list (car conjunction-goal)
                      (cons (car term) (cadr conjunction-goal)))))
          (t (let ((varstruct (list 'vari varnum level))
                   (conjunction-goal
                    (dynamic-flattener (cdr term) (add1 varnum) level)))
                  (list (cons (mk-is varstruct (car term))
                              (car conjunction-goal))
                        (cons varstruct (cadr conjunction-goal))))))))

(def or-process
  (lambda (database-left database terms-left goal environment level)
    (cond ((null database-left) nil)
          (t (let ((assertion
                    (rename-variables (car database-left) (list level))))
                  (let ((new-environment
                         (unify (de-inst (un-is goal))      ; THIS un-is
                                (car assertion)             ; TAKES CARE OF
                                environment)))              ; is-RESOLUTION
                       (or (and new-environment             ; IN COOPERATION
                                (and-process (append-is goal     ; WITH append-is
                                                        (cdr assertion)
                                                        terms-left)
                                             new-environment
                                             database
                                             (add1 level)))
                           (or-process (cdr database-left) ; assertion'S CONCLUSION
                                       database            ; DIDN'T unify WITH goal
                                       terms-left          ; (NON-APPLICABILITY)
                                       goal                ; OR ITS PREMISES ADDED
                                       environment         ; TO terms-left FAILED
                                       level))))))))       ; (BACKTRACKING)

(def append-is
  (lambda (goal prefix suffix)
    (cond ((is-t goal) (append-patt (s-patt-is goal) prefix suffix))
          (t (append prefix suffix)))))
(def append-patt
  (lambda (patt prefix suffix)
    (cond ((null prefix) (cons (mk-is patt 'true) suffix))
          ((null (cdr prefix)) (cons (mk-is patt (car prefix)) suffix))
          (t (cons (car prefix) (append-patt patt (cdr prefix) suffix))))))

(def unify
  (lambda (x y environment)                    ; WITHOUT OCCUR CHECK
    (let ((x (ultimate-assoc x environment))
          (y (ultimate-assoc y environment)))
         (cond ((equal x y) environment)
               ((or (anonymous-p x) (anonymous-p y)) environment)
               ((vari-t x) (cons (list x y) environment))
               ((vari-t y) (cons (list y x) environment))
               ((or (atom x) (atom y)) nil)
               (t (let ((new-environment (unify (car x) (car y) environment)))
                       (and new-environment
                            (unify (cdr x) (cdr y) new-environment))))))))
(def anonymous-p (lambda (x) (eq x 'id)))
```

```
(def ultimate-assoc
  (lambda (x environment)
    (cond ((vari-t x)
              (let ((binding (assoc x environment)))
                   (cond ((null binding) x)
                         (t (ultimate-assoc (cadr binding) environment)))))
          (t x))))
(def ultimate-instant
  (lambda (x environment)
    (cond ((vari-t x)
              (let ((binding (assoc x environment)))
                   (cond ((null binding) x)
                         (t (ultimate-instant (cadr binding) environment)))))
          ((atom x) x)
          (t (cons (ultimate-instant (car x) environment)
                   (ultimate-instant (cdr x) environment))))))

(def rename-variables
  (lambda (term listified-level)
    (cond ((vari-t term) (append term listified-level))
          ((atom term) term)
          (t (cons (rename-variables (car term) listified-level)
                   (rename-variables (cdr term) listified-level))))))

(def print-bindings
  (lambda (environment-left environment)
    (cond
     ((cdr environment-left)
      (let ((variable (caar environment-left)))
           (and (null (cddr variable))
                (pretty-print
                 (list variable '= (ultimate-instant variable environment)))))
      (print-bindings (cdr environment-left) environment)))))

(def more-p
  (lambda nil
    (let ((response (readl))) (and (neq (car response) 'more) response))))

(def gen-is
  (lambda (term goal)
    (cond ((is-t term) (mk-is (s-patt-is term) goal)) (t goal))))
(def mk-is (lambda (patt expr) (list 'is patt expr)))
(def un-is (lambda (term) (cond ((is-t term) (s-expr-is term)) (t term))))
(def s-expr-is (lambda (isterm) (caddr isterm)))
(def s-patt-is (lambda (isterm) (cadr isterm)))

(def final-is-p (lambda (term) (and (is-t term) (final-p (s-expr-is term)))))
(def final-p (lambda (term) (or (atom term) (inst-t term) (vari-t term))))

(def de-inst
  (lambda (x)
    (cond ((atom x) x)
          ((inst-t x) (cadr x))
          (t (cons (de-inst (car x)) (de-inst (cdr x)))))))
(def un-inst (lambda (x) (cond ((inst-t x) (cadr x)) (t x))))

(def is-t (lambda (x) (and (dtpr x) (eq (car x) 'is))))        ; dtpr IS consp
(def inst-t (lambda (x) (and (dtpr x) (eq (car x) 'inst))))    ; IN MOST LISPS
(def vari-t (lambda (x) (and (dtpr x) (eq (car x) 'vari))))    ; EXCEPT FRANZ

(def readl (lambda nil (prog2 (print '*) (lineread nil))))
(def pretty-print (lambda (x) (prog2 (pp-form x) (terpri))))
```