SEKI-PROJEKT

SEKI MEMO

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

FIT - PROLOG:
A Functional/Relational Language Comparison

Harold Boley

Memo SEKI-83-14 December 1983

FIT - PROLOG: A FUNCTIONAL/RELATIONAL LANGUAGE COMPARISON

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

Abstract

The programming languages FIT and PROLOG are compared as examples of functional and relational programming, respectively. This leads to some proposals concerning both languages.

As an introductory tutorial, PROLOG facts, questions, variables, conjunctions, and rules are reformulated in FIT.

A natural equivalence between functions and relations is exploited for their interchangeable FIT use. An ESCVAL operator is proposed which causes relation calls to return values of request variables and thus permits their function-like nesting. Function calls with request variables are introduced, showing a sense in which FIT functions are more general than PROLOG relations. Higher-order functions and relations are demonstrated to be available in FIT but not in PROLOG.

PROLOG structures and FIT compounds differ mainly in the fixed arity of the former and the variable length of the latter. FIT's compounds can also be interpreted as function calls that return themselves in normalized form.

Pattern matching in PROLOG [FIT] treats list heads and tails asymmetrically [symmetrically] and doesn't [does] allow for non-deterministic results. While PROLOG generalizes pattern-data matching to pattern-pattern unification, FIT generalizes it to adapter-data fitting.

PROLOG's Horn clauses in FIT become implicit fitters: Facts become special implicit adapters and rules become special implicit transformers; for PROLOG II constraints, transformers with LOCAL bodies or invocation adapters with COM[POSE-TRA]FO expressions can be used. While PROLOG interprets clauses in textual order, FIT interprets them in a specificity order which is modifiable by a SECURE operator. Although PROLOG's cut operator is not used in FIT, a proposal is made to distinguish the specification of clause ordering [by FIT's SECURE operator] and the specification of clause abandoning [by an EXCLUSIVE operator corresponding to 'initial'-restricted cuts]. EXCLUSIVE-marked COMFO-constrained rules are then used for functional and relational representations of guarded commands.

A comparison of the list processing capabilities of both languages exemplifies how FIT's adapters can make relational programming more concise than PROLOG's Horn clauses. The representation of sets as lists without duplicates leads to difficulties with PROLOG's standard intersection and union predicates, which can be overcome by representing them as the self-normalizing CLASS data structure in FIT.

Possible reasons for the poor readability of Warren's PROLOG serialise predicate are discussed and an alternative FIT function is formulated which shows the inherent simplicity of this problem. McDermott's PROLOG quadrat predicate is transformed into a more concise and readable ESCVAL form, which in turn is transformed into a corresponding FIT ESCVAL form and into a functional FIT form. Fermat's equation is formulated relationally, showing that for principal reasons some relations can not be used in all ways allowed by PROLOG's notation, a problem that does not arise in a corresponding functional FIT formulation.

Contents

1 INTRODUCTION

This paper attempts to compare in detail the programming languages FIT and PROLOG. It discusses some of their common and distinguishing features and may thus shed some new light on both languages. Hence it addresses readers who are interested in at least one of these languages. The paper can be read as a constructive critique of PROLOG-style predicate logic or relational programming from the standpoint of FIT-style applicative or functional programming. It also shows that FIT can be viewed as an integration of some of PROLOG's relational features with a functional LISP philosophy, at the same time avoiding the criticized PROLOG features.

More precisely, FIT consists of a kernel, pure FIT, and an interactive user-interface, impure FIT; FIT's present implementation is FIT-1. Pure FIT is regarded in principle as a functional language because it is based on purely functional features (Henderson 1980), augmented mainly by

1. Consistent-assignment variables, needed for patterns, which preserve functionality much like the well-known single-assignment variables.

2. Implicit adapters, permitting the direct representation of PROLOG facts and their retrieval using request variables.

Furthermore, since pure FIT-1 is implemented in a purely functional LISP subset, this paper can also be viewed as a preparatory step for a semantic comparison of PROLOG with unaugmented functional languages, like pure LISP, completing the implementation-oriented comparisons from (Warren et al. 1977) to (O'Keefe 1983). Finally, since PROLOG borrowed a lot from PLANNER-like languages, this, in turn, would entail an indirect functional formalization of a subset of PLANNER-like languages, complementing the logical/relational formalization of this subset in PROLOG. Actually, FIT-1 itself can be regarded as a direct functional reorganization of PLANNER, much like "Prolog may be regarded as a logically reorganized Planner" (Fuchi 1982).

The function augmentation of implicit adapters, besides allowing the representation of data base facts, also provides a succinct FIT notation for relation definitions [cf. section 7]. So, when we criticize relational programming, this applies to PROLOG as well as to a relational use of FIT. However, we feel the important thing is that both functional and relational features are available as possibilities in FIT. We dispute the contention that relational programming is 'simply a generalization' of functional programming and our critique centers on the omission of features like higher-order functions in PROLOG [cf. section 3.3].

For PROLOG critiques from other standpoints see (McDermott 1980) [PLANNER-like languages], (Robinson & Sibert 1982) [denotational semantics], (Kurokawa 1982) [software engineering], (Bibel 1983) [unrestricted first-order logic], (Feigenbaum & McCorduck 1983) [knowledge engineering], and (Shapiro 1983) [multi-processing]. Self-contained treatments of PROLOG and FIT can be found in the references of this paper. More references can be obtained from (Fuhlrott 1984) [nearly exhaustive PROLOG bibliography] and the author [complete FIT bibliography]. A global perspective of PROLOG's role in artificial intelligence, in particular in relation to that of LISP, can be found in (Boley 1982/83), which also contains references for all AI languages mentioned in this paper.

In spite of whatever complaints we may have to make about PROLOG in these pages, we do appreciate the excellent work done under the 'logic programming' heading [in particular, Kowalski's pioneering logic studies and the efficiency of Warren's von Neumann PROLOG compiler] and the impact it has had on the Japanese Fifth-Generation Computer Systems endeavour, both of which have strongly increased interest in artificial intelligence in general and in AI languages and machines in particular [as measured by the success of the book (Feigenbaum & McCorduck 1983)]. In our opinion it is still an open question, however, whether PROLOG's traditional orientation toward efficient implementation on available

sequential computers provides an ideal base language for projects in future non-orthodox parallel computer architectures. While the initial inefficiency of McCarthy's LISP implementation of LISP subsequently suggested new machine architectures, the initial efficiency of Colmerauer's FORTRAN implementation of PROLOG subsequently may make it possible to live with existing ones. It is perhaps precisely PROLOG's early efficiency that shows its affinity to von Neumann computers, indicating that it cannot be the right non-von Neumann language.

Striving for precise terminology, we prefer the term 'relational programming' instead of the often-used term 'logic programming' to characterize PROLOG's programming methodology. In our field there is some temptation to adopt 'fashionable terms' uncritically and normally one wouldn't even notice a redundancy like "Programming in PROgramming in LOGic" in a book title. But now, even the principal founder of logic programming has admitted that the present discussion is marked by the confusion of logic programming with PROLOG, logic programming with programming languages, and logic programming with Horn clause programming (Kowalski 1983). Below, we summarize the rationale for our terminological decision.

First, we think that the term 'logic programming' is less suitable because PROLOG's computational mechanisms only overlap with the deductive mechanisms of first-order predicate calculus:

1. PROLOG uses "extra-logical features" (VanEmden 1980) such as the cut operator and many other built-in predicates going far beyond first-order logic (McDermott 1980): "But perhaps PROLOG will take the world by storm and perhaps logic programming will be forgotten ..." (VanEmden 1980).

2. There are first-order formulas, such as those involving disjunction (Bowen 1982) and negation (Aida et al. 1983), which are not provable with PROLOG's Horn clause programming but only with "full first-order logic programming" (Bowen 1982).

Second, we think that the term 'relational programming' is more appropriate mainly because of two reasons:

1. The characteristic programming language feature of PROLOG is the transformation of relations, even for computing deterministic functions [for which earlier PLANNER-like languages resorted to LISP].

2. PROLOG can be regarded as an enrichment of relational data base systems by deductive relation retrieval.

Although there are many dialects of PROLOG, the most well-known and wide-spread version is that developed in Edinburgh, and we will base our comparison on this. Thus in the following the unqualified term 'PROLOG' will stand for 'Edinburgh PROLOG as described in (Clocksin & Mellish 1981)'.

At least those PROLOG examples not quoted from the literature have been tested, principally in DEC-10 Edinburgh PROLOG and in a few cases in micro-PROLOG and LOGLISP. The FIT examples not introduced as still

unimplemented suggestions have been tested in DEC-10 FIT-1, which is itself running in UCI LISP.

The following section [2] is a tutorial introduction which may be skimmed by readers who want to get to the essentials quickly or who already have some knowledge of PROLOG and FIT. Section 3 contains the central points of the discussion 'functional vs. relational', including relations that return values [ESCVAL operator], functions with request variables, and higher-order functions; it anticipates some of the material treated more extensively in later sections. The short section 4 deals with the data structures of both languages and may be skimmed by all those familiar with LISP, FIT, or PROLOG. Section 5 discusses pattern matching as needed for fact retrieval and rule invocation, including its unification [PROLOG] and fitting [FIT] generalizations. Then, section 6 treats clauses [facts and rules] and their constraints restriction, textual/specificity ordering, and cut/EXCLUSIVE/SECURE marking. The penultimate section [7] compares list/set processing in PROLOG and FIT and also demonstrates the use of FIT's adapter-driven computation for relational programming. Finally, the last section [8] gives more detailed examples [acknowledging Warren, McDermott, and Fermat], but also points to a number of further principal issues in functional/relational programming.

2 A TUTORIAL COMPARISON OF FIT AND PROLOG

This introductory comparison is based on the tutorial introduction in chapter 1 of the standard PROLOG textbook (Clocksin & Mellish 1981). It covers all the PROLOG features of this introductory chapter or, as the authors call it, of the "basic core of PROLOG". Some advanced PROLOG features are not discussed in this introductory comparison, but are treated in the remainder of this paper. Although the comparison can be regarded as an introduction to FIT for readers acquainted with PROLOG, it is not a general introduction to that language, because it concentrates on PROLOG-related FIT features. All PROLOG examples are taken from (Clocksin & Mellish 1981), sometimes with minor extensions; the subheadings are cited unchanged from this source.

2.1 Facts

A fact like "John likes Mary" in PROLOG is regarded as a relation, likes, that holds between two individuals; it is written as likes(john,mary) and is stored by a "."-terminated statement

```
likes(john,mary).
```

[Thus the period is part of the object language, PROLOG, not part of the meta language, English; to avoid confusion of language levels we will always omit meta-language punctuation after object-language expressions displayed between two blank lines.]

In FIT the fact is regarded as a [predicate] function, LIKES, which is 'true' for the two individuals; it is written as (LIKES JOHN MARY) and is stored by a unary GLOBAL expression whose argument is a one-element list containing the fact, i.e. by

```
GLOBAL:((LIKES JOHN MARY))
```

[The mathematical function/relation notation, f(a1,a2,...,aN), for LISP function calls and, more generally, FIT fitments is rewritten as a list with a distinguished first element f, (f a1 a2 ... aN), which in FIT for N=1 may be abbreviated to f:a1, i.e. in the example, with f=GLOBAL and a1=((LIKES JOHN MARY)), the parentheses are part of the argument rather than the call notation.]

Whereas in PROLOG "." is just a syntactical terminator, which in this context serves as a top-level cue to invoke the storing routine, in FIT GLOBAL is the storing function which can be called from any level.

To create a four-element [n-element] data base in PROLOG one must write four [n] "."-terminated relations like

```
valuable(gold).
female(jane).
owns(john,gold).
father(john,mary).
```

while in FIT one may use a four-element [n-element] list argument of a single GLOBAL expression

```
GLOBAL:((VALUABLE GOLD)
        (FEMALE JANE)
        (OWNS JOHN GOLD)
        (FATHER JOHN MARY))
```

as an alternative to four [n] corresponding GLOBAL expressions. PROLOG does not allow the storage of several data base facts as a single operation [the above PROLOG use of "." can be regarded as a postfix operator corresponding to the built-in assertz predicate; cf. ASSERT in PLANNER-like languages]. Instead, each fact must be stored individually, which has been reported to be a common source of syntax errors (Clocksin & Mellish 1981). On the other hand, in FIT the use of GLOBAL's additional pair of parentheses is not obvious when storing individual facts but becomes apparent when storing an entire data base. The deeper reason for this general form of GLOBAL is its use as a semantic primitive for multiple definition side-effects [for example, the pattern match (>X >Y :) generates two binding side-effects (>X) and (>Y), which are represented as GLOBAL:((>X) (>Y)); cf. section 5.1]. In order to avoid parenthesis omission errors when storing individual facts, a simple FIT extension

```
ASSERT:fact = GLOBAL:(fact)
```

could be defined.

Unlike in PROLOG, in FIT facts need not be stored globally but can also be stored locally, creating 'local data bases', by using the LOCAL instead of the GLOBAL storage operator, as exemplified in the next subsection and exploited as a module feature in section 7.1.

2.2 Questions

Presupposing the above global 'Mary' fact, a PROLOG question/answer sequence [to distinguish user questions from computer answers in such sequences, the answers will be underlined here and below]

```
?- likes(john,mary).
yes
```

in FIT becomes

```
(LIKES JOHN MARY)
(LIKES JOHN MARY)
```

Thus instead of printing a simple 'yes' or 'true', FIT follows a good PLANNER tradition and returns the instantiated form of requested facts [in such simple cases as above, this is identical to the question; but see the next subsection]. The question in FIT is regarded as a call of the LIKES predicate function with two arguments, JOHN and MARY, the only pair of arguments for which that predicate has so far been defined by a GLOBAL expression. While the PROLOG likes request just prints its answer 'yes', the FIT LIKES call returns its answer (LIKES JOHN MARY) as a function value which can be further processed by other function calls; for example, the nested call (CDR (LIKES JOHN MARY)) uses a CDR call to return the tail (JOHN MARY) of the result of the LIKES call. LIKES can be regarded as a predicate function, although the returned expression (LIKES JOHN MARY) is not equal to the truth-value "T" for 'true', because in FIT every expression not denoting 'false' or 'unknown' is interpreted as being 'true' [this corresponds to LISP's non-NIL = 'true' convention].

Again presupposing the above 'Mary' fact, the LOCAL data base question

```
(LOCAL ((VALUABLE SILVER) (LIKES JOHN JANE))
       (LIKES JOHN JANE) (LIKES JOHN MARY) (VALUABLE SILVER))
```

would return

```
(LIKES JOHN JANE) (LIKES JOHN MARY) (VALUABLE SILVER)
```

since all these facts are stored, the first and third locally, the second globally.

Summarizing the syntax introduced in these first two subsections, while PROLOG prefixes questions [with "?-"] and interprets unprefixed expressions as the assertion of facts, FIT prefixes assertions [with "GLOBAL:"] and interprets unprefixed expressions as questions.

2.3 Variables

The PROLOG facts and questions containing variables

```
likes(john,flowers).
likes(john,mary).
likes(paul,mary).
```

```
?- likes(john,X).
X=flowers

?- likes(X,mary).
X=john;
X=paul;
no
```

in pure FIT [here we assimilate the presentation of non-determinism to PROLOG's treatment] become

```
GLOBAL:((LIKES JOHN FLOWERS)
        (LIKES JOHN MARY)
        (LIKES PAUL MARY))

(LIKES JOHN |?X)
(GLOBAL ((>X FLOWERS)) (LIKES JOHN FLOWERS))

(LIKES |?X MARY)
(GLOBAL ((>X JOHN)) (LIKES JOHN MARY))
MORE
(GLOBAL ((>X PAUL)) (LIKES PAUL MARY))
MORE
jU
```

To distinguish variables from individuals, PROLOG uses a capitalization convention [inverse to the standard mathematical convention, as remarked in (Robinson & Sibert 1981) and corrected in LOGLISP] while FIT marks single-value-accepting variables by a "?" [SHOVEONE] prefix, multiple-value-accepting variables by a ">" [SHOVE] prefix, and open request variables by an additional "|" [VERTICAL] prefix. As answers to successful questions containing variables, PROLOG prints variable bindings 'X=flowers' etc. while pure FIT returns binary GLOBAL expressions (GLOBAL ((>X FLOWERS)) (LIKES JOHN FLOWERS)) etc. with the bindings in their first argument [internally, always the more general ">" prefix is used] and the instantiated expression in their second argument. Like user-initiated GLOBAL expressions these system-generated ones in impure FIT store their bindings in the data base; all GLOBAL expressions also return their second argument, which for unary GLOBALs is the empty imposition [an imposition is a possibly empty sequence of expressions]. For example, after pure FIT has evaluated (LIKES JOHN |?X) to (GLOBAL ((>X FLOWERS)) (LIKES JOHN FLOWERS)), impure FIT sets X to FLOWERS and returns (LIKES JOHN FLOWERS). Thus, while in PROLOG the bindings are just printed and gone in the next interaction step, in FIT they are stored in the global data base for later use [the recent LM-PROLOG (Kahn 1983) also has a facility for saving bindings until the next interaction step]. As in (Winston & Horn 1981), the "<" [PULL] prefix is used in FIT to fetch variable values, e.g., the value a variable received as the result of a previous interaction step. This allows the incremental interactive construction of answers to compound questions as in

```
(LIKES PAUL |?X); first give me the entity X that Paul likes
(GLOBAL ((>X MARY)) (LIKES PAUL MARY))
```

```
(LIKES JOHN <X); second check to see if John also likes that entity X
  (LIKES JOHN MARY); internal reformulation asking if John likes Mary
(LIKES JOHN MARY))
```

By now the naturalness of returning instantiated questions as answers should have become apparent: We asked FIT to find individuals replacing the variable X in propositional forms thus making them true propositions, and it returned these true propositions together with their X-bindings.

If global binding effects are not desired, request variables can be localized using LOCAL expressions. In a basic LOCAL form the request variables are listed in the first argument and the question appears in the second argument. The following question-answering sequence is an example [X is locally initialized with the empty imposition]:

```
(LOCAL (>X:) (LIKES PAUL |?X))
(LIKES PAUL MARY)
```

In an advanced LOCAL form the question itself is written to the left of a colon separator and an arbitrary expression making use of the request variables to the right of the ":". An example is this question-answering:

```
(LOCAL (LIKES PAUL |?X)
       :
       (APPEND '(HE LIKES) (LIST <X) '(AS FAR AS I KNOW)))
(HE LIKES MARY AS FAR AS I KNOW)
```

Returning to the first example of this subsection, the 'carry on' command use of PROLOG's ";" operator in FIT translates to MORE commands. PROLOG's 'no' responses for indicating failures in FIT often become jU failure signals [jump 'unknown'] rather than the literally corresponding jF failures [jump 'false'].

Excursus: The Closed-world Assumption

The reason for this is that in FIT the 'closed-world-assumption' [here implying that the system knows all about who likes Mary] is not built in. As another example, consider the FIT query (LIKES FRED MARY) and the corresponding PROLOG query likes(fred,mary) in the above respective data bases, in which no 'likes' relationship is stored for Fred: In FIT it yields jU ['I don't know'], while in PROLOG it prints 'no' ['I assume no']. This is because FIT, by default, regards facts as open-ended information about relations [e.g., 'likes'], while PROLOG assumes facts to completely define these relations. Instead of relying on a universal closed-world assumption, the FIT user may 'close off' each predicate individually if its clauses are to be regarded as 'definitional', so that the system will give negative information [jF] only for requests with that predicate for which no normal clause is successful. In LOGLISP, the user can also declare a predicate-restricted closed-world assumption [we may call this a 'closed-predicate specification], but must do this by applying the LISP function NULL to the result of a call to the LOGLISP procedure ANY (Robinson & Sibert 1981). For example, the LOGLISP definition

```
(NOT (LIKES x y)) <- (NULL (ANY 1 T (LIKES x y)))
```

would close off the LIKES relation. In FIT a closed-predicate specification belongs to the completely normal way of defining predicate functions: A clause with a minimally specific head pattern defines the predicate to be jF, so that this clause is used if and only if no other matching one with that predicate remains untried. For example, the FIT definition

```
(>(LIKES ?X ?Y) jF)
```

would also close off the LIKES relation [it sets the 'compound variable' (LIKES ?X ?Y) to the value jF, which, when typed in, should normally be quoted like 'jF]. The LOGLISP and FIT systems on the basis of these definitions would know all about who likes whom but make no assumptions about other relationships. For instance, in the previous data base this would cause the FIT query (LIKES FRED MARY) and similar ones like (LIKES FRED BILL) to yield jF but would not change a jU yielded by queries with other relations like (SISTER_OF FRED MARY). The closed-world assumption can be restricted even further to predicates with some given fixed arguments. For example, the FIT definition

```
(>(LIKES ?X MARY) jF)
```

would close off the LIKES relation for a second argument equal to MARY only; the system on the basis of this definition would know all about who likes Mary but make no assumptions about who likes other persons. This is sufficient for obtaining jF for the query (LIKES FRED MARY), but not for obtaining jF for the similar query (LIKES FRED BILL) [the system would modestly reply jU]. In general, FIT allows restricting closed-predicate specifications to exactly the scope required.

The predicates of 'closed subworlds' [e.g., of list processing; cf. section 7] can be closed off by a single definition

```
(>(CLOSEDPRED >X) jF)
```

provided that the second-order predicate CLOSEDPRED is 'true' for them. [For instance, the second-order definition (CLOSEDPRED MEMBER) could be used instead of the first-order definition (>(MEMBER ?X ?Y) jF) for closing off the MEMBER predicate; further CLOSEDPRED definitions could be used for closing off the other predicates in this paper.]

Incidentally, it is FIT's three-valued logic which permits a differentiation of what is known to be true, what is known to be false, and what is unknown, while PROLOG's two-valued logic leads to a confusion of the latter two categories. Although the closed-world assumption gives rise to certain nice formal properties [cf. the recent paper (Jaffar et al. 1983)], its practical usefulness is questionable. It enforces a narrow world view in PROLOG-based systems because what they actually assume is "All that I haven't heard of cannot be true". Presumably, it would not be prudent to endow future computer systems with such a built-in illusory assumption of omniscience. Another recent critique of the closed-world assumption of ordinary PROLOG may be found in (Hewitt &

de Jong 1983). Ironically, while Hewitt has abandoned his PLANNER tradition in this respect, Kowalski is still cultivating it (Kowalski 1983).

Perhaps it was the special syntactical position of predicates in the mathematical/logical notation R(a1,...,aN) for applications/relations, in contrast to LISP's modern Cambridge Polish prefix notation (R a1 ... aN), that prevented PROLOG from allowing questions asking for the predicate, using predicate variables [indeed micro-PROLOG, the only well-known PROLOG dialect which has some means of asking for predicates, resorts to its LISP-like "internal syntax" (Clark et al. 1982) for that purpose, as shown below]. Perhaps it was a fear of losing the semantics of first-order predicate calculus when permitting implicit request quantifiers ranging over predicates instead of over individuals only. And/or perhaps efficiency considerations were involved, because such requests cannot make use of a primary predicate indexing of facts. In FIT's attempt to permit what the user finds natural we allow such requests. For example, in the above data base we obtain

(|?X JOHN MARY)
(GLOBAL ((>X LIKES)) (LIKES JOHN MARY))

The natural-language paraphrase of this question, "Is there some relationship between John and Mary?", doesn't sound less natural than "Is there an entity that likes Mary?", the paraphrase of our previous request (LIKES |?X MARY). There are no syntactical problems with this when using Cambridge Polish prefix notation. The direct equivalence with first-order predicate calculus cannot be maintained anyway because higher-order constructs like mapping functions are indispensable [cf. section 3.2]. The indexing problems are easily solvable on the basis of current data base technology; indeed already LEAP (Feldman et al. 1972) allowed asking for all components of associative triples and PLANNER-like languages allow asking for all components of assertion n-tuples, implemented, e.g., by means of "coordinate indexing" (Rulifson et al. 1972).

In micro-PROLOG the extra-logical auxiliary dictionary program must be used for simulating such requests (Clark et al. 1982):

Wh(x (dict x)(x John Mary))
Answer is likes

Moreover, the "meta-variable" x used here is not a true request variable for predicates since it must be bound through the dict call by the time micro-PROLOG evaluates (x John Mary).

The variables used previously are typeless, as they always are in PROLOG, but only by default in FIT. Typed variables can be specified in FIT as follows. Every predicate pred may be used as a typed variable x?pred or x>pred, a value-accepting variable with an additional "x" [XAMINE] prefix. For example, x?FEMALE can only be bound to individuals for which the predicate FEMALE is true and x>LIKES can only be bound to pairs of individuals which are in a LIKES relationship.

2.4 Conjunctions

In the PROLOG data base

```
likes(mary,food).
likes(mary,wine).
likes(john,wine).
likes(john,mary).
```

the request conjunction [the ", " is used as an AND infix operator]

```
?- likes(john,mary), likes(mary,john).
```

is processed from left to right, the first goal succeeding and the second failing, so that the conjunction fails. In the corresponding FIT data base

```
GLOBAL:((LIKES MARY FOOD)
        (LIKES MARY WINE)
        (LIKES JOHN WINE)
        (LIKES JOHN MARY))
```

we can use an implicitly AND-connected imposition

```
(LIKES JOHN MARY) (LIKES MARY JOHN)
```

which also fails because the expression (LIKES MARY JOHN) does.

The question "Is there anything that Mary and John both like?", exemplifying conjunction-wide request variables, in PROLOG becomes

```
?- likes(mary,X), likes(john,X)
```

and is processed using backtracking as follows:

1. The first goal likes(mary,X) matches the first fact likes(mary,food), binding X to food and marking the place of this fact in the data base.

2. The instantiated second goal likes(john,food) fails, so backtracking occurs, i.e. X becomes unbound and the previous goal is tried again, starting from after the marked fact.

3. The first goal likes(mary,X) now matches the second fact likes(mary,wine), binding X to wine and marking that fact's place.

4. The instantiated second goal likes(john,wine) matches the third fact, marking its place.

5. Since both goals are satisfied 'X=wine' is printed.

In FIT the request conjunction can either be partitioned interactively as exemplified in the previous subsection or it can be written as the imposition

```
(LIKES MARY |?X) (LIKES JOHN |?X)
```

which is evaluated without backtracking thus:

1. the first goal (LIKES MARY |?X) yields a BREADTH expression containing all its matching facts [namely the first and second one], and simultaneously also the second goal (LIKES JOHN |?X) yields a BREADTH of all its matching facts [namely the third and fourth one], altogether yielding the intermediate imposition of BREADTH expressions

```
(BREADTH (GLOBAL ((>X FOOD)) (LIKES MARY FOOD))
         (GLOBAL ((>X WINE)) (LIKES MARY WINE)))
(BREADTH (GLOBAL ((>X WINE)) (LIKES JOHN WINE))
         (GLOBAL ((>X MARY)) (LIKES JOHN MARY)))
```

2. In combining the BREADTH results of both goals three candidate results

```
  (GLOBAL ((>X FOOD)) (LIKES MARY FOOD))
  (GLOBAL ((>X WINE)) (LIKES JOHN WINE))

  (GLOBAL ((>X FOOD)) (LIKES MARY FOOD))
  (GLOBAL ((>X MARY)) (LIKES JOHN MARY))

  (GLOBAL ((>X WINE)) (LIKES MARY WINE))
  (GLOBAL ((>X MARY)) (LIKES JOHN MARY))
```

   are rejected because of inconsistent X bindings and only one result,

```
(GLOBAL ((>X WINE)) (LIKES MARY WINE))
(GLOBAL ((>X WINE)) (LIKES JOHN WINE))
```

   remains, so that the result

```
(GLOBAL ((>X WINE)) (LIKES MARY WINE) (LIKES JOHN WINE))
```

   is returned.

Thus FIT abolishes depth-oriented, chronological backtracking in favour of breadth-oriented, non-chronological parallelism, avoiding a host of problems that plague PROLOG [not just beginning with the "cut"] from the start. Backtracking within a sequential conjunction on a sequential data base is perhaps PROLOG's most unfortunate [von Neumann] deviation from pure logic. Rather than regarding a data base of clauses [facts and rules] as a set, which, because it is unordered, has the crucial advantage of modularity, PROLOG regards it as an ordered collection, pointed to by place-markers which are pushed back and forth on it as if it were a SNOBOL string. "Paper-and-pencil simulations" (Clocksin & Mellish 1981) are required to keep track of what's going on.

The annoying difficulty with such a sequential data base can be seen in the example. The order of the four 'likes' facts [shortened, 1-2-3-4], i.e. the sequence of typing them in, first seemed to be immaterial in (Clocksin & Mellish 1981), as in logic, but now it becomes apparent that it has a profound impact on backtracking and efficiency [sometimes even on termination, i.e. total correctness,

because depth-first search may diverge into an infinite subtree although a solution exists somewhere else in the search tree]: Had we typed in the first two facts in reverse order [i.e., 2-1-3-4] no backtracking would have occurred at all. A similar reordering of the last two facts [i.e., 1-2-4-3] would also change backtracking behavior. However, regroupings [here, 3-4-1-2] or even interleavings [e.g., 1-3-2-4] would have no behavioral effect.

Avoiding such "arbitrary sequencing" (Leavenworth & Sammet 1974) in space and time, FIT follows predicate calculus in not imposing an arbitary order onto the data base items. It makes available all facts matching a question at once, as an explicit conflict set, and uses a 'most specific first' rule for "conflict resolution" (McDermott & Forgy 1978). If, as in the example, all facts are equally specific they form a BREADTH expression which can be processed by "OR parallelism" [(Conery & Kibler 1981), (Clark & Taernlund 1982)].

Another sequencing which causes PROLOG to deviate from predicate calculus is the left-to-right order imposed on conjunctions. Like LISP's AND this can be used to simulate 'if then' statements and other desired orderings; it is also available as an option, called ANDTH, in FIT. However, in the example, as is usually the case, we preferred to retain the non-sequenced meaning of logical conjunction by using simultaneously evaluating impositions. These can be processed by "AND parallelism" [(Conery & Kibler 1981), (Clark & Taernlund 1982)].

In FIT, if we do wish to use the bindings of a request variable produced in a first request inside a second request, we can replace occurrences of the request variable |?var in the second request by occurrences of a corresponding PULLTEMPORARY variable ^var, where "^" fetches the value that a variable received in an ongoing evaluation. For example, instead of our previous request imposition, we could write

```
(LIKES MARY |?X) (LIKES JOHN ^X)
```

Since data flow in FIT is not restricted to the direction 'left to right' but may as well proceed 'right to left', the request could also be replaced by

```
(LIKES MARY ^X) (LIKES JOHN |?X)
```

In order to transcribe PROLOG conjunctions literally into FIT, however, a left-to-right pass would be used, in which a PROLOG variable not yet encountered is replaced by a "|?"-variable and one already encountered by a "^"-variable.

2.5 Rules

A PROLOG rule like

```
likes(john,X) :- likes(X,wine)
```

in FIT can be rewritten as

```
(>(LIKES JOHN ?X) (LIKES <X WINE))
```

That is, in FIT a rule has the form of a variable-value association. It is generated by setting a compound variable like (LIKES JOHN ?X) to a quoted value like '(LIKES <X WINE). Typing in the setting

```
(>(LIKES JOHN ?X) '(LIKES <X WINE))
```

abbreviates [and, in FIT-1, temporarily expands to]

```
(GLOBAL ((>(LIKES JOHN ?X) (LIKES <X WINE))) '(LIKES <X WINE))
```

which actually stores (>(LIKES JOHN ?X) (LIKES <X WINE)) and returns (LIKES <X WINE).

Using this rule together with Mary's likings in the data base of the previous section, the PROLOG request

```
likes(john,mary)
```

is matched by the rule head likes(john,X), binding X to mary and marking its place in the data base. Then its instantiated body likes(mary,wine) is turned into another request, which is directly matched by a data base fact so that 'yes' is printed.

In FIT, rules are dealt with similarly, except that their pattern-directed invocation is treated completely within the FIT formalism itself. Intermediate computations like those for invocation matching may be observed in FIT-1's trace mode, which we will indicate as a sequence of indented expressions [as traces may be switched off in FIT-1, the I/O-oriented reader may ignore indented expressions here and later on]. Thus the corresponding question-answering becomes

```
(LIKES JOHN MARY)
  (LOCAL (LIST (LIKES LIKES) (JOHN JOHN) (?X MARY))
         :
         (LIKES <X WINE))
  (LOCAL ((>X MARY)) (LIKES <X WINE))
  (LOCAL ((>X MARY)) (LIKES MARY WINE))
(LIKES MARY WINE)
```

Pattern-directed invocation generates a LOCAL expression [cf. subsection 2.3] with the invocation match in its bindings [before the ":"] and the rule body as its body [after the ":"]. Here, the match is successful, yielding a simpler LOCAL in which X is bound to MARY. The body is evaluated inside this LOCAL scope and the successful result (LIKES MARY WINE) causes the LOCAL and its binding to disappear.

Note that FIT 'overanswers' the original question "Does John like Mary?" in returning not simply 'yes' but an expression interpretable as "Yes because Mary likes wine". The returned expression (LIKES MARY WINE) encodes the reason why the answer to the question (LIKES JOHN MARY) is 'true'. The non-false-and-non-unknown = 'true' convention permits regarding the answer expression as a simple 'true' answer and going into the expression and analyzing the reason for its being 'true' only if/when desired.

If we also presupposed John's likings in the data base of the previous section, we could answer the question without any rule by using a fact. However, since PROLOG uses clauses in textual order, it

would only apply the fact first if it were stored in front of the rule; otherwise it would still first use the rule. In FIT, on the other hand, the order of definitions in store is immaterial because the definitions matching a request are considered in the order of their specificity. Thus the fact in any case would be used first and the rule second, so that we would obtain a DEPTH expression [the ordered counterpart to BREADTH]

```
(DEPTH (LIKES JOHN MARY) suspension-generating-our-previous-result)
```

whose first element shows us that John likes Mary directly and whose suspended second element, only popped up and activated by a MORE request, would show us that he likes her because of her liking for wine, as discussed above.

The notation of rules with conjunction bodies should be clear from the foregoing. For example the PROLOG rule

```
likes(john,X) :- likes(X,wine), likes(X,food)
```

in FIT becomes

```
(>(LIKES JOHN ?X) (LIKES <X WINE) (LIKES <X FOOD))
```

Finally, let us consider rules with conjunction bodies containing conjunction-wide request variables. A PROLOG rule like

```
sister_of(X,Y) :-
            female(X),
            parents(X,M,F),
            parents(Y,M,F).
```

can be rewritten in FIT as

```
(>(SISTER_OF ?X ?Y)
  (LOCAL (>MOTH: >FATH:)
         (FEMALE <X)
         (PARENTS <X |?MOTH |?FATH)
         (PARENTS <Y |?MOTH |?FATH)))
```

F shouldn't be used as a variable in FIT because, as in some LISP's, it is the constant meaning 'false'; therefore we replace F and M by the more mnemonic FATH and MOTH, respectively. Since these variables are not 'formal parameters' of the rule, we have to declare them LOCAL in FIT if we don't want them to spread globally. In PROLOG all variables in rule bodies are treated alike, namely as 'logical variables' which, even if they spread globally, cannot collide because they are uniquely renamed.

Excursus: Interactive Programming

The PROLOG design decision to perform such read-time renaming, however, is detrimental to interactive programming. When typing in a clause interactively the PROLOG system changes ones mnemonic variable names under ones fingers into meaningless "_"-prefixed numbers. The meanings in variable names cannot be recovered by the 'listing' predicate which pretty-prints them as alphabetic ordinals "A", "B", "C", ... denoting, respectively, the first, second, third,

... variable used in a clause. For example, when typing in the above sister_of rule exactly in the form of (Clocksin & Mellish 1981), using their mnemonics M and F for mother and father, respectively, PROLOG forces the programmer to reconceptualize this as

```
sister_of(A,B) :-
   female(A),
   parents(A,C,D),
   parents(B,C,D).
```

One must accept the machine's abstract isomorphism between these sister_of rules and use its meaningless variable names, as if seeing which variable occurrences are used for input, for output, and/or for intermediate results weren't already hard enough with mnemonics. If you trace sister_of calls using the 'spy' predicate, what you see is not even "A", "B", "C" but something like "_24", "_109", "_110". Certainly, this treatment of variables is not a high-level feature of PROLOG. When you dump an interactively constructed program using the 'tell' and 'listing' predicates you, of course, also have the alphabetic ordinals in your file.

The only remedy is to prepare source files with an editor outside PROLOG and then reading such files into PROLOG in their entirety. But that isn't interactive programming: For each little change you have to leave the PROLOG system, enter the editor, make the change, restart PROLOG, and read in the affected file. Nor does it solve all problems: You still have to accomodate to alphabetic ordinals if you want to look at the definition of a clause during the interactive session; the traces still use these underscore numbers. Therefore newer PROLOG developments try to correct this fault, acknowledging the fact that throwing the user's variable names out of main memory was too high a price for gaining computer efficiency.

The 'sister_of' rule together with a data base describing some family relationships of Queen Victoria,

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward,victoria,albert).
parents(alice,victoria,albert).
```

permits PROLOG questions such as

```
?- sister_of(alice,edward).
```

which is processed thus: The question is matched by the rule head sister_of(X,Y), binding X to alice and Y to edward. Then the body conjuncts female(alice) followed by parents(alice,M,F) are satisfied, the latter binding M to victoria and F to albert. Finally, the instantiated goal parents(edward,victoria,albert) succeeds, answering 'yes'.

With FIT's version of the 'Victoria' data base

```
GLOBAL:((MALE ALBERT)
        (MALE EDWARD)
        (FEMALE ALICE)
        (FEMALE VICTORIA)
        (PARENTS EDWARD VICTORIA ALBERT)
        (PARENTS ALICE VICTORIA ALBERT))
```

the corresponding FIT question-answering process can be traced to show the following details:

```
(SISTER_OF ALICE EDWARD)
  (LOCAL (LIST (SISTER_OF SISTER_OF) (?X ALICE) (?Y EDWARD))
         :
         (LOCAL (>MOTH: >FATH:)
                (FEMALE <X)
                (PARENTS <X |?MOTH |?FATH)
                (PARENTS <Y |?MOTH |?FATH)))
  (LOCAL (>X:ALICE >Y:EDWARD >MOTH: >FATH:)
         (FEMALE <X)
         (PARENTS <X |?MOTH |?FATH)
         (PARENTS <Y |?MOTH |?FATH))
  (LOCAL (>X:ALICE >Y:EDWARD >MOTH: >FATH:)
         (FEMALE ALICE)
         (PARENTS ALICE |?MOTH |?FATH)
         (PARENTS EDWARD |?MOTH |?FATH))
  (LOCAL (>X:ALICE >Y:EDWARD >MOTH: >FATH:)
         (FEMALE ALICE)
         (GLOBAL (|>MOTH:VICTORIA |>FATH:ALBERT)
                 (PARENTS ALICE VICTORIA ALBERT))
         (GLOBAL (|>MOTH:VICTORIA |>FATH:ALBERT)
                 (PARENTS EDWARD VICTORIA ALBERT)))
  (LOCAL (>X:ALICE >Y:EDWARD >MOTH: >FATH:)
         (GLOBAL (|>MOTH:VICTORIA |>FATH:ALBERT)
                 (FEMALE ALICE)
                 (PARENTS ALICE VICTORIA ALBERT)
                 (PARENTS EDWARD VICTORIA ALBERT)))
  (LOCAL (>X:ALICE >Y:EDWARD >MOTH:VICTORIA >FATH:ALBERT)
         (FEMALE ALICE)
         (PARENTS ALICE VICTORIA ALBERT)
         (PARENTS EDWARD VICTORIA ALBERT))
(FEMALE ALICE)
(PARENTS ALICE VICTORIA ALBERT)
(PARENTS EDWARD VICTORIA ALBERT)
```

Notice how the invocation-generated LOCAL and the explicit LOCAL body merge when the invocation match ends successfully, how the two request-generated GLOBALs migrate upward and join successfully because of their consistent variable bindings, and how the GLOBAL bindings of MOTH and FATH are trapped in the LOCAL. Again, FIT overanswers the question: instead of 'yes' an imposition of the three instantiated subgoals is returned; their conjunction explains why the answer is 'true'.

In PROLOG in the previous data base the question

```
?- sister_of(alice,X).
```

is treated similarly, but prints 'X=edward' instead of 'yes'. However,

now the sister_of rule allows a possibly unwanted second answer, 'X=alice'.

The following trace gives an analysis of how the corresponding question is answered in FIT:

```
(SISTER_OF ALICE |?X)
  (LOCAL (LIST (SISTER_OF SISTER_OF) (?X ALICE) (?Y |?X))
         :
         (LOCAL (>MOTH: >FATH:)
                (FEMALE <X)
                (PARENTS <X |?MOTH |?FATH)
                (PARENTS <Y |?MOTH |?FATH)))
  (LOCAL (>X:ALICE >Y:|?X >MOTH: >FATH:)
         (FEMALE <X)
         (PARENTS <X |?MOTH |?FATH)
         (PARENTS <Y |?MOTH |?FATH))
  (LOCAL (>X:ALICE >Y:|?X >MOTH: >FATH:)
         (FEMALE ALICE)
         (PARENTS ALICE |?MOTH |?FATH)
         (PARENTS |?X |?MOTH |?FATH))
  (LOCAL (>X:ALICE >Y:|?X >MOTH: >FATH:)
         (FEMALE ALICE)
         (GLOBAL (|>MOTH:VICTORIA |>FATH:ALBERT)
                 (PARENTS ALICE VICTORIA ALBERT))
         (BREADTH (GLOBAL (|>X:EDWARD |>MOTH:VICTORIA |>FATH:ALBERT)
                          (PARENTS EDWARD VICTORIA ALBERT))
                  (GLOBAL (|>X:ALICE |>MOTH:VICTORIA |>FATH:ALBERT)
                          (PARENTS ALICE VICTORIA ALBERT))))
  . . .
(BREADTH (GLOBAL ((>X EDWARD)) (FEMALE ALICE)
                               (PARENTS ALICE VICTORIA ALBERT)
                               (PARENTS EDWARD VICTORIA ALBERT))
         (GLOBAL ((>X ALICE))  (FEMALE ALICE)
                               (PARENTS ALICE VICTORIA ALBERT)
                               (PARENTS ALICE VICTORIA ALBERT)))
```

Note that the accidental use of the name X for both the parameter variable ?X in the rule head and the request variable |?X in the question does not lead to a conflict. This is due to FIT's "|" [VERTICAL] prefix distinguishing request variables and sparing it from having to perform PROLOG's above discussed read-time renaming of rule variables.

The computation result is two GLOBAL expressions, used as elements of a BREADTH expression. The second GLOBAL contains two identical instantiations of the 'parents' subgoals of the rule body, showing why the rule regards ALICE as her own sister.

3 FUNCTIONAL AND RELATIONAL PROGRAMMING

While FIT is principally based on a functional programming style, PROLOG is based on a relational one. Therefore a comparison between the two languages entails a comparison of the two programming styles. It is a natural state of affairs that researchers in functional and relational programming have tried to explore their respective

programming disciplines to their ultimate limits. After a period of enthusiastic statements to the effect that 'relations are better than functions', even the most articulate advocate of relational programming now concedes, somewhat cryptically though, that functions can be "more natural" than relations: "Although functional notation is more user-friendly than relational notation, computation by means of rewrite rules is less versatile than backward reasoning." (Kowalski 1983). Actually, there is not only a sense in which relations are 'more general' than functions [cf. section 3.1] but also a sense in which funtions are 'more general' than relations [cf. section 3.2]. However, it now seems clear that both functional and relational programs have characteristic advantages and disadvantages for specific applications. Therefore it seems worthwhile to work toward a natural combination of both methodologies. There are several PROLOG-oriented approaches of function/relation combination, even if we omit indirect function uses in PROLOG that are achieved simply through a LISP interface in the traditional PLANNER-like manner. The diversity of proposals seems to indicate that there is no natural PROLOG solution to function/relation integration:

1. (Eggert & Schorre 1982) require preprocessing which gives rise to the well-known problems of superimposed levels [e.g. error messages from the lower level obstruct the higher level's abstraction effect].

2. (Kornfeld 1983) needs an additional equality theorem for the use of each relation as a function.

3. (Sato & Sakurai 1983) use syntax and semantics that are very hard to understand from their English description.

The FPL language (Bellia et al. 1982) extends a functional language [TEL] by "conditional equations and multi-output functions (described by a relational syntax)" but uses only relations equivalent to deterministic functions [cf. their f example below].

The natural deduction system of (Hansson et al. 1982) defines functions by "equalities or conditional equalities" which seem to interact nicely with the relational constructs [cf. their quick-sort example below].

In FIT we can freely define some algorithms as functions and other ones as relations and then dynamically use the functions as relations and the relations as functions, as desired.

In the first subsection we will show in which sense functions and relations are naturally equivalent and exploit this for their interchangeable FIT use [in section 7 we will exemplify how FIT's adapters can be used for relational programming]. In the second subsection we will treat characteristic functions as another FIT method of representing relations and develop the idea of using function calls with request variables. In the third subsection we will deal with higher-order functions and relations, not available in PROLOG.

3.1 Interchanging Functions and Relations

3.1.1 Mathematical/logical Foundations -

It is well-known that for each N-ary function f [N=0,1,2,...]

f: A1 x A2 x ... x AN -> V

there is an N+1-place relation f-P [we use the suffix "-P" [often abbreviated to "P", as in LISP predicates] to mark relation [or predicate] versions of functions]

f-P $\subseteq$ A1 x A2 x ... x AN x V

such that

f(a1,a2,...,aN) = v

iff

f-P(a1,a2,...,aN,v) holds.

Therefore, given the function f, the relation f-P can be defined thus:

f-P $\subseteq$ A1 x A2 x ... x AN x V

f-P(a1,a2,...,aN,v) := f(a1,a2,...,aN)=v

Since such a relation in PROLOG can be used similarly as a function by using a1, ..., aN as fixed arguments and v as an open argument [which, however, is not really returned as a value], PROLOG relations are often said to be 'more general' than functions.

Conversely, given the relation f-P, the function f and other ones can be defined using Hilbert's epsilon operator (Hilbert & Bernays 1939/70). This is a 'non-deterministic' version of the jota operator, i.e. epsilon(x) P(...,x,...), denotes one of the objects x for which P(...,x,...) holds. If the difference between denoting a value and returning a value is neglected, the epsilon operator can be used to define non-deterministic functions. A non-deterministic function establishes a not necessarily unique correspondence between domain and range elements and thus can still be regarded as a relation mathematically; it becomes function-like through the distinction of domain elements as input arguments and range elements as output values such that for given arguments some value is returned.

For the definition of the deterministic function f from the relation f-P the epsilon operator happens to act deterministically only:

f: A1 x A2 x ... x AN -> V

f(a1,...,aN) := epsilon(v) f-P(a1,...,aN,v)

However, the non-deterministic capability of the epsilon operator is necessary for the definition of N further [in general non-deterministic] functions fI [1$\leq$I$\leq$N] from the f-P relation:

fI: A1 x ... x AI-1 x AI+1 x ... x AN x V -> powerset(AI)

fI(a1,...,aI-1,aI+1,...,aN,v) :=
epsilon(aI) f-P(a1,...,aI-1,aI,aI+1,...,aN,v)

In the above definitions, the powerset range is used to explain a non-deterministic function as a mapping into a set of subsets [in pure FIT, of BREADTH expressions], rather than into a set of single elements; to accomodate for the special case of a non-deterministic function that returns exactly one value for some arguments, we identify singleton sets with their single elements [in pure FIT, (BREADTH e) = e]; the empty set represents failure [in pure FIT, (BREADTH) = jU]. As in epsilon expressions, in impure FIT one element of such a subset is selected on return from a call fI(a1,...,aI-1,aI+1,...,aN,v); unlike in epsilon expressions, in impure FIT there is the possibility of successive attempts at return from that function call, which enumerate the remaining subset elements.

In general, if we put M=N+1 [i.e. M=1,2,3...], an M-place relation can be used to define M (M-1)-ary functions. Some or all of these functions may be non-deterministic.

3.1.2 FIT Definitions -

In FIT the definition of a relation from a function is made by a trivial EQUAL call that implements "=".

The definition of functions from a relation is made by LOCAL expressions that implement an epsilon operator which really returns values rather than just denoting them [that is, other than QUTE (Sato & Sakurai 1983), FIT doesn't require epsilon expressions as a language construct but represents them by the much more versatile LOCAL expressions]. Such LOCALs contain a relation call with one open variable v [marked by a |?-prefix] and M-1 fixed places before the colon and contain the variable v [marked by a <-prefix] after the colon. Thus

(LOCAL (r p1 ... pI-1 |?v pI+1 ... pM) : <v)

implements

epsilon(v) r(p1,...,pI-1,v,pI+1,...,pM) for $1 \leq I \leq M$

In PROLOG neither definition is possible because of its lack of value-returning functions.

Now let us consider an example. For the binary function NTH, returning the Nth element X of a list L, there is the ternary relation NTH-P or NTHP, holding for triples (N,L,X) iff X occurs as the Nth L-element. In FIT, if the function NTH is defined by something like

(>(NTH 1 (?X #ID)) <X); NTH of N=1 and L=(elem ...) is elem
r(NTH SUB1 CDR); NTH of other N and L is NTH of SUB1 of N and CDR of L

then the relation NTHP can be defined by

```
(>(NTHP ?N ?L ?X) (EQUAL (NTH <N <L) <X))
```

Conversely, if the relation NTHP is defined by something like

```
(NTHP 1 (?X #ID) ?X); NTHP of N=1, L=(elem ...), and X=elem is true
r(NTHP SUB1 CDR ID); NTHP of other N, L, and X is (NTHP SUB1:N CDR:L X)
```

then the function NTH can be defined by

```
(>(NTH ?N ?L) (LOCAL (NTHP <N <L |?X) : <X))
```

The same relation NTHP can also be used to define two more binary functions [the following NTHP derivates, other than NTH, don't run in FIT-1, because general function-variable unification fitting is not implemented in this first version of FIT; cf. subsection 5.3]:

POSITION returns the numeric position N of a given element X in a list L.

```
(>(POSITION ?X ?L) (LOCAL (NTHP |?N <L <X) : <N))
```

NXLISTS returns the lists L which have the element X in position N.

```
(>(NXLISTS ?N ?X) (LOCAL (NTHP <N |?L <X) : <L))
```

Of these NTHP derivates, the function NTH is deterministic while the functions POSITION and NXLISTS are non-deterministic. POSITION [returning the position of an arbitrary occurrence of a given element in a given list] is finitely non-deterministic, while NXLISTS [returning an arbitrary list with a given element in a given position] is infinitely non-deterministic.

3.1.3 Several Request Variables -

The above representation of epsilon expressions by LOCAL expressions only makes use of a very special kind of LOCAL, whose left-imposition part is a relation call containing one request variable |?v and whose body is a single PULL variable <v.

A more general kind of LOCAL, whose left-imposition part is a relation call containing several [S] request variables |?v1, ..., |?vS and whose body consists of some permutation of corresponding PULL variables <v1, ..., <vS, can be used to define multi[S]-valued functions by abstracting S arguments from an M-place relation [S≤M]. It has the form

```
(LOCAL (r e1 ... eM) : <v1 ... <vS)
```

where eI is either a request variable |?vjI [1≤I≤M, 1≤jI≤S] or a fixed place pI and for each <vK there is exactly one eI = |?vK [1≤K≤S]. This LOCAL expression form corresponds to a generalized epsilon expression form with S epsilon variables

```
epsilon(v1,...,vS) r(e1,...,eM)
```

For example, the NTHP relation can also be used to define a 2-valued

function POSELEM which non-deterministically returns some position N together with the element X in it of a given list L.

```
(>(POSELEM ?L) (LOCAL (NTHP |?N <L |?X) : <N <X))
```

LOCALs whose bodies consist of a "∂" [INSTANTIATE] expression over the variables v1, ..., vS can be used to obtain the analogue to "answer" templates in some logic programming languages, e.g. in LOGLISP (Robinson & Sibert 1981) and micro-PROLOG (Clark et al. 1982) [remember, however, that in FIT such 'answers' are true function values, nestable inside arbitrary other function applications in the ordinary functional manner, whereas in PROLOG dialects these normally are just top-level print outs]. For example, POSELEMLST is a variant of POSELEM which returns one list of the form (THE POSITION IS N AND THE ELEMENT IS X) instead of two values N and X.

```
(>(POSELEMLST ?L)
 (LOCAL (NTHP |?N <L |?X)
        :
        ∂(THE POSITION IS <N AND THE ELEMENT IS <X)))
```

Almost-general LOCALs whose bodies consist of arbitrary expressions over the variables v1, ..., vS can be used for defining functions that perform arbitrary post-processing on the variables abstracted from a relation. For example FIXEDELEM is another variation of POSELEM which returns T iff the list L contains a fixed-point-like 'fixed element', i.e. a positive integer element that is equal to the numeric position in which it occurs in L [the "u" operator transforms jF to jU, which is necessary for discarding those non-deterministic possibilities for which EQ yields jF].

```
(>(FIXEDELEM ?L) (LOCAL (NTHP |?N <L |?X) : u(EQ <N <X)))
```

For example, (FIXEDELEM '(2 1 3 5 1)) non-deterministically yields the LOCAL position-element bindings {N=1, X=2}, {N=2, X=1}, {N=3, X=3}, {N=4, X=5}, {N=5, X=1}, one of which [characterizing 3 as a fixed element] makes the EQ call return T.

We regard the quick-sort definition in (Hansson et al. 1982) as another example for this generalization of epsilon expressions:

```
quick-sort(NIL)=NIL
quick-sort(x.y)=conc(quick-sort(y1),x.quick-sort(y2)) <--
                                        partition(x,y,y1,y2)
```

In FIT this becomes

```
(>(QUICK-SORT NIL) NIL)
(>(QUICK-SORT (?X ?YoLIST))
 (LOCAL (PARTITION <X <Y |?Y1 |?Y2)
        :
        (APPEND (QUICK-SORT <Y1) (CONS <X (QUICK-SORT <Y2)))))
```

Since LOCAL expressions, in addition to their generalized epsilon expression use, can play the role of "LET expressions" (Landin 1965), generalized for localizing names of request variables whose values are to be reused several times, the f "equation" in (Bellia et al. 1982),

```
f(x)=y <-- r(in:x;out:w1,z),h1(z)=w2,h2(z)=w3,g(w1,w2,w3)=y
```

can be formulated in FIT as [input variables in FIT are <-prefixed, output variables are |?-prefixed]

```
(>(F ?X) (LOCAL (R <X |?W1 |?Z) : (G <W1 (H1 <Z) (H2 <Z))))
```

and their NPL-style version

```
f(x)=g(w1,h1(z),h2(z)) where r'(x)=<w1,z>
```

can be expressed in FIT as

```
(>(F ?X) (LOCAL (?W1 ?Z : (R' <X)) : (G <W1 (H1 <Z) (H2 <Z))))
```

Note that no additional tuple notion [corresponding to <...>] is needed in FIT because R' returns an imposition which can be distributed among W1 and Z by a ":"-fitting [cf. section 5.1]. Thus the rationale for using a relational syntax given in (Bellia et al. 1982) would not apply to FIT.

Completely general LOCALs have an additional colon, separating the previously used 'then-part' from a new 'else-part'. This permits taking arbitrary action for relation calls that yield no bindings but a jF. For example, the above POSELEMLST definition can be modified to a final variant POSELSE, which returns 2-list-impositions of the form (N POSITION) (X ELEMENT), containing positions N and elements X of non-empty lists, and the 2-list-imposition (NO POSITION) (NO ELEMENT) for the empty list.

```
(>(POSELSE ?L)
 (LOCAL (NTHP |?N <L |?X)
        :
        (LIST <N POSITION)
        (LIST <X ELEMENT)
        :
        '(NO POSITION)
        '(NO ELEMENT)))
```

Then the call, say, (POSELSE '(A B C)) non-deterministically returns (1 POSITION) (A ELEMENT), (2 POSITION) (B ELEMENT), or (3 POSITION) (C ELEMENT), whereas, assuming (>(NTHP ?N NIL ?X) jF) is asserted as well, the call (POSELSE NIL) returns (NO POSITION) (NO ELEMENT).

3.1.4 The ESCVAL Operator -

As a notational convenience we could introduce another prefix operator "$" [ESCVAL], meaning "escape with value", which distinguishes a request variable such that the value it receives replaces the normal value of the entire request expression in which it occurs directly [this should not be confused with the above mentioned "output" variables as used in (Bellia et al. 1982) which are to be bound through relation calls]. More precisely, an ESCVAL expression of the form

```
(r p1 ... pI-1 $|?v pI+1 ... pM)
```

could be defined by our previous LOCAL expression

```
(LOCAL (r p1 ... pI-1 |?v pI+1 ... pM) : <v)
```

If, furthermore, ESCVAL or LOCAL expressions embedded in requests are evaluated 'by value', like FIT's but unlike PROLOG's embedded terms, this would allow the nesting of relation-like expressions, similar to the nesting of function calls. For example, the LISP/FIT function nesting

```
(PLUS (SQUARE 5) (SQUARE 3))
```

with ESCVAL would become

```
(PLUSP (SQUAREP 5 $|?S1) (SQUAREP 3 $|?S2) $|?P)
```

which is more concise than the equivalent LOCAL form

```
(LOCAL (PLUSP (LOCAL (SQUAREP 5 |?S1) : <S1)
              (LOCAL (SQUAREP 3 |?S2) : <S2)
              |?P)
       :
       <P)
```

The PROLOG conjunction corresponding to the function nesting, namely

```
?- squarep(5,S1), squarep(3,S2), plusp(S1,S2,P).
```

with ESCVAL would become

```
?- plusp(squarep(5,$S1),squarep(3,$S2),$P).
```

ESCVAL expressions can be regarded as generalizing both functions and relations because they return values, like functions, and are not based on a distinction of input and output arguments, like relations. A larger example of ESCVAL uses will be treated in section 8.2.

A more general ESCVAL operator might prefix an arbitrary expression such that its value replaces that of the directly superordinate expression; this would allow escaping values of request variables occurring indirectly in request expressions by prefixing all intermediate expressions with "$". More precisely, a general ESCVAL expression of the form

```
(r p1
   ...
   pI-1
   $(s q1 ... qJ-1 $(... $|?v ...) qJ+1 ... qN)
   pI+1
   ...
   pM)
```

could be defined by the LOCAL expression

```
(LOCAL (r p1
          ...
          pI-1
          (s q1 ... qJ-1 (... |?v ...) qJ+1 ... qN)
          pI+1
          ...
          pM)
       :
       <v)
```

3.2 Function Calls with Request Variables

3.2.1 Interpreting Relations as Characteristic Functions -

An M-place relation also defines an additional M-ary function, namely its characteristic function. Since, in FIT, relation calls return values, this functional view of relations is implicit in that language; for example, the FIT relation call [predicate function call]

```
(NTHP 3 '(A B C D) C)
```

returns the 'true' value (NTHP 1 (C D) C). In general, a 'false'-failing relation call in FIT yields jF, an 'unknown'-failing one yields jU, and a successful one returns T or any other value unequal to jF or jU. In PROLOG, relation calls don't return values; for example, the PROLOG relation call

```
?- nthp(3,[a,b,c,d],c).
```

prints 'yes' but doesn't return anything.

Suppose some FIT programmer doesn't want compound truth-values like (NTHP 1 (C D) C) as used for question-overanswering, which here can be interpreted as "Yes [C is the 3rd element of (A B C D)] because C is the 1st element of (C D)". This programmer may obtain the atomic truth-value T by rewriting facts which are adapters of the form

```
(r p1 ... pM)
```

as transformers of the form

```
(>(r p1 ... pM) T)
```

For example, the NTHP fact

```
(NTHP 1 (?X #ID) ?X)
```

can be rewritten as

```
(>(NTHP 1 (?X #ID) ?X) T)
```

Now the call

```
(NTHP 3 '(A B C D) C)
```

would return the atom T. Similarly, the usual numeric characteristic functions could be modelled directly be defining facts as transformers that return 1 instead of T [and 0 instead of jF].

The important observation is that all these relation-derived functions in FIT can still be used with request variables, so that, besides returning values, they also yield variable binding effects. For example, the call

```
(NTHP 3 '(A B C D) |?X)
```

would return some 'true' value, according to the already discussed alternatives used to define the NTHP fact, and it would also bind X to C.

3.2.2 Extending Relations to General Functions -

While the values returned by characteristic functions are primarily truth-values, nothing changes if we use arbitrary values. That is, in FIT not only predicate-like functions but also arbitrary general functions can be used with request variables.

To demonstrate this, we can start with another predicate function KNOWS, defined as a transformer fact for JOHN and MARY:

```
(>(KNOWS JOHN MARY) 1)
```

If we extend the two-valued characteristic function to a many-valued probabilistic, possibilistic, or fuzzy function, we can modifiy the previous fact to something like

```
(>(KNOWS JOHN MARY) .824)
```

Now a request like (KNOWS JOHN |?X) returns .824 and binds X to MARY. Instead of numeric certainty degrees we can also use symbolic ones as in

```
(>(KNOWS JOHN MARY) QUITE-WELL)
```

Now a request like (KNOWS JOHN |?X) returns QUITE-WELL and binds X to MARY. Symbolic values can not only represent degraded truth but also many other things, such as the person believing the fact as in

```
(>(KNOWS JOHN MARY) (OPINIONOF PAUL))
```

Now a request like (KNOWS JOHN |?X) returns (OPINIONOF PAUL) and binds X to MARY.

If the original fact contains variables [understood to be quantified universally], then the value may be an expression in these variables. For example, the fact

```
(NEEDS ?EVERYBODY A-PRODUCT)
```

cannot only be extended to a function with a constant value, like

(>(NEEDS ?EVERYBODY A-PRODUCT) IN THE EYES OF A-COMPANY)

but also to a function with a variable value, like

(>(NEEDS ?EVERYBODY A-PRODUCT) ESPECIALLY <EVERYBODY WILL NEED IT)

With the latter definition a request like (NEEDS JOHN |?X) returns ESPECIALLY JOHN WILL NEED IT and binds X to A-PRODUCT.

In general for each relation definition

(r p1 .... pM)

and arbitrary value v, which may depend on p1, ..., pM, there is a function definition

(r p1 .... pM) := v

or, in FIT,

(>(r p1 ... pM) v)

That is, each relation can be extended to a function. Although in PROLOG relation definitions of the above form correspond to facts only, rule-defined relations must also be 'grounded' in facts, to which, then, the function-generalization is applicable. The values of the ground facts can be calculated and passed back across the rule arrows. For this, a PROLOG rule definition [in FIT syntax]

(>(r p1 ... pM) (r1 ...) ... (rZ ...))

can be replaced by

(>(r p1 ... pM) (combine (r1 ...) ... (rZ ...)))

where combine is some function combining the values returned by the conjuncts of the rule body. For numeric values combine=TIMES or combine=MIN may be applied, as usual; for symbolic values there are many combination possibilities, the most trivial being combine=LIST [combine must fulfill the requirement of strictness, so that jF and jU signals can escape from its calls].

Since such a function in FIT can be used as a relation [i.e. with arbitrary fixed and open arguments] by simply ignoring the value returned as long as it is 'true', FIT functions can be said to be 'more general' than relations.

Of course, the M-ary function (>(r p1 ... pM) v), derived from the M-ary relation (r p1 ... pM), can again be represented as an M+1-ary relation (r p1 ... pM v); for example, (>(KNOWS JOHN MARY) (OPINIONOF PAUL)) thus becomes (KNOWS JOHN MARY (OPINIONOF PAUL)). And of course, the M+1-ary relation could again be generalized to an M+1-ary function (>(r p1 ... pM v) v'), e.g., (KNOWS JOHN MARY (OPINIONOF PAUL)) to (>(KNOWS JOHN MARY (OPINIONOF PAUL)) QUITE-WELL), etc. ad infinitum. However, this misses the point that relation calls don't exploit the linguistic dimension of returning useful values although nothing would prevent them from doing so.

Finally note that relations generalized to functions by means of extending definitions, can still be used as functions by means of the ESCVAL operator [subsection 3.1.4], because this use is defined by the LOCAL semantics of ESCVAL: After the function extension (>(r p1 ... pM) v) the ESCVAL expression (r p1 ... pI-1 $|?w pI+1 ... pM) expands to (LOCAL (r p1 ... pI-1 |?w pI+1 ... pM) : <w), hence throws away the value v returned by (r p1 ... pI-1 |?w pI+1 ... pM) and instead returns the value of w. For example, after (>(KNOWS JOHN MARY) QUITE-WELL) the request (KNOWS $|?W MARY) via (LOCAL (KNOWS |?W MARY) : <W), (LOCAL (GLOBAL ((>W JOHN)) QUITE-WELL) : <W), and (LOCAL ((>W JOHN)) <W) returns JOHN. In other words, the programmer need not be aware of what kind of 'true' value an expression would return if used without an ESCVAL operator; if used with ESCVAL, it always returns the ESCVAL-marked subexpression.

3.2.3 Using Functions like Relations -

We now proceed to three examples of functions not derived from relations and still usable with request variables, like relations.

A unary function FATHEROF can be defined by a set of individual settings such as

```
(>(FATHEROF ATHENA) ZEUS)
(>(FATHEROF APOLLO) ZEUS)
(>(FATHEROF ZEUS) CRONUS)
```

This FATHEROF function can not only be called with a fixed argument as in (FATHEROF ATHENA) returning ZEUS, but also with a request variable argument as in (FATHEROF |?X) non-deterministically returning ZEUS and binding X to ATHENA, returning ZEUS and binding X to APOLLO, or returning CRONUS and binding X to ZEUS. In general, function calls all of whose arguments are request variables enumerate the function's range and bind the variables to the corresponding domain elements. In the FATHEROF example the call returns all persons known to be fathers and binds the single request variable argument to their children.

A binary function PARENTS can be defined similarly as

```
(>(PARENTS ABRAHAM HAGAR) ISHMAEL)
(>(PARENTS ABRAHAM SARAH) ISAAC)
```

[The persons of this definition were quoted previously to illustrate various things; in the PROLOG literature the males occur in (Clocksin & Mellish 1981) and the females were added in (Fuhlrott 1982).]

This PARENTS function can be called as follows. The child of ABRAHAM and SARAH is obtained when both arguments are correspondingly fixed as in (PARENTS ABRAHAM SARAH), returning ISAAC. All children of ABRAHAM and any woman are obtained when the first argument is fixed to ABRAHAM and the second is left open as in (PARENTS ABRAHAM |?X), non-deterministically returning ISAAC and binding X to SARAH or returning ISHMAEL and binding X to HAGAR. All children of HAGAR and any man are obtained when the second argument is fixed to HAGAR and the first is left open as in (PARENTS |?X HAGAR), returning ISHMAEL and binding X to ABRAHAM. In general, function calls some of whose

arguments are request variables and the other ones are fixed enumerate the function's subrange under the fixed arguments and bind the variables to the remaining domain elements. All children of any man and woman are obtained when both arguments are left open as in (PARENTS |?X |?Y), returning ISAAC and binding X to ABRAHAM and Y to SARAH or returning ISHMAEL and binding X to ABRAHAM and Y to HAGAR.

In general, of course, the PARENTS function is non-deterministic even for fixed fathers and mothers. This can be expressed using the "v" prefix [cf. section 6.2] as in

```
(v(PARENTS ISAAC REBECCA) ESAU)
(v(PARENTS ISAAC REBECCA) JACOB)
```

Now even with both arguments fixed as in (PARENTS ISAAC REBECCA) we non-deterministically obtain ESAU or JACOB. With the first argument fixed to ISAAC and the second left open as in (PARENTS ISAAC |?X) we also obtain ESAU or JACOB, in both cases X becoming bound to REBECCA. Requests like (PARENTS |?X REBECCA) and (PARENTS |?X |?Y) behave similarly.

A recursive function HONOI for initializing homogeneously colored towers of Hanoi of given hight by constructing them as impositions of the disks available in the data base can be defined as

```
(>(HONOI ?DIAMETER ?COLOR)
 (DISK <DIAMETER <COLOR)
 (HONOI (SUB1 <DIAMETER) <COLOR))
(>(HONOI 0 ?COLOR))
(DISK 1 RED)
(DISK 3 RED)
(DISK 2 RED)
(DISK 1 BLUE)
(DISK 3 BLUE)
(DISK 2 BLUE)
```

This function can be called with fixed color argument as in (HONOI 3 RED), returning the imposition

```
(DISK 3 RED) (DISK 2 RED) (DISK 1 RED)
```

or with an open color argument as in (HONOI 3 |?X), returning the impositions

```
(DISK 3 BLUE) (DISK 2 BLUE) (DISK 1 BLUE)
```

or

```
(DISK 3 RED) (DISK 2 RED) (DISK 1 RED)
```

The second HONOI call chooses a single color for all disks of a tower, here blue or red. Nonhomogeneously colored towers cannot be constructed because all occurrences of the color request variable |?X must be bound consistently. Although in both cases we called the HONOI function with fixed diameter arguments, it can also be called with an open diameter argument, but then in FIT-1 only the diameter 0 is chosen and the empty tower is constructed.

3.3 Higher-order Functions and Relations

3.3.1 The Direct Approach -

Let us begin with expressions consisting of several functions. A nested function call, in the simplest case having the form

g(h(a))

for PROLOG must first be rewritten as a conjunction of two function calls communicating via a temporary variable x,

x=h(a), y=g(x)

which can then be put into a relational form

h-P(a,x), g-P(x,y)

This leads to a flat system of relation calls with many temporary variables whose correspondence is often difficult to establish. On the other hand, FIT's LISP-like nesting form

(g (h a))

leads to deeply nested function calls with many closing parentheses. To avoid unnecessary parentheses in FIT a right-associative 'apply' infix operator ":" can be used for unary functions, simplyfing the above nesting to

g:h:a

PROLOG's need for temporary "object variables" makes it impossible to use higher-order functions or "function-level operators" in the sense of (Backus 1982) in that language. An example of what cannot be expressed in PROLOG is a function composition like

goh

which in Backus' FP, in FIT, and in other functional languages can be passed as an argument and returned as a value, independently of the arguments to which it will be eventually applied. Only on application to an argument a can the composition goh be expanded to the nesting g(h(a)), and only then could the above rewriting to the corresponding relational PROLOG form begin.

More generally, PROLOG's restriction to first-order predicate calculus prevents operations on relations, i.e. it forces everything that is said to be said about individuals [Backus' objects]. Higher-order functions or predicates [relations] are not available. Thus a very useful dimension of abstraction is not exploited in PROLOG.

For example, in FIT we can form the composition of the successor function g=SUCC and the square function h=SQUARE, SUCCoSQUARE, where "o" is an infix shorthand for the higher-order COMPOSE function, which becomes explicit in the unshortened notation (COMPOSE SUCC SQUARE). Higher-order functions can be defined in FIT like first-order functions. For example, although COMPOSE is built into FIT-1, it could be defined in FIT-1 itself by

3.3 Higher-order Functions and Relations

3.3.1 The Direct Approach -

Let us begin with expressions consisting of several functions. A nested function call, in the simplest case having the form

```
g(h(a))
```

for PROLOG must first be rewritten as a conjunction of two function calls communicating via a temporary variable x,

```
x=h(a), y=g(x)
```

which can then be put into a relational form

```
h-P(a,x), g-P(x,y)
```

This leads to a flat system of relation calls with many temporary variables whose correspondence is often difficult to establish. On the other hand, FIT's LISP-like nesting form

```
(g (h a))
```

leads to deeply nested function calls with many closing parentheses. To avoid unnecessary parentheses in FIT a right-associative 'apply' infix operator ":" can be used for unary functions, simplyfing the above nesting to

```
g:h:a
```

PROLOG's need for temporary "object variables" makes it impossible to use higher-order functions or "function-level operators" in the sense of (Backus 1982) in that language. An example of what cannot be expressed in PROLOG is a function composition like

```
goh
```

which in Backus' FP, in FIT, and in other functional languages can be passed as an argument and returned as a value, independently of the arguments to which it will be eventually applied. Only on application to an argument a can the composition goh be expanded to the nesting g(h(a)), and only then could the above rewriting to the corresponding relational PROLOG form begin.

More generally, PROLOG's restriction to first-order predicate calculus prevents operations on relations, i.e. it forces everything that is said to be said about individuals [Backus' objects]. Higher-order functions or predicates [relations] are not available. Thus a very useful dimension of abstraction is not exploited in PROLOG.

For example, in FIT we can form the composition of the successor function g=SUCC and the square function h=SQUARE, SUCCoSQUARE, where "o" is an infix shorthand for the higher-order COMPOSE function, which becomes explicit in the unshortened notation (COMPOSE SUCC SQUARE). Higher-order functions can be defined in FIT like first-order functions. For example, although COMPOSE is built into FIT-1, it could be defined in FIT-1 itself by

3.3 Higher-order Functions and Relations

3.3.1 The Direct Approach -

Let us begin with expressions consisting of several functions. A nested function call, in the simplest case having the form

```
g(h(a))
```

for PROLOG must first be rewritten as a conjunction of two function calls communicating via a temporary variable x,

```
x=h(a), y=g(x)
```

which can then be put into a relational form

```
h-P(a,x), g-P(x,y)
```

This leads to a flat system of relation calls with many temporary variables whose correspondence is often difficult to establish. On the other hand, FIT's LISP-like nesting form

```
(g (h a))
```

leads to deeply nested function calls with many closing parentheses. To avoid unnecessary parentheses in FIT a right-associative 'apply' infix operator ":" can be used for unary functions, simplyfing the above nesting to

```
g:h:a
```

PROLOG's need for temporary "object variables" makes it impossible to use higher-order functions or "function-level operators" in the sense of (Backus 1982) in that language. An example of what cannot be expressed in PROLOG is a function composition like

```
goh
```

which in Backus' FP, in FIT, and in other functional languages can be passed as an argument and returned as a value, independently of the arguments to which it will be eventually applied. Only on application to an argument a can the composition goh be expanded to the nesting g(h(a)), and only then could the above rewriting to the corresponding relational PROLOG form begin.

More generally, PROLOG's restriction to first-order predicate calculus prevents operations on relations, i.e. it forces everything that is said to be said about individuals [Backus' objects]. Higher-order functions or predicates [relations] are not available. Thus a very useful dimension of abstraction is not exploited in PROLOG.

For example, in FIT we can form the composition of the successor function g=SUCC and the square function h=SQUARE, SUCCoSQUARE, where "o" is an infix shorthand for the higher-order COMPOSE function, which becomes explicit in the unshortened notation (COMPOSE SUCC SQUARE). Higher-order functions can be defined in FIT like first-order functions. For example, although COMPOSE is built into FIT-1, it could be defined in FIT-1 itself by

```
(>((COMPOSE ?G ?H) >X) (<G (<H <X)))
```

A composition can then be used exactly like an ordinary function, say with the argument 3 as in

```
((COMPOSE SUCC SQUARE) 3)
```

which returns 10. It can also be used as the paramenter of the higher-order "#" [REPEAT] function, which applies its parameter to its arbitrary number of arguments, obtaining (REPEAT (COMPOSE SUCC SQUARE)) or #(COMPOSE SUCC SQUARE). The object #(COMPOSE SUCC SQUARE) can again be used as an ordinary function, for instance with the four arguments 0, 1, 2, and 3 as in

```
(#(COMPOSE SUCC SQUARE) 0 1 2 3)
```

returning 1 2 5 10.

3.3.2 Warren's Simulation Method -

As discussed above, such compositions cannot be expressed in PROLOG as it stands. Nor is it possible to define a higher-order compose predicate, analogous to FIT's higher-order COMPOSE function definition, by something like

```
compose(G,H)(X,Z) :- H(X,Y), G(Y,Z).
```

which could then be invoked by

```
?- compose(succp,squarep)(3,Ans).
```

The only way out is to implement compositions as part of a new language on top of PROLOG. In other words, goh must be used as a data object, namely as a term compose(g,h). For instance, (Warren 1982) introduces a predicate apply, which really is an interpreter of a language of 'term programs' like compose(g,h). One defining clause of the interpreter apply may specify what to do with such compose structures:

```
apply(compose(G,H),X,Z) :- apply(H,X,Y), apply(G,Y,Z).
```

But now we must also specify apply clauses for every function g and h that is to be used in the composition-extended PROLOG; e.g. for succp and squarep we have to provide the apply definitions

```
apply(succp,X1,X2) :- succp(X1,X2).
apply(squarep,X1,X2) :- squarep(X1,X2).
```

along with the ordinary succp and squarep definitions

```
succp(X,Y) :- Y is X+1.
squarep(X,Y) :- Y is X*X.
```

After these preparations we can use apply for compose(succp,squarep) and the argument 3 as in

```
?- apply(compose(succp,squarep),3,Ans).
```

which binds Ans to 10. However we get an error, if we try the analogue of repeating the compose term over several arguments, by mapping it over a list of these arguments with the maplist relation for predicates described in (Clocksin & Mellish 1981). We must again define a special apply version for mapping, which we call 'mapplylist':

```
mapplylist(_,[],[]).
mapplylist(P,[X|L],[Y|M]) :-
   apply(P,X,Y),
   mapplylist(P,L,M).
```

Using this we can eventually simulate what we want:

```
?- mapplylist(compose(succp,squarep),[0,1,2,3],Ans).
```

binds Ans to the list [1,2,5,10].

We don't regard this simulation of higher-order functions in PROLOG as a proper extension of that language because it doesn't permit the direct use of the original language kernel [e.g. succp, squarep, maplist] from the newly defined constructs. Warren is really beginning to define a new interpreter when he introduces apply definitions, although he doesn't seem to notice this status of apply. He even argues that the 'extension' can be regarded as "syntactic sugar" for standard first-order logic; this use of the term "syntactic sugar" has completely lost the original meaning of (Landin 1965), whose LET extension does leave the underlying LAMBDA kernel language untouched. In (Warren 1982) it is stated that for the higher-to-first-order reduction a clause

"apply(foo,X1,...,Xn) :- foo(X1,...,Xn).

is supplied for each predicate foo which needs to be treated as a data object", and we have done this for succp and squarep, but actually this means that one needs an additional clause for all predicates one ever wants to use as arguments of higher-order predicates. However, when you define a predicate like succp or squarep you normally don't know whether you or other programmers will need it at some later point in a higher-order construct like compose, twice, or whatever. After some errors caused by missing apply clauses you will certainly contemplate a convention for generally supplying predicates with the additional apply clause. However, since many of these clauses would never be used, the resulting increase of code would be unacceptable because it makes programs less readable and more storage consumptive. It was therefore proposed by (Nebel 1983) to define apply using PROLOG's "=.." and "call" predicates, which abbreviates Warren's clauses for, say, n=2 to the single general clause

```
apply(Foo,X1,X2) :- Q=..[Foo,X1,X2], call(Q).
```

that could be further generalized by always using, like LISP's APPLY, one argument list X instead of some fixed number n of arguments X1, ..., Xn. Although this definition is very concise, it does not only depend on the extra-logical "=.." and "call" features but must also be positionned judiciously, namely after all other, specific apply clauses. This, in turn, entails that all these specific clauses must be augmented by a cut operator to prevent calls like apply(compose(succp,sqrtp),9,Ans) from falling into the last, catch-all

apply definition if their body fails [say, because sqrtp is undefined]. In our example this leads to

```
apply(compose(G,H),X,Z) :- !, apply(H,X,Y), apply(G,Y,Z).
apply(twice(G),X,Z) :- !, apply(G,X,Y), apply(G,Y,Z).
      ...
apply(Foo,X1,X2) :- Q=..[Foo,X1,X2], call(Q).
```

One might therefore start to consider building apply clauses implicitly into the PROLOG interpreter, thus taking the first step toward really extending PROLOG for higher-order constructs. Warren's simulation method may be theoretically nice, but it isn't practical.

Kowalski, unlike Warren, has recently acknowledged that higher-order functions are a serious problem for PROLOG-like languages; however, his attempt to use a logical metalanguage for simulating higher-order functions is still quite "complicated" (Kowalski 1983), and looks even less practical than Warren's simulation.

3.3.3 New Higher-order Functions from Old -

Noticing the relationship between the above compose and twice definitions, we may, in addition, ask if the really nice features of functional programming, like the definition of higher-order functions [e.g. TWICE] by other higher-order functions [e.g. COMPOSE], as opposed to their above "object-level" (Backus 1982) definitions, can in principle be expressed nearly as nicely in relational programming. For example, the TWICE-by-COMPOSE definition in FIT can be formulated very concisely with

```
(>(TWICE ?G) (COMPOSE <G <G))
```

which may be called on the top-level, as in

```
(TWICE ADD1)
(COMPOSE ADD1 ADD1)
```

returning a higher-order function, or in a functional position, as in

```
((TWICE ADD1) 0)
  ((COMPOSE ADD1 ADD1) 0)
  (ADD1 (ADD1 0))
2
```

applying the higher-order function and returning a data object. Concerning PROLOG, even if a definition

```
apply(twice(G),X,Z) :- apply(compose(G,G),X,Z).
```

in ordinary PROLOG, can be shortened to

```
twice(G)(X,Z) :- compose(G,G)(X,Z).
```

in an extended PROLOG, the redundant object variables X and Z cannot be omitted, i.e. the definition cannot be shortened to something like

```
twice(G) :- compose(G,G).
```

without introducing functions as a true counterpart to relations.

To see the relevance of the above discussion for day-to-day relational programming, consider the PROLOG clauses

```
grandfatherofp(X,Z) :- parentofp(X,Y), fatherofp(Y,Z).
uncleofp(X,Z) :- parentofp(X,Y), brotherofp(Y,Z).
   ...
```

where "..." stands for analogous rules for grandmotherofp, auntofp etc. Such relations could be redefined on a higher level of abstraction in a most concise manner as

```
grandfatherofp :- compose(fatherofp,parentofp).
uncleofp :- compose(brotherofp,parentofp).
   ...
```

without requiring all these object variables X, Y, and Z but instead using the previously discussed higher-order compose predicate. While this is only a suggestion for an extended PROLOG, the corresponding functional definitions

```
(>GRANDFATHEROF  (COMPOSE FATHEROF PARENTOF))
(>UNCLEOF (COMPOSE BROTHEROF PARENTOF))
   ...
```

are a reality in FIT-1.

4 PROLOG STRUCTURES AND FIT COMPOUNDS

As an alternative to LISP lists, PROLOG uses so-called 'structures', also called 'compound terms'. A structure consists of a functor f of arity N and arguments a1, a2, ..., aN; the arguments may again be structures. It is written in the usual mathematical/logical prefix notation

```
f(a1,a2,...,aN)
```

FIT's generalization of LISP lists are 'compounds', but only their list-like subset is considered explicitly here and later on. The above PROLOG structure in FIT can be represented by a compound of length N+1 with a distinguished first element f~ and remaining elements a1~, a2~, ..., aN~. It is written in LISP's Cambridge Polish prefix notation

```
(f~ a1~ a2~ ... aN~)
```

where f~ is a FIT atom corresponding to the PROLOG-functor f and a1~, a2~, ..., aN~ are recursively rewritten subexpressions corresponding to a1, a2, ..., aN, respectively, down to the ground-level of PROLOG constants which are rewritten to FIT constants by integer~=integer and atom~=ATOM.

As an example let us consider the notation of LISP's dotted pairs as PROLOG structures and a corresponding FIT representation. Such a structure uses a functor f="." of arity N=2 and two arguments, say, a1=alfa and a2=beta, hence it may look like

.(alfa,beta)

The corresponding compound of length N+1=3 uses the distinguished atom f~=DOT and arguments a1~=ALFA and a2~=BETA, i.e. it is

(DOT ALFA BETA)

Similarly a PROLOG structure nesting like

.(alfa,.(beta,.(gamma,nil)))

becomes the FIT compound nesting

(DOT ALFA (DOT BETA (DOT GAMMA NIL)))

PROLOG structures have an important restriction as compared to LISP lists and FIT compounds, namely their fixed arity. Besides the binary "."-functor PROLOG could use a triple functor allowing structures like triple(alfa,beta,gamma), a quadruple functor allowing structures like quadruple(alfa,beta,gamma,delta) etc. but not a general tuple functor allowing all these structures tuple(alfa,beta), tuple(alfa,beta,gamma), tuple(alfa,beta,gamma,delta) etc. FIT, on the other hand, besides DOT compounds not only allows TRIPLE compounds like (TRIPLE ALFA BETA GAMMA), QUADRUPLE compounds like (QUADRUPLE ALFA BETA GAMMA DELTA) etc. but also general TUPLE compounds like (TUPLE ALFA BETA), (TUPLE ALFA BETA GAMMA), (TUPLE ALFA BETA GAMMA DELTA) etc.

A PROLOG functor f has either a single fixed arity N or it is 'overloaded' by a, usually small, finite number k of fixed arities N1, ..., Nk. Occurrences of an arity-overloaded functor f are sometimes written along with their arities N1, ..., Nk as f/N1, ..., f/Nk, which can also be regarded as k different functors, each with its own fixed arity. Lists and compounds, on the other hand, can be used with a distinguished first element followed by a varying, potentially infinite number of arguments, with available computer memory being the only restriction on the maximum argument number. For example, sets whose cardinality is an arbitrary non-negative integer cannot be represented as unnested PROLOG structures but can be represented as unnested FIT compounds using the distinguished first element CLASS and varying numbers of remaining elements, as shown in the following table. The left column shows the usual mathematical set notation, the inner column shows equivalent FIT compounds, and the right column shows a corresponding PROLOG-like functor-argument notation, which however, is not realizable in PROLOG because for each number k of different arities for which the functor 'class' might be defined there is a number k+1 such that 'class' is not defined for arity Nk+1 [the table shows k=4, N1=0, N2=2, N3=3, N4=6].

{}	(CLASS)	class()
{1,3}	(CLASS 1 3)	class(1,3)
{A,B,C}	(CLASS A B C)	class(a,b,c)
{A,B,C,1,2,3}	(CLASS A B C 1 2 3)	class(a,b,c,1,2,3)
...	...	...

Programmers used to LISP, where many functions, e.g. associative ones like APPEND, have an arbitrary number of arguments, must feel that this is an unnecessary restriction on expressiveness; and indeed, the LISP-based LM-PROLOG (Kahn 1983) introduces variable-arity functors into a LISP/PROLOG environment.

We now show how the FIT CLASS compounds, exemplified in the inner column, may be defined for arbitrary k. In general, FIT compounds, in contrast to PROLOG structures, can be interpreted as value-returning function calls, where the distinguished first element ['functor'] plays the role of a function applied to the arguments in the remaining element positions. This permits FIT's so-called 'self-normalizing collections', generalizing those in QA4/QLISP (Rulifson et al. 1972), which are compounds that return their own normalized form. For example, CLASS in FIT is defined as a normalization function for sets, removing duplicate arguments and sorting the remaining ones lexicographically. Thus (CLASS 1 3) returns itself and (CLASS B A C B C) returns (CLASS A B C). The CLASS definition can be expressed in FIT itself by [the variable >X enables varying arities k]

```
(>(CLASS >X) (CONS CLASS (SORT @(<X) LEXORDER NODUPS)))
```

with SORT being LISP's sorting function or its FIT redefinition shown in section 8.1 ["@" instantiates a list whose contents is the imposition of CLASS elements]. For efficiency, however, we normally use a CLASS version defined entirely in FIT-1's implementation language, LISP.

In ordinary PROLOGs, variable-length structures can only be simulated by nestings of fixed-length structures. In particular, PROLOG borrows LISP's representation of N-element lists as nestings of N 2-element dotted pairs. Thus our previous right-recursive nesting of dotted pairs

```
.(alfa,.(beta,.(gamma,nil)))
```

in PROLOG can be abbreviated to the so-called 'list notation'

```
[alfa,beta,gamma]
```

i.e. it corresponds to the LISP list

```
(alfa beta gamma)
```

However, in PROLOG this is only a variable-length surface syntax for basically fixed-length "."-structures. We feel that this is no solution to the fixed-length restriction, for the following reasons:

1. PROLOG's list notation does not abstract from its underlying dotted pair form, because for the pattern-matching selection of list elements a "|"-operator is used which corresponds directly to the "."-operator [This is similar to the CAR and

CDR functions in LISP which, however, can be viewed as selectors of an abstract data type; dotted pairs never need to become visible to the LISP programmer and modern LISP textbooks such as (Winston & Horn 1981) don't even use them for association lists. In FIT no binary dotted-pair structure at all becomes visible on pattern-matching selection of compound elements, independent of their implementation; cf. section 5.1].

2. For variable-length structures other than lists no surface syntax is provided, although this would be very desirable for sets, i.e. writing {a,b,c}, etc. [Since the available bracket types are not sufficient, FIT uses only ordinary parentheses, as in (CLASS A B C), whose 'type' can be seen from the distinguished first element, here CLASS. Since in PROLOG variable-length structures are represented as dotted pairs using an 'auxiliary' "."-functor, variable-length structures cannot use a 'proper' functor, analogous to a distinguished element in FIT, say CLASS].

Mainly for these reasons we feel that (Stefik et al. 1983) are correct in depicting the connection of list operations to PROLOG as a "patch approach" because they were added to the language after the initial design.

Besides their disadvantages, PROLOG's structures have also two advantages as compared with ordinary LISP lists, which they share, however, with FIT's collection compounds.

1. The functor of a structure indicates the 'type' of that entire structure, which may sometimes enhance readability and which can help in matching. E.g. the matching of data with incompatible types, say of apples a and b with pears a and b, immediately fails in PROLOG's structure representation,

   ```
   apples(a,b) = pears(a,b).
   ```

   whereas that matching would yield an unwanted success in a naive type-less LISP list representation

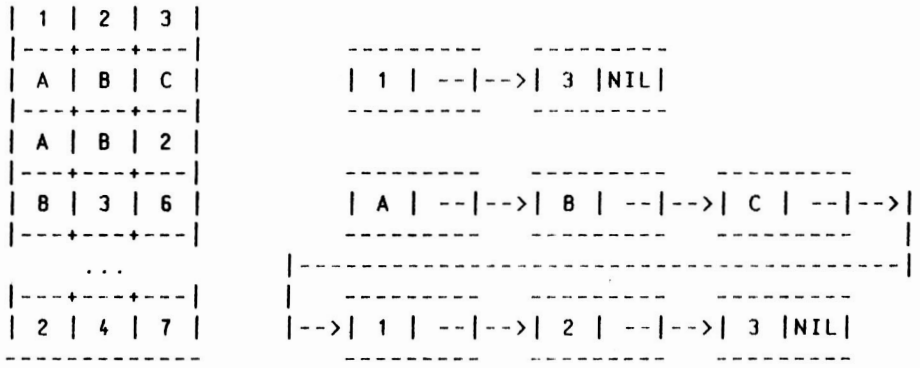
   ```
   (SETQ APPLES '(A B))
   (SETQ PEARS '(A B))
   (MATCH APPLES PEARS) or (MATCH '(A B) '(A B))
   ```

   but it again immediately fails in FIT's typed collection compound representation

   ```
   ('(APPLES A B) '(PEARS A B))
   ```

2. Access to the arguments of PROLOG structures can be implemented efficiently [constant time] because their fixed length allows array-like random access to every argument [cf. the vector of cells called a "frame" in "structure sharing" (Warren et al. 1977)], whereas LISP lists are less efficient [linear time] because their varying length seems to require CDRing through from left to right to the desired element [even if the "CDR-coding" technique of the LISP machine (Weinreb et

al. 1983) is used]. However, FIT collection compounds which are only used with a fixed arity or with a small number of fixed arities can also be implemented array-like instead of with list pointers. For varying-length compounds many instances of which have some fixed arity, a mixture of list-like [varying-length case] and array-like [fixed-length case] implementation is possible. For example if there are many three-element sets, their compound notations, e.g. (CLASS A B C), (CLASS A B 2), (CLASS B 3 6), ..., (CLASS 2 4 7), can be implemented as the array shown in the table on the left side below. Other sets, such as (CLASS 1 3) and (CLASS A B C 1 2 3), can be implemented as the ordinary lists shown in the pointer diagram on the right below.



5 PATTERN MATCHING AND GENERALIZATIONS

FIT and PROLOG are both languages built around generalized concepts of pattern matching. These are based on the asymmetric pattern-data matching found in many AI languages, for which FIT provides a richer set of primitives than PROLOG [cf. subsection 5.1]. This basic matching concept is generalized differently in the two languages. FIT generalizes it to asymmetric adapter-data fitting, where the adapter, an operator derived from a pattern by allowing it to contain functions, both tests and locally transforms data [cf. subsection 5.2]. PROLOG generalizes it to symmetric pattern-pattern unification, where both patterns are made equal through variable-value substitutions [cf. subsection 5.3].

5.1 Basic Matching: Variables in Patterns

Although basic patterns may consist of variables and constants, we restrict our attention to pattern variables here, pattern constants being almost the same in PROLOG and FIT [except that unprefixed atoms are first applied as functions in FIT, so that a "%" prefix is used to enforce an exclusive constant interpretation].

In PROLOG, pattern variables can match arbitrary list elements but not arbitrary list segments. The only kind of list segment a PROLOG variable can match is a tail segment [this must be specified in the pattern by a "|" head/tail separator, i.e. it is not a property of the

variable but of the entire pattern]. An initial or an intermediate list segment cannot be matched in PROLOG; hence multiple segments aren't possible either. Thus in PROLOG there is a fundamental asymmetry between head and tail, inherited from the binary dotted pair representation of lists as "."-structures. Although this representation is hidden in the list notation, it comes to the surface during matching. PROLOG's dotted pair matching is well-known from some other PLANNER-like AI languages, such as CONNIVER.

FIT, like most other PLANNER-like AI languages, such as FUZZY, uses both element variables [prefixed by "?"] and segment variables [in FIT additionally 'post'-fixed by "oLIST"]; thus we formalize a segment variable as a fitter composition of a "?"-variable with the LIST function. Unlike previous languages, FIT also allows the use of imposition variables [prefixed by ">"] which like segment variables match sequences of list elements but unlike these are bound to the element sequences themselves, rather than to their LISTified form. FIT's segment and imposition variables can occur at arbitrary positions and arbitrarily often inside patterns.

The following table compares matching in PROLOG and FIT, showing the higher expressiveness of FIT patterns. For each PROLOG match example, except the first, a directly corresponding FIT match [using "oLIST"] is written in the same row and a more typical, imposition-variable FIT match [using ">"] is written in the next row. Further FIT rows show variations on the original match, with segment and imposition variables occurring in non-tail positions and occurring more than once. The bindings resulting from matches are written below each match [the empty imposition is denoted by (IMPOSITION)]. For non-deterministic matches, not possible in PROLOG, each set of bindings is written in a separate line.

PROLOG	FIT
[X,Y,X] = [a,b,a]. X=a, Y=b	('(?X ?Y ?X) '(A B A)) X=A, Y=B
[X|Y] = [a,b,c]. X=a, Y=[b,c]	('(?X ?YoLIST) '(A B C)) X=A, Y=(B C)
	('(?X >Y) '(A B C)) X=A, Y=B C
	('(?XoLIST ?Y) '(A B C)) X=(A B), Y=C
	('(>X ?Y) '(A B C)) X=A B, Y=C
[X|Y] = [a,b]. X=a, Y=[b]	('(?X ?YoLIST) '(A B)) X=A, Y=(B)
	('(?X >Y) '(A B)) X=A, Y=B
	('(?XoLIST ?Y) '(A B)) X=(A), Y=B

```
                                ('(>X ?Y) '(A B))
                                X=A, Y=B

[X|Y] = [a].                    ('(?X ?YoLIST) '(A))
X=a, Y=[]                       X=A, Y=()

                                ('(?X >Y) '(A))
                                X=A, Y=(IMPOSITION)

                                ('(?XoLIST ?Y) '(A))
                                X=(), Y=A

                                ('(>X ?Y) '(A))
                                X=(IMPOSITION), Y=A

[X,Y|Z] = [a,b,c,d].            ('(?X ?Y ?ZoLIST) '(A B C D))
X=a, Y=b, Z=[c,d]               X=A, Y=B, Z=(C D)

                                ('(?X ?Y >Z) '(A B C D))
                                X=A, Y=B, Z=C D

                                ('(?XoLIST ?Y ?Z) '(A B C D))
                                X=(A B), Y=C, Z=D

                                ('(>X ?Y ?Z) '(A B C D))
                                X=A B, Y=C, Z=D

                                ('(?X ?YoLIST ?Z) '(A B C D))
                                X=A, Y=(B C), Z=D

                                ('(?X >Y ?Z) '(A B C D))
                                X=A, Y=B C, Z=D

                                ('(?X ?YoLIST ?ZoLIST) '(A B C D))
                                X=A, Y=(B C D), Z=()
                                X=A, Y=(B C), Z=(D)
                                X=A, Y=(B), Z=(C D)
                                X=A, Y=(), Z=(B C D)

                                ('(?X >Y ?ZoLIST) '(A B C D))
                                X=A, Y=B C D, Z=()
                                X=A, Y=B C, Z=(D)
                                X=A, Y=B, Z=(C D)
                                X=A, Y=(IMPOSITION), Z=(B C D)

                                ('(?X >Y >Z) '(A B C D))
                                X=A, Y=B C D, Z=(IMPOSITION)
                                X=A, Y=B C, Z=D
                                X=A, Y=B, Z=C D
                                X=A, Y=(IMPOSITION), Z=B C D

                                ('(?XoLIST ?Y ?ZoLIST) '(A B C D))
                                X=(A B C), Y=D, Z=()
                                X=(A B), Y=C, Z=(D)
                                X=(A), Y=B, Z=(C D)
                                X=(), Y=A, Z=(B C D)

                                . . .
```

```
('(>X >Y >Z) '(A B C D))
X=A B C D, Y=(IMPOSITION), Z=(IMPOSITION)
X=A B C, Y=D, Z=(IMPOSITION)
X=A B C, Y=(IMPOSITION), Z=D
X=A B, Y=C D, Z=(IMPOSITION)
X=A B, Y=C, Z=D
X=A B, Y=(IMPOSITION), Z=C D
X=A, Y=B C D, Z=(IMPOSITION)
X=A, Y=B C, Z=D
X=A, Y=B, Z=C D
X=A, Y=(IMPOSITION), Z=B C D
X=(IMPOSITION), Y=A B C D, Z=(IMPOSITION)
X=(IMPOSITION), Y=A B C, Z=D
X=(IMPOSITION), Y=A B, Z=C D
X=(IMPOSITION), Y=A, Z=B C D
X=(IMPOSITION), Y=(IMPOSITION), Z=A B C D
```

An important difference between PROLOG and FIT pattern matching not shown in the table should be mentioned. While a successful match in PROLOG simply prints the resulting variable bindings, in FIT it returns the data instance matched and as its effect yields the bindings. For example, the match in the first table row in impure FIT would return (A B A) and bind X to A and Y to B. The next subsection will show that pattern's returning of unchanged data instances generalizes gracefully to adapter's returning of modified data instances. Semantically, the values returned and the bindings yielded are treated as one value pair of the form (GLOBAL (bindings) values). Thus the example match in pure FIT would return the GLOBAL expression (GLOBAL ((>X A) (>Y B)) (A B A)). GLOBAL expressions may migrate out of other expressions, uniting their bindings consistently and leaving their values behind. Indeed, the above GLOBAL expression results from an intermediate LIST expression with three embedded GLOBAL expressions as shown in the following trace of the sample match evaluation:

```
('(?X ?Y ?X) '(A B A))
  (LIST (?X A) (?Y B) (?X A))
  (LIST (GLOBAL ((>X A)) A) (GLOBAL ((>Y B)) B) (GLOBAL ((>X A)) A))
(GLOBAL ((>X A) (>Y B)) (A B A))
```

Finally, matches in FIT can not only be performed on lists but also on impositions. For example, the list match in the first row of the above table can be rewritten to the imposition match [the colon separates pattern and data impositions]

```
(?X ?Y ?X : A B A)
```

which also binds X to A and Y to B but returns the imposition A B A instead of the list (A B A). For the other table rows in the FIT column the same parenthesis-saving imposition-rewriting is possible.

5.2 Fitting: Special Elements in Patterns or Functions in Adapters

Since non-trivial adapters are a main theme of FIT [dealt with in (Boley 1983)] but are absent in PROLOG, they are not explored in great detail in the context of this FIT/PROLOG comparison; however, section 7 will show the use of adapters for defining functions.

5.2.1 Simple Adapters -

Most pattern matchers provide something like 'don't care' or 'match all' pattern elements, in PROLOG called 'anonymous variables' and written "_". In FIT this special [non-constant, non-variable] pattern element is formalized using the identity function ID, which is applicable to one arbitrary element and returns it unchanged. Patterns containing functions in FIT are called 'adapters'. Thus a PROLOG pattern

```
[A,_,_]
```

successfully matching lists like [a,b,b] and [a,b,c], but neither [a,b] nor [a,b,b,c,c], becomes the FIT adapter

```
(A ID ID),
```

successfully fitting lists like (A B B) and (A B C), but neither (A B) nor (A B B C C), i.e.

```
('(A ID ID) '(A B B)) returns (A B B),
('(A ID ID) '(A B C)) returns (A B C),
('(A ID ID) '(A B)) yields jF,
('(A ID ID) '(A B B C C)) yields jF.
```

ID is only a trivial example of the arbitrary functions allowed in FIT adapters. A similar example is the absorption function AB, definable by (>(AB ?X)), which is applicable to one arbitrary element and returns the empty imposition:

```
('(A AB AB) '(A B B)) returns (A),
('(A AB AB) '(A B C)) returns (A),
('(A AB AB) '(A B)) yields jF,
('(A AB AB) '(A B B C C)) yields jF.
```

A less trivial function is NUMBERP, a predicate for numbers, as applicable in the successful adapter fitting

```
('(A NUMBERP C) '(A 2 C)), returning (A T C)
```

and in the failing adapter fitting

```
('(A NUMBERP C) '(A B C)), yielding jF.
```

Functions inside adapters need not be unary, as shown by the successful adapter fitting [matching A to A and applying LESSP to 2 3]

```
('(A LESSP) '(A 2 3)), returning (A T)
```

and the failing adapter fitting

('(A LESSP) '(A 3 2)), yielding jF.

Besides such predicate-like functions, arbitrary general functions are also allowed inside adapters. For instance, one adapter fitting generalization of the match in the first row in the table in subsection 5.1 is

('(?X LIST ?X) '(A B A))

which binds X to A and returns (A (B) A). The semantic tracé of this evaluation corresponds to that in section 5.1:

```
('(?X LIST ?X) '(A B A))
  (LIST (?X A) (LIST B) (?X A))
  (LIST (GLOBAL ((>X A)) A) (B) (GLOBAL ((>X A)) A))
(GLOBAL ((>X A)) (A (B) A))
```

There are operators making new fitters from old, e.g. the "#" [REPEAT] operator. For example, the 'repeated identity' #ID allows the following fittings:

```
('(A #ID) '(A B B)) returns (A B B)
('(A #ID) '(A B C)) returns (A B C)
('(A #ID) '(A B)) returns (A B)
('(A #ID) '(A)) returns (A)
('(A #ID) '(A B B C C)) returns (A B B C C)
('(A #ID) '(A B C D E F G)) returns (A B C D E F G).
```

Similarly, (A #B) successfully fits (A B B), (A B), and (A), but none of the other examples above. Also, (A #NUMBERP) successfully fits (A 2 3), returning (A T T) and (A #ADD1) successfully fits (A 2 3), returning (A 3 4).

5.2.2 A PROLOG Simulation -

In PROLOG, the adapter (A #ADD1), for instance, can be simulated by a relation named a_repsucc, using maplist (Clocksin & Mellish 1981) for modeling "#":

```
a_repsucc([a|L],[a|M]) :- maplist(succp,L,M).
```

Now FIT's fitment ('(A #ADD1) '(A 2 3)), returning (A 3 4), can be simulated by PROLOG's relation call a_repsucc([a,2,3],Ans), binding Ans to [a,3,4]. Notice that PROLOG must give a name, like a_repsucc, to every program, even if it is used only once, whereas anonymous programs are allowed in most other languages [cf. not only FIT's adapters above but also LISP's LAMBDA expressions and FIT's TRAFOs below].

A slightly more general adapter, (#SUB1 0 #ADD1), successfully fits number lists containing a 0, returning the predecessors of all numbers before the 0 and the successors of all numbers after the 0. For example the adapter fitment ('(#SUB1 0 #ADD1) '(3 6 0 4 2 7)) returns (2 5 0 5 3 8). In PROLOG this must be simulated by a considerably more general predicate, named reppred_0_repsucc, using recursion for modeling the

first "#" application [note the 'reverse' clause order required here]:

```
reppred_0_repsucc([0|L],[0|M]) :- maplist(succp,L,M).
reppred_0_repsucc([X|L],[Y|M]) :- predp(X,Y), reppred_0_repsucc(L,M)
```

Then the relation call corresponding to the above adapter fitment is reppred_0_repsucc([3,6,0,4,2,7],Ans), binding Ans to [2,5,0,5,3,8]. If we wanted to model both "#"-applications with maplist, i.e. by using maplist not only for the segment after the 0 but also for that before the 0, we might apply append [cf. section 7.1] for locating the 0 and splitting the list into the required segments:

```
reppred_0_repsucc(In,Out) :-
   append(PIn,[0|SIn],In),
   maplist(predp,PIn,POut),
   append(POut,[0|SOut],Out),
   maplist(succp,SIn,SOut).
```

Although in this reppred_0_repsucc version the 'interleaved' order of the append and maplist calls [in contrast to 'first I/O partition, then mapping' orders], proposed by (Fuhlrott 1983) and tested in micro-PROLOG, may not look obvious, it is crucial for preventing non-deterministic calls like reppred_0_repsucc([3,0,5,0,7],Ans) and conjunctive relation calls like

```
?- reppred_0_repsucc([3,0,7],Ans), member(4,Ans).
```

from diverging on their backtrack search for a second solution. The fact that the relational reppred_0_repsucc representation of an adapter as simple as (#SUB1 0 #ADD1) involves these non-trivial programming considerations indicates that relational programming may at times appear quite low-level if compared with higher-order functional or adapter programming. Still, like the FIT adapters, in PROLOG neither a_repsucc nor the two versions of reppred_0_repsucc work if used from right to left: For example, reppred_0_repsucc(Ans,[2,0,8]) yields a 'finite error' in the recursive version and an 'infinite error' in the append-using version.

5.2.3 TRAFO and COMFO Expressions -

Apart from the fact that adapters themselves are normally unnamed, functions inside adapters need not be named, like NUMBERP, but may also be anonymous, like (TRAFO ?X (GREATERP <X 8)), corresponding to LISP's (LAMBDA (X) (GREATERP X 8)). For example, the adapter fitment

```
('(1 (TRAFO ?X (GREATERP <X 8)) 8 ?X) '(1 9 8 3))
```

successfully applies the TRAFO expression (TRAFO ?X (GREATERP <X 8)) to 9, binds ?X to 3, and returns (1 T 8 3). Note that the TRAFO variable ?X is unrelated to the variable ?X, bound to 3: While the former is local to the TRAFO, the latter is global to the adapter. If the TRAFO body (GREATERP <X 8) is regarded as a type check over the TRAFO variable ?X, analogous to the type check performed by NUMBERP for the typed variable x?NUMBERP, then the TRAFO's localization effect may well be incorrect.

To leave the variable ?X global to the adapter, the composition (COMPOSE (TRAFO ID (GREATERP ^X 8)) ?X) can be used instead. For example,

```
('(1 (COMPOSE (TRAFO ID (GREATERP ^X 8)) ?X) 8 ?X) '(1 9 8 3))
```

successfully applies the composition to 9 by first binding ?X to 9 and then evaluating (GREATERP ^X 8) in the global environment thus created, but altogether fails because of the inconsistency of this environment with the binding of ?X to 3. A similar, but altogether successful, adapter fitment is

```
('(1 (COMPOSE (TRAFO ID (GREATERP ^X 8)) ?X) 8 ?X) '(1 9 8 9))
```

binding X to 9 and returning (1 T 8 9). Such COMPOSE expressions are more generally usable and may thus be given a name, COMFO [COMPOSE-TRAFO], which can be introduced by the definition

```
(COMFO pattern body) = (COMPOSE (TRAFO ID body) pattern)
```

or, more generally,

```
(COMFO pattern1 ... patternM : body1 ... bodyN) =
(COMPOSE (TRAFO #ID body1 ... bodyN) pattern1 ... patternM)
```

Using a COMFO expression, our previous adapter is shortened to (1 (COMFO ?X (GREATERP ^X 8)) 8 ?X) and its sample fittings become:

```
('(1 (COMFO ?X (GREATERP ^X 8)) 8 ?X) '(1 9 8 3)) yields jF

('(1 (COMFO ?X (GREATERP ^X 8)) 8 ?X) '(1 9 8 9)) returns (1 T 8 9) and
                                                   binds X to 9.
```

Notice that the COMFO expression has the same structure as the initial TRAFO example. Indeed, TRAFO and COMFO form a nice symmetrical pair, as characterized by the equations [the first generalizes beta-reduction in LAMBDA calculus]

```
((TRAFO pattern body) expr)  =  (LOCAL  (pattern expr) : body)
((COMFO pattern body) expr)  =  (GLOBAL (pattern expr) : body)
```

For the use of COMFO expressions in invocation adapters see section 6.3.2.

5.2.4 Boolean Fitter Operators -

Finally, consider the 'boolean pattern operators' POR, PAND, and PNOT which are available in almost all classic pattern matchers [see, e.g., (Rulifson et al. 1972)]. In FIT they are generalized to 'boolean fitter operators' and are formally explained by the respective logical connectives for disjunction, conjunction, and negation. For example,

```
((POR (?X ?Y ?X) (?X ?Y ?Y) (?X ?X ?Y)) '(A B B))
```

succeeds because one of the pattern matches to which it is reduced, ('(?X ?Y ?Y) '(A B B)) succeeds.

((PAND (ID NUMBERP ID) (?X LIST ?X)) '(A 2 C))

fails because one of the adapter fittings, ('(?X LIST ?X) '(A 2 C)) fails.

((PNOT LESSP) 3 2)

succeeds because (LESSP 3 2) fails. In PROLOG boolean operators on patterns are lacking, perhaps because they cannot be generalized to unification in a simple manner [cf. subsection 5.3].

5.3 Unification: Variables in Two Patterns

PROLOG uses unification implicitly for fact retrieval and rule invocation. Unification can also be done explicitly by the user with the "=" [equality] predicate. For example one unification generalization of the match in the first row in the table in subsection 5.1 is

[X,Y,X] = [a,b,Z].

which binds X and Z to a and Y to b. The prominent role of unification in PROLOG becomes even more important in UNIFORM (Kahn 1981), which uses augmented unification as its sole basis. However, the notion that PROLOG itself bases its computation entirely on unification is exaggerated: this would only be true if there were facts only; rules, although invoked through unification of their heads with a request, through resolution transform the request into a conjunction of other requests in the unification-extended environment. FIT's adapters, on the other hand, share with facts the property of being 'invocation-computing': all adapter computation is performed during invocation fitting [an adapter has completed its work when its invocation has been completed]; no global rule-like head-to-body transformation is performed [a rule has completed its work only when the computation of its body has been completed].

FIT-1 uses a restricted form of implicit unification but doesn't use explicit unification since it regards patterns as operator-like active entities ['fitters'] which are matched to operand-like passive entities ['fittees'] in the usual asymmetric operator-operand manner. Furthermore, the general patterns permitted in FIT [arbitrary numbers of imposition or segment variables] would make symmetric pattern-pattern unification matches computationally as complex as string unification. Finally, symmetric adapter-adapter unification fitting poses new problems which are not yet well understood. [A function paired with a variable may leave its application pending until that variable receives a value; a function paired with a function might generate a value of its range which is also in the range of the other function.]

To be sure, there would be no problem in implementing unification for FIT if patterns to be unified had to have the restricted form of PROLOG patterns. To put it differently, PROLOG and any other language would have the same problems as FIT would if it desired to incorporate more general patterns [in particular, multiple segment variables, which are very convenient for the user and pose no serious problems in

asymmetric matching] and still desired to perform symmetric unification matching on these [in particular, the problem of string unification complexity]. QLISP may actually have had these problems among others.

We thus decided to restrict FIT-1's explicit fitting to the asymmetric case until issues of unification matching are better understood [for an overview of what is known and what is still open see (Siekmann & Szabo 1982)]. In any case, with FIT-1's other match generalizations [e.g. adapters] available, this restriction didn't turn out to be such a great hindrance in practical programming tasks.

6 HORN CLAUSES AND IMPLICIT FITTERS

Definitions in PROLOG are made by storing Horn clauses and in FIT by storing fitters into the global data base. Stored fitters are also called 'implicit fitters' and are dual to 'explicit fitters' which the user directly fits to fittees. PROLOG's Horn clauses are divided into facts and rules. FIT's corresponding implicit fitters are divided into implicit adapters and transformers. However, to represent PROLOG facts only very special FIT adapters, namely simple patterns, are needed. Similarly, to represent PROLOG rules only very special FIT transformers are required; alternatively, PROLOG rules can often be more concisely represented as FIT adapters [cf. section 7].

6.1 Facts

A PROLOG fact is a structure of the form f(a1,a2,...,aN), globally stored by

```
f(a1,a2,...,aN).
```

where the arguments aI's can be constants, variables, or substructures. That the period after the structure indicates the storing, not the query of the structure, can only be seen at the lack of a "?-" prefix. [In PROLOG's rudimentary interactive programming the system by default is in a mode where it expects each input to be a query, hence uses "?-" directly as a prompt. To store facts, the user must first switch off this prompt by entering a storing mode. After storage is completed, one must not forget to reenter the default mode before asking a query.]

A corresponding FIT fact is a compound of the form (f~ a1~ a2~ ... aN~), globally stored by

```
GLOBAL:((f~ a1~ a2~ ... aN~))
```

where the tilded symbols are transformed versions of those in PROLOG as explained in section 4, with one addition: PROLOG variables are rewritten to FIT variables by Variable~=?VARIABLE. The storing is simply indicated by the embedding of the compound into a GLOBAL:(. . .) expression. [In FIT's LISP/PLANNER-like interactive programming no mode change is necessary for storing, hence a modeless "*"-prompt is used. The "GLOBAL:" prefix makes clear that a, possibly one-element, list of facts is to be stored.]

In both PROLOG and FIT, structures/compounds containing constants and variables along with other such structures/compounds can be used as explicit patterns in explicit matches or, after having stored them in the data base, as implicit patterns in implicit matches. Therefore PROLOG and FIT facts actually are implicit patterns.

Simple facts without variables were exemplified in section 2.1; for a fact example with variables, consider the phrase "The successor of something is greater than that thing", which can be stored as the PROLOG fact

```
greater(successor(X),X).
```

and as the FIT fact

```
GLOBAL:((GREATER (SUCCESSOR ?X) ?X))
```

In this example, the first argument of the greater structure is a successor structure. Notice that the top-level functor greater is a predicate whereas the sublevel functor successor is a function. In general, PROLOG, like predicate logic, allows a functional notation in sublevels but not on the top-level [in sublevels it doesn't matter that these notations cannot be evaluated, on the top-level it would]; thus, unlike the above PROLOG example, successor(X) :- X+1 is not a legitimate PROLOG clause. FIT, like all functional languages, allows functions on every level; thus, just like the above FIT example, (>(SUCCESSOR ?X) (ADD1 <X)) is a legitimate FIT clause.

When now the PROLOG question

```
?- greater(successor(3),3).
```

or the FIT question

```
(GREATER (SUCCESSOR 3) 3)
```

is posed, an implicit match corresponing to the explicit PROLOG match

```
greater(successor(X),X) = greater(successor(3),3).
```

or to the explicit FIT match

```
('(GREATER (SUCCESSOR ?X) ?X) '(GREATER (SUCCESSOR 3) 3))
```

is used to answer it affirmatively. The main difference between explicit and implicit matches is the treatment of resulting variable bindings [here X=3]: Bindings of variables occurring in explicit patterns become visible; those of variables occurring in implicit patterns remain hidden.

When the PROLOG question

```
?- greater(successor(Y),3).
```

is posed, an implicit unification match corresponing to the explicit PROLOG unification match

```
greater(successor(X),X) = greater(successor(Y),3).
```

is used to answer it affirmatively and binding Y to 3; when the FIT question

(GREATER (SUCCESSOR |?Y) 3)

is posed, an implicit unification match is used not corresponing to an explicit unification match and binding Y to 3 [in the implicit unification match the binding Y=3 becomes visible because Y occurs in the request pattern; the binding X=3 remains hidden because X occurs in the implicit pattern]. The current FIT-1 only supports such restricted implicit unification matching but no explicit unification matching, as discussed in subsection 5.3.

6.2 Rules

A PROLOG rule has the form structure0 :- structure1, ..., structureN and is globally stored by

structure0 :- structure1, ..., structureN.

where the structureI's are structures as in facts. The storing is again indicated by the period after the structures in the absence of a "?-" prefix. A corresponding FIT rule has the form

```
(TRAFO structure0~
       (LOCAL (>var1: ... >varM:) structure1~ ... structureN~))
```

and is stored globally as

```
({>,v}structure0~
     '(LOCAL (>var1: ... >varM:) structure1~ ... structureN~))
```

where the tilded structureI's are the usual transformed versions of those in PROLOG rules and the varI's are the request variables being used in structure1~, ..., structureN~. Their LOCAL declaration is necessary to prevent name conflicts between the request variables of different rule bodies. If there are no request variables a FIT rule can be simplified to (TRAFO structure0~ structure1~ ... structureN~) which is stored as

```
({>,v}structure0~ 'structure1~ ... 'structureN~)
```

Here and later on the meta-language expression "{>,v}" stands for either of the object-language symbols ">" or "v". The ">" [SHOVE] and "v" [VEL] prefixes effect rule storing by setting rule heads, structure0~, to rule bodies. The SHOVE prefix specifies an ordinary setting, where several body assignments to the same head cause the old rules to be erased on storage of the new ones. The VEL prefix specifies a 'non-deterministic' setting, where several body assignments to the same head cause all rules to be stored and subsequently to be used non-deterministically. Since settings evaluate to GLOBAL expressions [cf. section 5.1], no user-provided "GLOBAL:" prefix is necessary for rule storing.

The "'" [QUOTE] prefix in front of the LOCAL body and the structureI~ bodies [1≤I≤N] prevents their evaluation at storing-time; in internal store, the "'" prefix is removed; hence such QUOTEs are usually elided from the FIT examples.

As an example consider the phrase "something is even if it is an integer divisible by two", which can be stored as the PROLOG rule

```
even(X) :- integer(X), divisible(X,2).
```

and as the FIT rule

```
(>(EVEN ?X) (INTEGER <X) (DIVISIBLE <X 2))
```

When now the PROLOG question

```
?- even(8).
```

or the FIT question

```
(EVEN 8)
```

is asked, the rule head even(X) or (EVEN ?X) is matched to the question even(8) or (EVEN 8) and the rule body is evaluated with the resulting binding X=8.

In PROLOG such an implicit rule application doesn't correspond to an explicit one that is directly specified by the user. In FIT it corresponds to the explicit rule application

```
((TRAFO (EVEN ?X) (INTEGER <X) (DIVISIBLE <X 2)) '(EVEN 8))
```

using the explicit TRAFO notation of transformers which generalizes the usual LAMBDA expressions. Often TRAFO expressions are used with the isolated variables of patterns, instead of with complete invocation patterns, as their left-hand sides; this form of TRAFO specifies 'anonymous' rules and is equivalent with LAMBDA expressions. For example, the previous TRAFO application can be shortened to [the 'name' EVEN is omitted]

```
((TRAFO ?X (INTEGER <X) (DIVISIBLE <X 2)) 8)
```

While PROLOG allows only such relational rules [computing truth values], FIT also allows functional rules [computing arbitrary values]. For example, the phrase "the division of a first thing by a second thing is the quotient and the remainder of the first by the second" can be stored in FIT as the rule

```
(>(DIVISION ?X ?Y) (QUOTIENT <X <Y) (REMAINDER <X <Y))
```

In PROLOG the phrase must first be put into the awkward relational form "four things are in a division relation if the first three things are in a quotient relation and the first two things and the fourth thing are in a remainder relation" before it can be stored as the rule

```
divisionp(X,Y,Q,R) :- quotientp(X,Y,Q), remainderp(X,Y,R).
```

which in FIT could also be stored as

```
(>(DIVISIONP ?X ?Y ?Q ?R) (QUOTIENTP <X <Y <Q) (REMAINDERP <X <Y <R))
```

Functional FIT rules can also be used in explicit applications. For example,

```
((TRAFO (DIVISION ?X ?Y) (QUOTIENT <X <Y) (REMAINDER <X <Y))
 '(DIVISION 7 2))
```

returns 3 1. Using the pattern variables alone as TRAFO left-hand sides we get the anonymous rule application

```
((TRAFO ?X ?Y : (QUOTIENT <X <Y) (REMAINDER <X <Y))
 7 2)
```

Let us summarize a fact/rule implicit/explicit tradeoff in FIT/PROLOG:

While FIT unification can only be used implicitly to access facts and rules stored in the data base, PROLOG unification can also be used explicitly on non-stored structures.

While PROLOG rules can only be used implicitly, namely after they have been stored in the data base [and named by a functor], FIT rules can also be used explicitly [and anonymously] without such prior storing.

6.3 Clauses with Constraints

6.3.1 PROLOG II Constraints and their LOCAL Representation -

Although the simplicity of Horn clauses has definite advantages with respect to formal semantics it now seems clear that they are too simple for real-live programming. One possible generalization of Horn clauses has been recently proposed in (Colmerauer 1983) for PROLOG II. In this proposal a clause can be augmented by "constraints" which are sets of equalities and inequalities over variables. All constraints must be fulfilled for a clause to be successful. Facts and rules with constraints c1, c2, ..., cK in PROLOG II are written thus [a fact is regarded as a rule with an empty body]

```
structure0 -> , {c1, c2, ..., cK};
structure0 -> structure1 ... structureN, {c1, c2, ..., cK};
```

where the cI's either have the form varR=varS or the form varR\=varS [we use Edinburgh PROLOG's "\=" to denote inequality]. For clauses without constraints [K=0] in PROLOG II the meaningless part ", {}" is omitted.

In FIT these clauses may be rewritten to [for simplicity we assume that no request variables are used]

```
({>,v}structure0~ c1~ c2~ ... cK~)
({>,v}structure0~
      (LOCAL c1~ c2~ ... cK~ : structure1~ ... structureN~))
```

where cI = varR=varS yields cI~ = (EQ varR~ varS~) and cI = varR\=varS yields cI~ = (NEQ varR~ varS~). Constraints in FIT thus become

implicitly conjoined rule bodies [for facts] or left-imposition arguments of "if then" LOCALs used as rule bodies [for rules]. A constrained fact (>structure0~ c1~ c2~ ... cK~) can be viewed as an abbreviation for a constrained rule with empty LOCAL body (>structure0~ (LOCAL c1~ c2~ ... cK~ :)). Clauses without constraints are rewritten into FIT as usual.

An example of a constrained fact is the following diffchain predicate, holding for all triples without equal adjacent elements:

```
diffchain(x,y,z) -> , {x\=y, y\=z};
```

In FIT this can be rewritten as

```
(>(DIFFCHAIN ?X ?Y ?Z) (NEQ <X <Y) (NEQ <Y <Z))
```

In order to illustrate a constrained rule let us consider a slightly corrected version of the out-of definition in (Colmerauer 1983), a simple list predicate which in functional notation would trivialize to NOToMEMBER.

```
out-of(u,nil) ->;
out-of(u,v.l) ->
   out-of(u,l),
   {u\=v};
```

In FIT this can be rewritten as

```
(OUT-OF ?X NIL)
(>(OUT-OF ?X (?Y ?LoLIST))
 (LOCAL (NEQ <X <Y) : (OUT-OF <X <L)))
```

As useful as Colmerauer's constraints may be, it remains doubtful whether these simple equality and inequality constraints are sufficient for all applications. For instance, many programmers [not only in fields like operations research] may wish to have the full set of relational operators [i.e. also including "<", "≤", ">", and "≥"] for expressing inequation constraints. As examples, consider the predicates lesschain and least-of derived, respectively, from diffchain and out-of by replacing "\=" by "<". This is possible in FPL (Bellia et al. 1982). In FIT it is also no problem because LOCAL expressions can, of course, not only be used with the EQ and NEQ predicates but allow for arbitrary constraints [incl. LESSP, LE, GREATERP, and GE]. For example, as we used (LOCAL (NEQ <X <Y) : ...) in the OUT-OF program, we can use (LOCAL u(GREATERP <X <Y) : ...) in, say, Euclid's algorithm for computing the greatest common divisor [the "u" operator transforms jF to jU, which is necessary for handling the non-determinism arising from the first two rules]:

```
(v(EUCLID ?X ?Y)
 (LOCAL u(GREATERP <X <Y) : (EUCLID (DIFFERENCE <X <Y) <Y)))
(v(EUCLID ?X ?Y)
 (LOCAL u(GREATERP <Y <X) : (EUCLID <X (DIFFERENCE <Y <X))))
(>(EUCLID ?X ?X) <X)
```

For another such example see the FIT FERM definition in section 8.3.

By means of LOCALs with arbitrary predicates FIT, unlike PROLOG II, can also be used to directly represent conditional term rewriting systems, independent of the kind of condition.

In general, the constraints cI~ in the above FIT rule schema may have the form of arbitrary structures, in addition to that of arithmetical relationships. It is noteworthy that (Bellia et al. 1982) use "equations" that also allow for arbitrary structure constraints, with the syntax

structure0,c1,c2,...,cK <-- structure1,...,structureN

where k≥0 and N≥0 [i.e. the constraints and the body may be empty]. For their fixed-point semantics, however, they move the constraints to the body structures to obtain ordinary Horn clauses of the form

structure0 <-- structure1,...,structureN,c1,c2,...,cK

Unfortunately, since the evaluation order inside pure Horn clause bodies is not determined, the two forms are not equivalent in general [consider a non-terminating cI and a failing structureJ], but only under a special well-formedness condition. Ironically it happens to be the case that with the impure Horn clauses of most PROLOG implementations, the priority of the constraint conjuncts over the ordinary body conjuncts in the original equation may be expressed as

structure0 :- c1, c2, ..., cK, structure1, ..., structureN.

because now the constraints happen to be evaluated before the proper body part; however, now there is also an unwanted left-to-right order inside c1, c2, ..., cK and inside structure1, ..., structureN. Instead of relying on the hazards of the evaluation order, in FIT we use the LOCAL form introduced previously, which always evaluates the constraints [the imposition to the left of ":"] first, without ordering the evaluation inside the constraints or the structures.

Constraints may just be the first step in the replacement of simple conjunctive relation calls in Horn rule bodies by arbitrary functional program bodies. As further extensions of Horn logic the "somewhat complicated" macros in ESP (Chikayama 1983) or the recent proposals in (Kowalski 1983) ["It has proved necessary to extend Horn clause programming in various ways"] may be mentioned. Instead of reworking the semantics of PROLOG with each such new generalization of the original Horn clause formalism, it might be preferable to use general functional rule bodies from the very start, as done in FIT to formalize the semantics of functionally representable rules of PLANNER-like languages.

6.3.2 The COMFO Representation of Constraints -

At this point readers familiar with FIT may wonder whether constraints can somehow be brought to the pattern side (Hussmann 1983). And indeed, according to FIT's general philosophy of performing non-trivial computation in the invocation adapter instead of in the body, another method for representing constrained clauses is to move the constraints from the body to the head, as follows.

A fact with a constraints body of the form

```
({>,v}(r p1 ... pM) c1 c2 ... cK)
```

by means of the COMPOSE expressions introduced in section 5.2.3, can first be rewritten as

```
(r (COMPOSE (TRAFO #ID c1^ c2^ ... cK^) p1 ... pM))
```

Also, a rule with a LOCAL constraints body of the form

```
({>,v}(r p1 ... pM) (LOCAL c1 c2 ... cK : s1 ... sN))
```

by means of these COMPOSE expressions, can first be rewritten as

```
({>,v}(r (COMPOSE (TRAFO #ID c1^ c2^ ... cK^) p1 ... pM)) s1 ... sN)
```

Each cI^ is obtained from cI by replacing "<"-occurrences by "^"-occurrences and by omitting possible "u"-prefixes. The "<"/"^"-replacement accounts for the fact that the constraints now operate on variables global across the invocation adapter. The "u"-omission becomes possible because in invocation computations jF failures are automatically treated as jU failures.

For example, the DIFFCHAIN fact of subsection 6.3.1 in this way becomes

```
(DIFFCHAIN (COMPOSE (TRAFO #ID (NEQ ^X ^Y) (NEQ ^Y ^Z)) ?X ?Y ?Z))
```

And the first two EUCLID rules in this way become

```
(v(EUCLID (COMPOSE (TRAFO #ID (GREATERP ^X ^Y)) ?X ?Y))
 (EUCLID (DIFFERENCE <X <Y) <Y))
(v(EUCLID (COMPOSE (TRAFO #ID (GREATERP ^Y ^X)) ?X ?Y))
 (EUCLID <X (DIFFERENCE <Y <X)))
```

Then, with the help of the COMFO abbreviation introduced in section 5.2.3, the COMPOSE forms can be shortened to

```
(r (COMFO p1 ... pM : c1^ c2^ ... cK^))
```

and

```
({>,v}(r (COMFO p1 ... pM : c1^ c2^ ... cK^)) s1 ... sN)
```

For example, the DIFFCHAIN fact is shortened to

```
(DIFFCHAIN (COMFO ?X ?Y ?Z : (NEQ ^X ^Y) (NEQ ^Y ^Z)))
```

And the EUCLID rules are shortened to

```
(v(EUCLID (COMFO ?X ?Y : (GREATERP ^X ^Y)))
 (EUCLID (DIFFERENCE <X <Y) <Y))
(v(EUCLID (COMFO ?X ?Y : (GREATERP ^Y ^X)))
 (EUCLID <X (DIFFERENCE <Y <X)))
```

Note that the COMFO's pattern imposition is the clause's original invocation pattern without the function name and without parentheses

[i.e. the imposition of its CDR]. Of course, this large scope of the COMFO pattern is only necessary if the constraints actually act over variables which are maximally separated from one another; this happens to be the case in the DIFFCHAIN and EUCLID examples. In all other cases the scope of the COMFO pattern can be reduced, possibly by breaking the COMFO expression into several smaller COMFO expressions. For example, the rule

```
(>(FOO ?A ?B ?C ?D ?E) (LOCAL (GREATERP <B <C) (LESSP <D <E) : ...))
```

by the general COMFO transformation becomes

```
(>(FOO (COMFO ?A ?B ?C ?D ?E : (GREATERP ^B ^C) (LESSP ^D ^E))) ...)
```

and by breaking the COMFO down as far as possible becomes

```
(>(FOO ?A
       (COMFO ?B ?C : (GREATERP ^B ^C))
       (COMFO ?D ?E : (LESSP ^D ^E)))
 ...)
```

Often, however, the COMPOSE form can also be simplified without the help of COMFO, using specific properties of functions. In the EUCLID example, (TRAFO #ID (GREATERP ^X ^Y)) can be replaced directly by GREATERP and (TRAFO #ID (GREATERP ^Y ^X)) can be replaced by its inverse, LESSP. This renders the first two EUCLID rules in their maximally concise form:

```
(v(EUCLID (COMPOSE GREATERP ?X ?Y))
 (EUCLID (DIFFERENCE <X <Y) <Y))
(v(EUCLID (COMPOSE LESSP ?X ?Y))
 (EUCLID <X (DIFFERENCE <Y <X)))
```

6.4 Clause Ordering

A PROLOG data base is an ordered set of clauses, i.e. it has the form

```
clause1.
clause2.
  ...
clauseI.
  ...
clauseZ.
```

where the order of the indices 1, 2, ..., I, ..., Z is relevant and each clauseI is a PROLOG fact or rule. A PROLOG question over that data base uses the first matching clauseF [with the smallest index F] and only on its failure considers the textually consecutive clauses.

A corresponding FIT data base is an unordered set of clauses, i.e. it has the form

```
clausep1~
clausep2~
  ...
clausepI~.
  ...
clausepZ~
```

where p1, p2, ..., pI, ..., pZ is any permutation of 1, 2, ...,I, ..., Z and each clausepI~ is a FIT fact or rule. A FIT question over that data base uses the most specific matching clauseS~ [independent of its index S] or the subset of equally maximum specific matching clauses clauseS1, ..., clauseSk and only on its or their failure considers the nextmost specific clauses.

These PROLOG and FIT data bases lead to equivalent behaviors only in the following two cases.

1. PROLOG's clause ordering and FIT's specificity ordering are immaterial. An example is the two-clause PROLOG data base

   ```
   human(socrates).
   mortal(X) :- human(X).
   ```

   which in FIT becomes

   ```
   (HUMAN SOCRATES)
   (>(MORTAL ?X) (HUMAN <X))
   ```

   The orderings are immaterial here because in PROLOG no possible request is matched by both clauses and also in FIT no PROLOG-like request is matched by both clauses [we exclude here non-PROLOG-like FIT requests such as (|?WHATIS SOCRATES) that would be matched by both clauses]. Both the PROLOG request

   ```
   ?- mortal(WHO).
   ```

   and the FIT request

   ```
   (MORTAL |?WHO)
   ```

   would yield the correct 'Socrates' binding of WHO.

2. PROLOG's clause ordering coincides with FIT's specificity ordering. An example is the three-clause PROLOG data base

   ```
   inhabit(whale,sea).
   inhabit(X,land) :- mammal(X).
   mammal(whale).
   ```

   which in FIT can be written as

   ```
   (INHABIT WHALE SEA)
   (>(INHABIT ?X LAND) (MAMMAL <X))
   (MAMMAL WHALE)
   ```

   The orderings coincide here because in PROLOG the 'whale' fact textually precedes the 'mammal' rule and in FIT the 'whale'

fact is more specific than the 'mammal' rule. Therefore both the PROLOG request

```
?- inhabit(whale,WHAT).
```

and the FIT request

```
(INHABIT WHALE |?WHAT)
```

would yield the correct 'sea' binding of WHAT. However, in the 'permuted' data bases

```
inhabit(X,land) :- mammal(X).
inhabit(whale,sea).
mammal(whale).
```

and

```
(>(INHABIT ?X LAND) (MAMMAL <X))
(INHABIT WHALE SEA)
(MAMMAL WHALE)
```

the textual and specificity orderings no longer coincide because the former has changed and the latter has remained the same. Therefore only FIT would still yield the correct 'sea' binding in the 'whale' request, whereas PROLOG would yield an incorrect 'land' binding.

The dependence on textual order in PROLOG and the independence of that order in FIT accounts for a greater modularity of the latter language. In PROLOG, when adding a new clause the current data base has to be examined carefully to ensure that the clause is inserted at the correct textual position [not to speak of the difficulty of how to perform such inserts using PROLOG's assert/retract primitives, once it is clear where to do it]. In FIT, however, the current data base need not be examined at all; only the specificity of the new clause matters, and this is an intrinsic property of the clause itself. FIT's higher modularity also simplifies automatic addition of clauses, which is necessary for knowledge acquisition by AI systems.

6.5 Cut, SECURE, and EXCLUSIVE

6.5.1 Cut Contrasted with SECURE -

The PROLOG cut operator is not available in FIT because of the well-known problems with this imperative programming construct [(VanEmden 1980), (Clocksin & Mellish 1981)]. However, FIT provides a functional SECURE operator which is comparable to the rule-choice-confirming use of cut as described below.

Let clause! abbreviate either a PROLOG rule of the form [which we'll call 'intermediate cut']

```
structure0 :- structure1, ..., !, ..., structureN.
```

where one cut operator "!" occurs somewhere between the body requests ['intermediate'] or a PROLOG rule of one of the distinguished forms [which we'll call, respectively, 'initial cut' and 'final cut']

structure0 :- !, structure1, ..., structureN.

and

structure0 :- structure1, ..., structureN, !.

which activate the cut immediately after a successful invocation match ['initial'] and only after a successful evaluation of the entire body ['final'], respectively; a fact of the form structure0. through cut becomes a rule of the form structure0 :- !. [regarded as 'initial', though coextensive with 'final'].

Let \clause abbreviate a FIT fact of the form

\structure0

or a FIT rule of the form

({>,v}\structure0 structure1 ... structureN)

where the SECURE operator "\" marks the invocation pattern and is always activated immediately after a successful invocation match [thus SECURE is always 'initial'].

If in a PROLOG data base

```
clause1.
clause2.
  ...
clauseM!.
  ...
clauseZ.
```

some clauseM is "!" [cut] marked and in a corresponding FIT data base

```
clausep1~
clausep2~
  ...
\clauseM~
  ...
clausepZ~
```

a corresponding clauseM~ is "\" [SECURE] marked, then a request matched by the marked clause is processed thus:

In PROLOG clauseM is only applied if none of the clauses clause1, ..., clauseM-1 also matches the request. Otherwise, clauseM would only be applied on failure of all these preceding matching ["!"-less] clauses. Once applied, the cut mark "!" of clauseM makes all possibly matching clauses clauseM+1, ..., clauseZ inapplicable for that request. Thus, if clauseM should fail [this can happen if clauseM has the non-final-cut form structure0 :- ..., !, structureC, ..., structureN. and one of structureC, ..., structureN fails] the entire request fails. Similarly, if later requests conjoined with the clauseM-using request fail [this

can happen even if clauseM has the final-cut form structure0 :- structure1, ..., structureN, !.], this request can produce no further alternatives.

In FIT the SECURE mark "\" of clauseM prioritizes it such that it is applied independently of other ["\"-less] clauses with possibly higher specificity that may also match the request. When applied, clauseM doesn't make other ["\"-less] clauses inapplicable but only deprioritized for that request. Thus, if clauseM should fail other clauses may still cause the request to succeed. Similarly, if later requests conjoined with the clauseM-using request fail, other clauses for this request may still produce further alternatives. If several clauses are SECURE-marked all of them are prioritized against all other ["\"-less] clauses. For the 'fine prioritization' inside the set of SECURE clauses their specificity is used [if no single SECURE clause is maximally specific an entire BREADTH is prioritized].

Let us consider two simple examples of CUT and SECURE uses.

In the previous 'whale' example one might wish to make the general 'inhabit' rule inapplicable if the specific 'whale' fact matches. In PROLOG this may be done by marking that fact by a cut:

```
inhabit(whale,sea) :- !.
inhabit(X,land) :- mammal(X).
mammal(whale).
```

With FIT's SECURE this cannot be done because marking the 'whale' fact in this way, yielding

```
\(INHABIT WHALE SEA)
(>(INHABIT ?X LAND) (MAMMAL <X))
(MAMMAL WHALE)
```

wouldn't change anything, as the marked fact is more specific than the 'inhabit' rule in any case [but see subsection 6.5.2].

As another example consider the PROLOG data base

```
knows(john,mary).
knows(X,president).
```

and its FIT counterpart

```
(KNOWS JOHN MARY)
(KNOWS ?X PRESIDENT)
```

A PROLOG requests like

```
?- knows(john,WHOM).
```

first binds WHOM to mary and then to president because of the textual ordering. A corresponding FIT request

```
(KNOWS JOHN |?WHOM)
```

first binds WHOM to MARY and then to PRESIDENT because of the specificity ordering. Now, if we want to reverse the order of these

answers, i.e. 'privileging' the President, in PROLOG we have to reorder the data base, yielding

```
knows(X,president).
knows(john,mary).
```

In FIT the same effect is obtained by marking the 'President' fact as SECURE, yielding

```
(KNOWS JOHN MARY)
\(KNOWS ?X PRESIDENT)
```

In the PROLOG data base a corresponding cut mark as in

```
knows(john,mary).
knows(X,president) :- !.
```

wouldn't change anything because the 'President' fact is still only reached after the 'Mary' fact. On the other hand, in PROLOG a combination of reordering the data base and cut, as in

```
knows(X,president) :- !.
knows(john,mary).
```

would allow the knows(john,WHOM) request to succeed only once, binding WHOM to president and forgetting about mary. A corresponding combination of reordering and SECURE in the FIT data base, as in

```
\(KNOWS ?X PRESIDENT)
(KNOWS JOHN MARY)
```

would, of course, still allow (KNOWS JOHN |?WHOM) to succeed twice, first with the 'President' and then with the 'Mary' binding [but see subsection 6.5.2].

More sophisticated examples of unrestricted cut and of SECURE may be found in section 7 and a further discussion of SECURE in (Boley 1983).

6.5.2 Cut Restricted to EXCLUSIVE -

In FIT the prioritization of SECUREd clauses is formalized semantically by putting the 'activation record' of a prioritized clause into the first argument position of a DEPTH expression and putting those of other matching clauses into later DEPTH positions. In a formalization of [initial] cut's semantics of making non-prioritized clauses inapplicable, only the activation record of the prioritized one would be kept and the other ones could be thrown away ['abandoned']. [FIT's FINALIZE primitive, a functional version of MICRO-PLANNER's, selects the first successfully evaluated DEPTH element and could thus be used to formalize the 'abandon' semantics of final cuts, not of the initial cuts to be discussed here.] An EXCLUSIVE SECURE version could then be introduced for obtaining a cut-like rigid control in situations where a normal SECURE would seem to be too permissive.

In our opinion the PROLOG use of a cut operator makes programs hard to read mainly because it relies on the textual data base order. Thus one step toward the solution of the cut problem would be the disentangling of the 'abandon' semantics and the 'textual order' semantics. Now, in FIT we don't use 'textual order' semantics but a 'specificity order' semantics modifiable by the SECURE operator. On this basis we could introduce an initial-cut-like EXCLUSIVE operator [also abbreviated with "!"] usable in isolation as in !clauseM~, abandoning less specific and other equally specific ["\"-less] clauses, or together with the SECURE operator as in !\clauseM~, abandoning all other matching clauses. We would thus have separated the abandonment information from the ordering information. If several equally prioritized matching clauses are EXCLUSIVE-marked, only one of them would have to be kept and all others could be abandoned. [EXCLUSIVE, unlike SECURE, is not yet implemented in FIT-1!]

In the 'inhabit' example, all the FIT data bases

```
!(INHABIT WHALE SEA)                    !\(INHABIT WHALE SEA)
(>(INHABIT ?X LAND) (MAMMAL <X))        (>(INHABIT ?X LAND) (MAMMAL <X))
(MAMMAL WHALE)                          (MAMMAL WHALE)


(>(INHABIT ?X LAND) (MAMMAL <X))        (>(INHABIT ?X LAND) (MAMMAL <X))
!(INHABIT WHALE SEA)                    !\(INHABIT WHALE SEA)
(MAMMAL WHALE)                          (MAMMAL WHALE)
```

for the 'whale' request would abandon the less specific 'land' rule because this is excluded by the more specific [in the right-hand-side data bases, redundantly SECURE marked] 'whale' fact, i.e. they would act like the PROLOG data base

```
inhabit(whale,sea) :- !.
inhabit(X,land) :- mammal(X).
mammal(whale).
```

In the 'knows' example, both the FIT data bases

```
(KNOWS JOHN MARY)
!(KNOWS ?X PRESIDENT)
```

and

```
!(KNOWS ?X PRESIDENT)
(KNOWS JOHN MARY)
```

wouldn't change the behavior of the unmarked data base because nothing is left to exclude for the less specific 'President' fact, i.e. they would act like the PROLOG data base

```
knows(john,mary).
knows(X,president) :- !.
```

However, both the FIT data bases

```
(KNOWS JOHN MARY)
!\(KNOWS ?X PRESIDENT)
```

and

```
!\(KNOWS ?X PRESIDENT)
(KNOWS JOHN MARY)
```

would permit success only for the prioritized President which excludes Mary, i.e. they would act like the PROLOG data base

```
knows(X,president) :- !.
knows(john,mary).
```

In FIT, EXCLUSIVE and SECURE could be used not only in the definition of predicate functions like KNOWS but also in the definition of general functions like FAC, not possible in PROLOG. For instance, the usual simple factorial definition

```
(>(FAC 0)  1)
(>(FAC ?N) (TIMES <N (FAC (SUB1 <N))))
```

has the disadvantage that [at the bottom of recursions] the call (FAC 0) is matched by both clauses, in pure FIT returning

```
(DEPTH 1 suspension-which-would-diverge-to-negative-infinity)
```

This can be avoided by making the invocation pattern of the second clause disjoint from that of the first, i.e. by exchanging the untyped variable ?N by the typed variable x?POSINT for positive integers:

```
(>(FAC 0)  1)
(>(FAC x?POSINT) (TIMES <POSINT (FAC (SUB1 <POSINT))))
```

Alternatively [saving repeated POSINT checks for each recursive FAC call, redundant for all but the initial and the final call], the first clause could be marked by an EXCLUSIVE operator:

```
(>!(FAC 0)  1)
(>(FAC ?N) (TIMES <N (FAC (SUB1 <N))))
```

Since the pattern (FAC 0) is more specific than the pattern (FAC ?N) no SECURE operator is needed here. If, instead, we used equally specific and disjoint invocation patterns like (FAC x?ZEROP) and (FAC x?POSINT), no EXCLUSIVE operator would be needed and the SECURE operator would be reduced to a matter of style and efficiency:

```
(>\(FAC x?ZEROP)  1)
(>(FAC x?POSINT) (TIMES <POSINT (FAC (SUB1 <POSINT))))
```

Finally, if we used equally specific and non-disjoint invocation patterns like (FAC x?ZEROP) and (FAC x?NUMBERP) both EXCLUSIVE and SECURE would be called for:

```
(>!\(FAC x?ZEROP)  1)
(>(FAC x?NUMBERP) (TIMES <NUMBERP (FAC (SUB1 <NUMBERP))))
```

The examples illustrate the following property of the EXCLUSIVE operator. In addition to not relying on the textual data base order between rules, EXCLUSIVE is higher-level than cut because it doesn't rely on ordering inside rule bodies. Just as the WHILE statement

corresponds to a very restricted form of goto, the EXCLUSIVE operator corresponds to a very restricted form of cut, characterized by the following properties:

1. Only one cut is permitted for each clause [the 'single cut' property].

2. This cut can only occur in a fixed position, namely immediately after the invocation match [the 'initial cut' property].

This means that EXCLUSIVE, as well as SECURE, applies to a clause in its entirety, in contrast to unrestricted cuts, which may be sprinkled throughout clause bodies. Therefore, understanding a FIT clause involves only checking whether it is EXCLUSIVE [SECURE] at all, rather than how often or where it has some such property, as required for understanding a PROLOG clause. This is in parallel with WHILE, which, unlike unrestricted gotos, applies to a program block in its entirety, heightening its understandability in a similar manner.

Another use of EXCLUSIVE in the following subsection 6.5.3 will exhibit further advantages of the initial-cut property.

6.5.3 From Guarded Commands to Constrained EXCLUSIVE Rules -

The combination of EXCLUSIVE clauses and constrained clauses [section 6.3] yields an interesting kind of rule, which may be seen as a functional version of "guarded commands" (Dijkstra 1975) and "productions" (Newell 1973). This combination is enabled by the COMFO constraints method, introduced in section 6.3.2. Like every other rule, a COMFO constrained rule can be marked by an EXCLUSIVE operator, obtaining

```
({>,v}!(r (COMFO p1 ... pM : c1 c2 ... cK)) s1 ... sN)
```

On invocation, this rule fits its head (r (COMFO p1 ... pM : c1 c2 ... cK)) to the expression to be evaluated, thereby checking the constraints c1, c2, ..., cK over the variables among p1, ..., pM. If this constraint-checking invocation fitting succeeds, the EXCLUSIVE operator causes other possibly successful rules to be abandoned.

Then a guarded command of the form

```
guard -> statement1;...;statementN
```

can be represented as the rule [the tilde denotes a transformation from Dijkstra's ALGOL-like syntax to FIT syntax]

```
({>,v}!(D (COMFO : guard~)) statement1~ ... statementN~)
```

The rule uses a dummy name r=D and a COMFO expression with an empty pattern [i.e. M=0; in that case equivalent to a TRAFO] and a body consisting of a single constraint [i.e. K=1], c1=guard~, operating over global variables.

Dijkstra's guarded-command-based alternative construct

```
if guarded-command1
[] guarded-command2
   ...
[] guarded-commandZ
fi
```

can be rewritten in FIT as [the tilde transforms guarded commands as demonstrated above]

```
guarded-command1~
guarded-command2~
   ...
guarded-commandZ~
```

That is, the isolated guarded-command rules are simply written into a [possibly LOCAL] FIT data base. Note that while "a guarded command by itself is not a statement" (Dijkstra 1975), its FIT representation is a rule, usable by itself or as part of a larger construct.

For example, Dijkstra's "program that for fixed x and y assigns to m the maximum value of x and y",

```
if x ≥ y -> m := x
[] y ≥ x -> m := y
fi
```

in FIT can be rewritten as

```
(>!(D (COMFO : (GE <X <Y))) (>M <X))
(>!(D (COMFO : (GE <Y <X))) (>M <Y))
```

After (>X 3) and (>Y 5) this can be called by (D), which sets M to 5.

Deviating from Dijkstra's imperative global-state-oriented programming style, a functional method of transcribing a guarded-command-based construct consists of the introduction of a new function for it such that the imported global variables of the construct become the arguments of the function and the exported global variables are replaced by the function's returned values. A guarded command of such a construct is transcribed using the function's name instead of D, a non-trivial COMFO expression with the guard operating on the function's arguments arg1, ..., argM, and functional expressions as statements [here the tilde denotes a functional transformation]:

```
({>,v}!(name (COMFO arg1 ... argM : guard~))
       statement1~ ... statementN~)
```

Thus the alternative construct functionally becomes [using the abbreviations args = arg1 ... argM and, for 1≤j≤Z, statementsj~ = statementj,1~ ... statementj,Nj~]

```
({>,v}!(name (COMFO args : guard1~)) statements1~)
({>,v}!(name (COMFO args : guard2~)) statements2~)
  ...
({>,v}!(name (COMFO args : guardZ~)) statementsZ~)
```

For example, the maximum program can be represented as a function named MAX with two arguments X and Y and one returned value:

```
(>!(MAX (COMFO ?X ?Y : (GE ^X ^Y))) <X)
(>!(MAX (COMFO ?X ?Y : (GE ^Y ^X))) <Y)
```

This can be called by (MAX 3 5), returning 5.

Dijkstra's guarded-command-based repetitive construct could be reformulated into FIT similarly, additionally using tail-recursion for representing iteration. For example, Dijkstra's "program for the greatest common divisor of two positive numbers",

```
x := X; y := Y;
do x > y -> x := x - y
[] y > x -> y := y - x
od
```

can be functionally rewritten in FIT as

```
(v!(EUCLID (COMFO ?X ?Y : (GREATERP ^X ^Y)))
 (EUCLID (DIFFERENCE <X <Y) <Y))
(v!(EUCLID (COMFO ?X ?Y : (GREATERP ^Y ^X)))
 (EUCLID <X (DIFFERENCE <Y <X)))
(>(EUCLID ?X ?X) <X)
```

However, this doesn't change anything in the EXCLUSIVE-less COMFO version of subsection 6.3.2, because, after the constraints check, always exactly one rule remains, so that there is nothing left to exclude for this single rule. This shows that the implicit 'abandon semantics' [cf. subsection 6.5.2] of Dijkstra's guarded commands is not required in his principal EUCLID example [nor in other programs whose guards are disjoint rather than overlapping as in the maximum program]. Since it is clear that the cut operator should not be used without need, the same should hold for its restricted EXCLUSIVE form, so that the earlier EUCLID version of subsection 6.3.2 appears preferable to the present one, derived from guarded commands. The non-abandoning, logically 'purer' version cannot be specified with guarded commands, because of their built-in abandon semantics.

A relational transcription method for guarded-command-based constructs, intermediate between the imperative and the functional one, can be derived from the functional method, provided that the exported variables become result variables of the relation. A guarded command of such a construct looks like the functional one except that it uses a relation name [by convention having a "-P" suffix], additional result variables res1, ..., resL, and relational expressions as statements [here the tilde denotes a relational transformation]:

```
({>,v}!(name-P (COMFO arg1 ... argM : guard~) res1 ... resL)
       statement1~ ... statementN~)
```

Thus the alternative construct relationally becomes [using the previous abbreviations together with ress = res1 ... resL]

```
({>,v}!(name-P (COMFO args : guard1~) ress) statements1~)
({>,v}!(name-P (COMFO args : guard2~) ress) statements2~)
  ...
({>,v}!(name-P (COMFO args : guardZ~) ress) statementsZ~)
```

Often this raw relational transcription can be simplified.

For example, the maximum program can be represented as a relation named MAXP [a short form of MAX-P] with two input variables X and Y and one result variable M:

```
(>!(MAXP (COMFO ?X ?Y : (GE ^X ^Y)) ?M) (EQ <M <X))
(>!(MAXP (COMFO ?X ?Y : (GE ^Y ^X)) ?M) (EQ <M <Y))
```

Of course, the statements (EQ <M <X) and (EQ <M <Y) can be omitted here by replacing the ?M arguments directly by ?X and ?Y, respectively:

```
(>!(MAXP (COMFO ?X ?Y : (GE ^X ^Y)) ?X) )
(>!(MAXP (COMFO ?X ?Y : (GE ^Y ^X)) ?Y) )
```

Then, if, as in the above maximum program, the statement part of rules becomes empty through the relational transcription, these transformation rules [transformers] can be further simplified to adaptation rules [adapters]:

```
!(MAXP (COMFO ?X ?Y : (GE ^X ^Y)) ?X)
!(MAXP (COMFO ?X ?Y : (GE ^Y ^X)) ?Y)
```

The relational versions can be called by (MAXP 3 5 |?ANS), binding ANS to 5.

In (Kowalski 1979) a relational formulation of Dijkstra's maximum program is discussed as an example of "don't care" non-determinism1, characteristic for guarded commands and usable for a form of intelligent backtracking; however, it is not stated that such a "don't care" specification requires an extra-logical feature equivalent to PROLOG's cut operator. This is demonstrated in the following PROLOG-like version of the maximum program which, like the previous versions, presupposes no clause order:

```
maxp(X,Y,X) :- X >= Y, !.
maxp(X,Y,Y) :- Y >= X, !.
```

[If interpreted as ordinary PROLOG, with clause order, the second cut would be redundant.]

Notice that this must take the form of PROLOG rules [transformers], even though no goals follow after the guard evaluation or constraints check. PROLOG facts [adapters] cannot be used, since the constraints are themselves represented as goals.

In (Clark & Gregory 1981) the term "committed" instead of "don't care" non-determinism is used and the cut operator between guards and other goals is called "clause bar" [written as "|"]. Finally, in CONCURRENT PROLOG (Shapiro & Takeuchi 1983) relational guarded commands are called "guarded-clauses" and the "|" cut is adopted under the name "commit operator".

FIT's constrained EXCLUSIVE rules are preferable to CONCURRENT PROLOG's guarded-clauses for the following reasons:

1. Although the commit operator has the single-cut property it doesn't have the initial-cut property of the EXCLUSIVE operator; it thus misses the advantages of initial cuts:

   1. The left-right division [the 'arrow'] of transformation rules coincides syntactically with the initial cut [both are thus joinable to a 'cut arrow', as used implicitly in Dijkstra's guarded commands]; in this way the cut is limited to a position in the transformer which is special in any case, so that readability is improved.

   2. It is advantageous to consider constraints checks as generalized pattern-directed invocation, i.e. carrying them out as part of the 'left-hand-side' invocation fitting of a rule [mirroring the left-hand-side evaluation of Dijkstra's guards]: If the rule constrained in this way has an initial cut, a completion of the invocation fitting means a real completion of the rule selection, in contrast to the preliminary completion permitted by a non-initial cut, which can be continuously revised until the body evaluation reaches the cut.

   3. Only initial cuts preserve the left-right symmetry of rules, i.e. permit 'cut-symmetrical' rules; this becomes important if the arrow direction is reversed to switch from backward reasoning to forward reasoning [exploiting the multiple readability of Horn clauses through "top-down"/"bottom-up inference" (Kowalski 1979), rather than through "invertibility" (Kowalski 1983)]: While top-down/bottom-up reversals make no sense with non-initial cuts, they can be meaningful with initial cuts.

2. EXCLUSIVE-marked transformation rules can be simplified to EXCLUSIVE-marked adaptation rules if the constraints are checked during invocation fitting and if there are no other goals [cf. the last MAXP version].

3. Constrained rules are more general than guarded-clauses in that they can not only be used relationally for defining predicates [cf. MAXP] but also functionally for defining general functions [cf. MAX].

7 LIST AND SET PROCESSING

We now compare list and set processing in FIT and PROLOG. Since sets will be represented as lists without duplicate elements, the term 'list processing' in the following will encompass set processing. As in PROLOG in FIT we will define relations rather than functions for list processing. In this way the comparison between FIT and PROLOG becomes easier than via a translation of FIT's list-processing functions to

PROLOG's list-processing relations. At the same time it shows how FIT's adapters [cf. section 5.2] can make relational programming, PROLOG's domain of expertise, more concise than even PROLOG's transformers [Horn clauses] can. In such adapters frequent use will be made of compositions of the form ABo?var, which give some subexpression a name var, usable at another place, and then erase this subexpression; using COMFO expressions [cf. section 5.2.3], this could also be formulated as (COMFO ?var).

To avoid confusion between list-function names coined by LISP [also used in FIT] and corresponding relation names in FIT we will append the letter "P" to every relation [predicate] name which PROLOG borrowed from the name of a general function in LISP.

As usual, the FIT examples of this section have been tested in FIT-1. However, only the pure predicate use of the definitions is completely implemented in FIT-1; definition uses with request variables are not yet operational in full generality, because of the restricted unification fitting performed in this current FIT implementation. We won't use the EXCLUSIVE operator for representing initial cuts here, but the reader may easily supply it where desired [cf. section 6].

7.1 Elementary List Processing

For the following comparison we will use the PROLOG examples of chapter 7.5 in (Clocksin & Mellish 1981) and reformulate them in FIT.

Finding the last element of a list: The recursive PROLOG definition

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

can be directly mirrored by a recursive FIT definition using a constant-adapter for the boundary condition and a transformer for the recursive case:

```
(LASTP ?X (?X))
(>(LASTP ?X (ID ?YoLIST)) (LASTP <X <Y))
```

However, the tail-recursive transformer can be replaced by a REVA-adapter [marked by an "r"-prefix that causes the result of the adapter fitting to be re-evaluated], making the FIT definition more concise and free of single-occurrence variables [for details on these adapter concepts see (Boley 1983)]:

```
(LASTP ?X (?X))
r(LASTP ID (AB #ID))
```

Now, since there is no need for the left-to-right processing performed by the above definitions, the two adapters can be collapsed into a single constant-adapter:

```
(LASTP ?X (#ID ?X))
```

This is a most concise, declarative, and pictorial description of the desired last list element. In PROLOG such a very-high-level formulation can only be approximated by a transformer presupposing the definition of append [see below]:

```
last(X,L) :- append(_,[X],L).
```

Checking for consecutive elements: The recursive PROLOG definition

```
nextto(X,Y,[X,Y|_]).
nextto(X,Y,[_|Z]) :- nextto(X,Y,Z).
```

could also be directly mirrored in FIT using a transformer, but let us directly consider the more concise REVA-adapter version:

```
(NEXTTOP ?X ?Y (?X ?Y #ID))
r(NEXTTOP ID ID (AB #ID))
```

Again, without left-to-right commitment these adapters collapse into one constant adapter:

```
(NEXTTOP ?X ?Y (#ID ?X ?Y #ID))
```

And again, this most concise version in PROLOG can only be approximated by a transformer depending on append [see below]:

```
nextto(X,Y,L) :- append(_,[X,Y|_],L).
```

Appending lists: The recursive PROLOG definition

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

by our previous method becomes the FIT adapters

```
(APPENDP () ?L ?L)
r(APPENDP (ABo?X #ID) ID (ABo?X #ID))
```

Through the use of parallel imposition variables this becomes trivialized to

```
(APPENDP (>R) (>S) (>R >S))
```

The "flexibility of append" in PROLOG, which allows (Clocksin & Mellish 1981) to "define several other predicates in terms of it" [cf. last and nextto above], consists of the fact that append can be used to divide a list almost symmetrically into two segments, so that its first argument and a tail of its second argument can be used to simulate two segment variables. However, this is a very indirect and cumbersome way of bi-partitioning lists, not to speak of n-partitionings, that require the analogue of nested append expressions [PROLOG's append relation corresponds to LISP's binary *APPEND function, not to its n-ary APPEND function]. As an example consider the append-based member predicate definition in (Clocksin & Mellish 1981):

```
member(El,List) :- append(_,[El|_],List).
```

The append call uses an anonymous variable as its first argument, which

acts as an arbitrary left segment. As its second argument append uses a list with the membership candidate El as its head and another anonymous variable as its tail, the latter acting as an arbitrary right segment. So the two segment contexts around El are not symmetric syntactically, because the left one appears as a top-level argument of append, whereas the right one is embedded in the tail of an append argument. This occludes the complete semantic symmetry of the two segment contexts of an element occurring somewhere in a list.

In FIT the semantic symmetry is made visible syntactically, using direct notations for segments, here anonymous #ID segments:

```
(MEMBER ?EL (#ID ?EL #ID))
```

[Incidentally, which notation to use for anonymous segments is not at issue here. The three characters "..." as, e.g., used in LISP70 may at first seem more natural than the three characters "#ID", but the latter can be semantically decomposed into the very natural "#" and "ID" operators.]

Reversing a list: PROLOG's efficient reverse definition

```
rev2(L1,L2) :- revzap(L1,[],L2).

revzap([X|L],L2,L3) :- revzap(L,[X|L2],L3).
revzap([],L,L).
```

in FIT becomes

```
(>(REV2P ?L1 ?L2) (REVZAP <L1 () <L2))

r(REVZAP (ABo?X #ID) ((TRAFO : ^X) #ID) ID)
(REVZAP () ?L ?L)
```

The transformer initializing REVZAP can be made a REVA-adapter by naming REVZAP also REV2P and using (TRAFO : ()) to generate the empty list from the empty imposition:

```
r(REV2P ID (TRAFO : ()) ID)

r(REV2P (ABo?X #ID) ((TRAFO : ^X) #ID) ID)
(REV2P () ?L ?L)
```

The PROLOG rev2 definition with its unnecessarily globally accessible revzap subordinates illustrates a major shortcoming of that language, which may even disqualify it as an implementation language for large software engineering projects: Although PROLOG was developed in the same time period as abstract data types, and logical ADT specification appears trivial (Bibel 1983) as well, PROLOG has no information hiding and modularization facilities. There are now proposals to augment PROLOG with ADTs (Nakashima & Suzuki 1983) and module concepts [(Bendl et al. 1980), (Clark et al. 1982), (Eggert & Schorre 1982), (Chikayama 1983)], but the lack of an obvious 'winner' among these unrelated candidates seems to indicate that modules are hard to integrate with PROLOG's base components. For example, (Eggert & Schorre 1982) reformulate the rev2 definition as the following module exporting the name reverse:

```
module(reverse).
  r(nil,L,L).
  r(Y.L1,L2,R) <- r(L1,Y.L2,R).

  reverse(L,R) <- r(L,nil,R).
endModule.
```

However, like their function extension [cf. section 3], this is implemented through preprocessing, which surely is not the right approach for realizing a concept as basic as modules. In FIT the available LOCAL [data base] primitive [cf. section 2.2] can be used for defining modules [R happens to act both as a variable and a relation name]:

```
(>(REVERSEP ?L ?R)
 (LOCAL ((R NIL ?L ?L)
         (>(R (?Y ?L1oLIST) ?L2 ?R) (R <L1 (CONS <Y <L2) <R)))
        (R <L NIL <R)))
```

While in Eggert/Schorre's modules the unit of export is relation names, in FIT it is relation calls. Therefore in the former module the names r and reverse must be carefully distinguished, whereas in the latter there would be no problem if the names R and REVERSEP were joined to REVERSEP:

```
(>(REVERSEP ?L ?R)
 (LOCAL ((REVERSEP NIL ?L ?L)
         (>(REVERSEP (?Y ?L1oLIST) ?L2 ?R)
          (REVERSEP <L1 (CONS <Y <L2) <R)))
        (REVERSEP <L NIL <R)))
```

The LOCALized REVERSEP definitions are just as invisible externally as were the LOCALized R definitions. Therefore, externally still only calls like (REVERSEP '(1 2 3) |?ANS) are possible, not calls like (REVERSEP '(1 2 3) NIL |?ANS).

Deleting one element: The recursive PROLOG definition

```
efface(A,[A|L],L) :- !.
efface(A,[B|L],[B|M]) :- efface(A,L,M).
```

can be directly translated to the FIT definition [the SECURE operator "\" prioritizes the less specific first definition]

```
\(EFFACE ?A (?A ?LoLIST) ?L)
r(EFFACE ID (ABo?B #ID) (ABo?B #ID))
```

If an arbitrary A-element rather than the left-most occurrence is to be removed the adapters can be collapsed into

```
(EFFACE ?A (>L ?A >R) (>L >R))
```

PROLOG's additional clause for recognizing when the second argument becomes reduced to the empty list,

```
efface(_,[],[]).
```

in FIT becomes

```
(EFFACE ID () ())
```

Deleting all occurrences of an element: The PROLOG definition

```
delete(_,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L1],[Y|L2]) :- delete(X,L1,L2).
```

in FIT becomes

```
r(DELETEP ?X (#ID ABo?X #ID) ID)
(DELETEP ID ?L ?L)
```

Substitution: The PROLOG definition

```
subst(_,[],_,[]).
subst(X,[X|L],A,[A|M]) :- !, subst(X,L,A,M).
subst(X,[Y|L],A,[Y|M]) :- subst(X,L,A,M).
```

is erroneous because it accepts, for instance, the list [1,2] as the input and the result of substituting a new element [unequal to 1], say 0, for the old element 1: The relation call subst(1,[1,2],0,[1,2]) is not matched by the second clause, since A cannot be both 0 and 1; unfortunately, however, it is matched by the third clause, since both lists happen to start with the same element Y=1; thus an illegitimate recursion subst(1,[2],0,[2]) takes place, which via subst(1,[],0,[]) yields an incorrect 'yes' answer. The subst definition could be corrected using "\=" in the third clause to ensure that the first list element is not the old element [since this is the last subst clause no cut is necessary after the 'constraint check' X\=Y]:

```
subst(X,[Y|L],A,[Y|M]) :- X\=Y, subst(X,L,A,M).
```

In FIT the definition becomes [the "\" prioritizations guarantee that the last definition is used only when no other one applies]

```
r\(SUBSTP ?X (#ID ABo?X #ID) ?A (#ID ABo?A #ID))
(>\(SUBSTP ?X (#ID ?X #ID) ID ID) jF)
(SUBSTP ID ?L ID ?L)
```

Here, the critical example, (SUBSTP 1 '(1 2) 0 '(1 2)) is not matched by the more specific first clause, an adapter generalizing PROLOG's second clause; therefore it is matched by the less specific second clause, which correctly yields jF.

Perhaps the error in PROLOG originated from formulating subst too closely in analogy to delete ["this is quite similar to delete, except instead of deleting a desired element, we substitute some other element in its place" (Clocksin & Mellish 1981)]: The second delete clause is only inapplicable when the old element is not the first element of the argument list, whereas the second subst clause is also inapplicable in the 'unusual' case that the new element is not the first element of the result list. The case is 'unusual' at least in the view of functional programming where result lists are returned values rather than arguments; posssibly, (Clocksin & Mellish 1981) had only LISP's natural functional subst use in mind, not the strange but basic relational subst use of checking whether 'four given s-expressions are in a substitution relation'. This will be further discussed in the context

```
(EFFACE ID () ())
```

Deleting all occurrences of an element: The PROLOG definition

```
delete(_,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L1],[Y|L2]) :- delete(X,L1,L2).
```

in FIT becomes

```
r(DELETEP ?X (#ID ABo?X #ID) ID)
(DELETEP ID ?L ?L)
```

Substitution: The PROLOG definition

```
subst(_,[],_,[]).
subst(X,[X|L],A,[A|M]) :- !, subst(X,L,A,M).
subst(X,[Y|L],A,[Y|M]) :- subst(X,L,A,M).
```

is erroneous because it accepts, for instance, the list [1,2] as the input and the result of substituting a new element [unequal to 1], say 0, for the old element 1: The relation call subst(1,[1,2],0,[1,2]) is not matched by the second clause, since A cannot be both 0 and 1; unfortunately, however, it is matched by the third clause, since both lists happen to start with the same element Y=1; thus an illegitimate recursion subst(1,[2],0,[2]) takes place, which via subst(1,[],0,[]) yields an incorrect 'yes' answer. The subst definition could be corrected using "\=" in the third clause to ensure that the first list element is not the old element [since this is the last subst clause no cut is necessary after the 'constraint check' X\=Y]:

```
subst(X,[Y|L],A,[Y|M]) :- X\=Y, subst(X,L,A,M).
```

In FIT the definition becomes [the "\" prioritizations guarantee that the last definition is used only when no other one applies]

```
r\(SUBSTP ?X (#ID ABo?X #ID) ?A (#ID ABo?A #ID))
(>\(SUBSTP ?X (#ID ?X #ID) ID ID) jF)
(SUBSTP ID ?L ID ?L)
```

Here, the critical example, (SUBSTP 1 '(1 2) 0 '(1 2)) is not matched by the more specific first clause, an adapter generalizing PROLOG's second clause; therefore it is matched by the less specific second clause, which correctly yields jF.

Perhaps the error in PROLOG originated from formulating subst too closely in analogy to delete ["this is quite similar to delete, except instead of deleting a desired element, we substitute some other element in its place" (Clocksin & Mellish 1981)]: The second delete clause is only inapplicable when the old element is not the first element of the argument list, whereas the second subst clause is also inapplicable in the 'unusual' case that the new element is not the first element of the result list. The case is 'unusual' at least in the view of functional programming where result lists are returned values rather than arguments; posssibly, (Clocksin & Mellish 1981) had only LISP's natural functional subst use in mind, not the strange but basic relational subst use of checking whether 'four given s-expressions are in a substitution relation'. This will be further discussed in the context

of a similar problem with the intersection and union relations in subsection 7.2. In any case, the subst error seems to indicate that relational formulations can make programs as simple as LISP's SUBST function error-prone because of the increased number of arguments and their possible unexpected usage [it is true that some of these errors won't come to the surface as long as nobody uses these relations in a 'strange' manner, but how do you explain to your students that basic 'yes/no' questions without any request variables are 'strange'?].

Sublists: The PROLOG definition

```
sublist([X|L],[X|M]) :- prefix(L,M), !.
sublist(L,[_|M]) :- sublist(L,M).

prefix([],_).
prefix([X|L],[X|M]) :- prefix(L,M).
```

in FIT trivializes to

```
(SUBLIST (?X >L) (#ID ?X >L #ID))
```

The PROLOG definition is cumbersome because its sublist and prefix parts handle overlapping cases, a redundancy which can be seen in the almost identical first sublist clause and second prefix clause [the cut in the former is disputable anyway, because, although it prevents calls from incorrectly falling into the second clause, it also prevents calls like sublist([b,SECOND,THIRD],[a,b,c,d,e,b,e,a]) from finding not only SECOND=c and THIRD=d but also SECOND=e and THIRD=a].

The above definitions don't allow empty sublists, although these are sublists according to a literal interpretation of the definition "list X is a sublist of list Y if every item in X also appears in Y, ..." (Clocksin & Mellish 1981). Thus, the non-emptiness restriction may well be an artifact of PROLOG's task sharing between sublist and prefix. In FIT the removal of the non-emptiness restriction makes the definition even simpler:

```
(SUBLIST (>L) (#ID >L #ID))
```

The structural similarity of this definition and the MEMBER definition suggests another, still simpler definition,

```
(SUBIMP >L (#ID >L #ID))
```

which generalizes MEMBER by just replacing its ?EL occurrences by >L occurrences. Alternatively, SUBIMP can also be regarded as a generalization of NEXTTOP from two to arbitrarily many consecutive elements. For example, (SUBIMP B C D '(A B C D E)) would succeed but (SUBIMP A C D '(A B C D E)) would fail. A definition like SUBIMP is impossible in PROLOG because of the formal imposition argument allowing for a variable number of actual arguments.

Since the FIT adapter definitions directly capture the essence of the list predicate functions involved, semantically similar functions become similar syntactically. Thus, an automatic program understanding system would only have to attempt a unification of, say, the definitions

```
(NEXTTOP ?X ?Y (#ID ?X ?Y #ID)) and
(SUBIMP    >L  (#ID   >L  #ID))
```

to recognize that the former is a special case of the latter because the substitution L=X Y allows NEXTTOP and SUBIMP to become equal. Although PROLOG programs make heavy use of unification, they themselves are not easily unifiable data structures and an automatic recognition of a corresponding relationship between nextto and sublist would involve much more than a simple unification [how often was the relationship found 'by hand'?].

7.2 Manipulating Sets

For the following comparison we will use the PROLOG examples of chapter 7.6 in (Clocksin & Mellish 1981) and reformulate them in FIT. Other than in the case of elementary list operations, there is no general shortening effect through the FIT definitions here. This is partly because we directly define all FIT operations in terms of primitives, whereas PROLOG builds on the member predicate [which could also be done in FIT], and partly because the PROLOG definitions for intersection and union are 'incomplete' in that they don't account for the unorderedness of sets. Although the permutation predicate is the most basic predicate on sets represented as lists [namely set equality] and, generally, sorting is prerequisite to set processing, in (Clocksin & Mellish 1981) this is only discussed in the following chapter, without any connection between the two chapters.

The PROLOG permutation predicate

```
permutation(L,[H|T]) :-
   append(V,[H|U],L),
   append(V,U,W),
   permutation(W,T).
permutation([],[]).
```

in FIT can be shortened to the definition

```
r(PERMUTATION (ABo?Z #ID) (#ID ABo?Z #ID))
(PERMUTATION () ())
```

whose meaning could be paraphrased as "A list is in a permutation relation with another list if the elements of the first list can be removed from left to right, simultaneously removing identical elements somewhere from the second list, so that both lists become empty at the same time."

The member predicate for sets is omitted here because it is the same as that for lists.

The PROLOG subset predicate

```
subset([A|X],Y) :- member(A,Y), subset(X,Y).
subset([],Y).
```

in FIT can be redefined as

```
r(SUBSET (ABo?A #ID) (#ID ABo?A #ID))
(SUBSET () ID)
```

whose meaning could be paraphrased as "A list is in a subset relation with another list if the elements of the first list can be removed from left to right, simultaneously removing identical elements somewhere from the second list, so that the first list becomes empty before or together with the second list."

Notice the similarity of the PERMUTATION and SUBSET definitions in FIT and their crucial syntactical ()/ID difference, which faithfully reflects their semantic difference. No such syntax/semantics correspondence between the PROLOG permutation and subset definitions is perceivable.

The PROLOG disjoint predicate

```
disjoint(X,Y) :- not(( member(Z,X), member(Z,Y) )).
```

in FIT can be redefined 'negatively' as

```
(>\(DISJOINT (#ID ?Z #ID) (#ID ?Z #ID)) jF)
(DISJOINT ID ID)
```

The PROLOG intersection predicate

```
intersection([],X,[]).
intersection([X|R],Y,[X|Z]) :-
   member(X,Y),
   !,
   intersection(R,Y,Z).
intersection([X|R],Y,Z) :- intersection(R,Y,Z).
```

in FIT becomes

```
(INTERSECTIONP () ID ())
r(INTERSECTIONP (ABo?X #ID) (#ID ?X #ID) (#ID ABo?X #ID))
r(INTERSECTIONP (AB #ID) ID ID)
```

The PROLOG union predicate

```
union([],X,X).
union([X|R],Y,Z) :- member(X,Y), !, union(R,Y,Z).
union([X|R],Y,[X|Z]) :- union(R,Y,Z).
```

in FIT becomes

```
(>(UNIONP () ?X ?Y) (PERMUTATION <X <Y))
(>(UNIONP ?X () ?Y) (PERMUTATION <X <Y))
r\(UNIONP (#ID ABo?X #ID) (#ID ABo?X #ID) (#ID ABo?X #ID))
r(UNIONP (ABo?X #ID) (#ID ABo?Y #ID) (#ID ABo?X #ID ABo?Y #ID))
r(UNIONP (ABo?X #ID) (#ID ABo?Y #ID) (#ID ABo?Y #ID ABo?X #ID))
```

Although in (Clocksin & Mellish 1981) one finds the correct set characterization "A set is a collection of elements, rather like a list, but it does not make sense to ask "where" or "how many times" something is an element of a set", the authors don't account for the "where" irrelevance consistently. While the PROLOG set operations

member, subset, and disjoint are insensitive to the order of the elements in lists representing sets, the operations intersection and union are not. For instance, the fact that the intersection of {r,a,p,i,d} and {p,i,c,t,u,r,e} is {r,i,p}, an example given in (Clocksin & Mellish 1981), cannot be verified by the PROLOG intersection program quoted above from the same book. The call intersection([r,a,p,i,d],[p,i,c,t,u,r,e],[r,i,p]) incorrectly prints 'no' because the order in the result set differs from the order in the first argument. A correct 'yes' answer can only be obtained if the list-represented set {r,i,p} is given in the permutation {r,p,i} corresponding to the element order in {r,a,p,i,d}, i.e. by intersection([r,a,p,i,d],[p,i,c,t,u,r,e],[r,p,i]). This problem is caused by the second clause which runs through its first argument [X|R] and its third argument [X|Z] in a synchronized manner, imposing the same order on both arguments. PROLOG's union operation suffers from the same unwanted synchronization in its third clause; there is an additional problem with the first clause, union([],X,X), which forces the two X occurrences to be equal as lists [incl. order], not as sets [this problem can be traced back to the strange elision of permutation from the discussion of sets]. Thus not even the equation {} U {a,b} = {b,a} can be verified because the trivial call union([],[a,b],[b,a]), which may recursively result from calls like union([a],[a,b],[b,a]), incorrectly prints 'no'. To obtain the correct 'yes' answer one must write union([],[a,b],[a,b]) or union([a],[a,b],[a,b]).

The FIT set operations are insensitive to the order of elements in lists, which thus become true set represention. This order insensitivity comes for free by virtue of the inherent parallelism of adapters, with two exceptions [both will be eliminated later]. 1. In the first two UNIONP clauses we cannot use adapters (UNIONP () ?X ?X) and (UNIONP ?X () ?X) but have to use transformers with a PERMUTATION call in their body. 2. The last UNIONP clause r(UNIONP (ABo?X #ID) (#ID ABo?Y #ID) (#ID ABo?Y #ID ABo?X #ID)) is only necessary for permitting reductions like (UNIONP '(A) '(B) '(B A)) => (UNION () () ()), where different elements in the argument sets occur in inverse order in the result set. While the first ordering problem can also be solved in PROLOG by exchanging the fact union([],X,X) by the rule union([],X,Y) :- permutation(X,Y), there seems to be no FIT-like simple addition to the PROLOG definitions that would account for the second ordering problem.

Perhaps this problem with the PROLOG definitions is due to the fact that the authors used the predicates intersection and union 'function-like' only, with the third argument of calls being a variable, so that there was no possibility for a 'wrong' order; this is even more probable since a similar problem appeared for the PROLOG subst definition, discussed in subsection 7.1 [while the subst predicate accepts argument tuples which are not related, the intersection and union predicates reject argument tuples which are related]. However, this would support a feeling among functional programmers that it can be very unnatural to keep track of all readings of a relation: one may even forget to think of the basic predicate reading [where all arguments are fixed] if the relation is normally used only function-like [where one argument is variable].

Actually, a functional definition of set union is trivial if it can build on FIT's CLASS function, which performs the often-needed set-normalization, namely sorting without duplicates [cf. section 4];

CLASS can be used in the form CDRoCLASS [e.g., (CDRoCLASS B A B B) via (CDR (CLASS A B)) returns (A B)]:

```
(>(UNION (>X) (>Y)) (CDRoCLASS <X <Y))
```

The UNION function can then be used to define the UNIONP relation:

```
(>(UNIONP ?X ?Y (>Z)) (EQUAL (UNION <X <Y) (CDRoCLASS <Z)))
```

However, in FIT we prefer to represent sets not just as lists without duplicates but directly as CLASS collections, which finally renders the definition of set union as simple as it is conceptually:

```
(>(UNION (CLASS >X) (CLASS >Y)) (CLASS <X <Y))
(>(UNIONP ?X ?Y ?Z) (EQUAL (UNION <X <Y) <Z))
```

If we now use (UNIONP (CLASS A) (CLASS A B) (CLASS B A)) for verifying {a} U {a,b} = {b,a} the third embedded CLASS call normalizes to (CLASS A B) and we get the call (UNIONP (CLASS A) (CLASS A B) (CLASS A B)). The body of UNIONP calls (UNION (CLASS A) (CLASS A B)), which just hands the two CLASS contents to another CLASS, giving (CLASS A A B) that normalizes to (CLASS A B). This UNION result is EQUAL to the normalized third UNIONP argument.

The functional CLASS collection can also be used to simplify our original relational definition of set union: 1. The first two clauses need no more PERMUTATION tests because normalized CLASS collections are set-equal iff they are list-equal. 2. If the fitting of CLASS collections is also defined as commutative, as described using FIT in (Boley 1980) for the more general DRLHs, then the last UNIONP definition clause [where different elements in the argument sets occur in inverse order in the result set] becomes superfluous and in no definition clause does more than one #ID context in a set remain necessary:

```
(UNIONP () ?X ?X)
(UNIONP ?X () ?X)
r\(UNIONP (CLASS ABo?X #ID) (CLASS ABo?X #ID) (CLASS ABo?X #ID))
r(UNIONP (CLASS ABo?X #ID) (CLASS ABo?Y #ID) (CLASS ABo?X ABo?Y #ID))
```

8 THREE EXAMPLES

Finally, let us consider three examples in detail. The first shows a PROLOG programming paradigm, the second is a more neutral PROLOG example, and the third demonstrates a problematic PROLOG relation. All examples are reformulated in FIT; for a more typical FIT programming example, however, see Wang's algorithm in (Boley 1983).

Since in the first two examples a PROLOG relation from the literature, whose name doesn't end in "P", will be represented as a FIT function, we won't maintain the "P"-naming convention in this section.

8.1 Warren's SERIALISE Algorithm

The SERIALISE program has been used as a standard PROLOG example since its introduction in (Warren et al. 1977). We quote from that paper:

"The second example displays many of the characteristics which make Prolog an agreeable language for compiler writing (as applied in the case of our own Prolog compiler). The task is to generate a list of serial numbers for the items of a given list, the members of which are to be numbered in alphabetical order eg.

```
(p.r.o.l.o.g.nil) -> (4.5.3.2.3.1.nil)
```

As with many Prolog programs, the key to arriving at the required algorithm is to first conceive a procedure which checks whether a proposed list of serial numbers is a correct solution. This can be done by pairing up the items of the input list with their proposed serial numbers as an "association list", arranging these pairs in alphabetical order, and then finally checking whether the serial numbers are in the correct consecutive order. i.e.-

```
serialise(L,R) :-
   pairlists(L,R,A),
   arrange(A,T),
   numbered(T,1,N).
```

The pairing is done by a procedure very similar to the pairlis function of the Lisp 1.5 manual, but with the pairs represented as terms 'pair(X,Y)':-

```
pairlists((X.L),(Y.R),(pair(X,Y).A)) :-
   pairlists(L,R,A).
pairlists(nil,nil,nil).
```

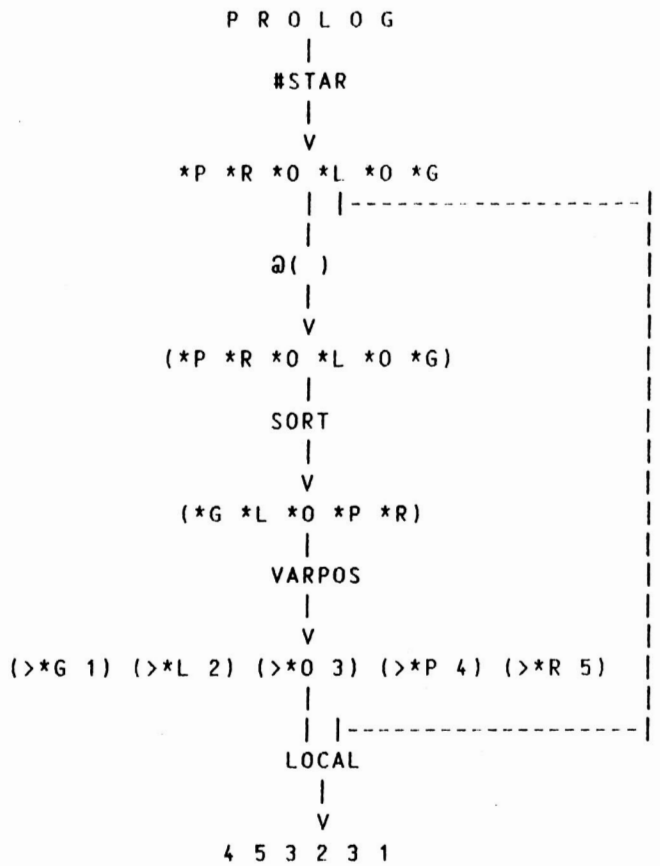
The arrangement in alphabetical order and checking of the numbers could be done using only lists, however it is much more convenient to use binary trees. We represent a tree as a term of the form 'void' ("the void tree") or 'tree(T1,X,T2)' (" a tree with X at the root and subtrees T1 and T2").

```
arrange((X.L),tree(T1,X,T2)) :-
     partition(L,X,L1,L2),
     arrange(L1,T1),
     arrange(L2,T2).
arrange(nil,void).

partition((X.L),X,L1,L2) :- partition(L,X,L1,L2).
partition((X.L),Y,(X.L1),L2) :-
   before(X,Y), partition(L,Y,L1,L2).
partition((X.L),Y,L1,(X.L2)) :-
   before(Y,X), partition(L,Y,L1,L2).
partition(nil,Y,nil,nil).

before(pair(X1,Y1),pair(X2,Y2)) :- X1 < X2.

numbered(tree(T1,pair(X,N1),T2),N0,N) :-
   numbered(T1,N0,N1),
   N2 is N1+1,
   numbered(T2,N2,N).
numbered(void,N,N)."
```

The above program is quite involved and difficult to understand, in spite of the English explanations. Principally, this may be due to the fact that SERIALISE is an instance of those problems for which a relational solution ["check I/O pairs"] is more difficult than a functional solution ["generate output from input"]. Another reason for the program's poor readability is its operation on binary trees instead of on lists, which the authors feel is "much more convenient", but which certainly is a retrograde step to a lower-level data structure [lists are composed of binary trees]. [Incidentally, the PROLOG program features three kinds of binary trees: The standard "." functor for representing lists as binary trees, an isomorphic, hence redundant, "pair" functor for representing dotted pairs in association lists, and a "tree" functor for representing binary trees with labeled roots.] The below FIT version, instead, uses lists internally and impositions [saving unnecessary parentheses] for I/O.

A negative effect of performing the quicksort-like arrangement on an intermediate binary tree structure instead of on sequences is the resulting lack of modularity. It is not possible to regard the sorting subtask as elementary first and only later refine it by writing a sorting module in the usual top-down manner or by using a quicksort from the local program library. To understand PROLOG's serialise, reasoning about finding the serial numbers must be interleaved with reasoning about sorting. In FIT the sorting aspect is completely separated from other aspects of the program. The proper SERIALISE kernel thereby essentially reduces to a four-liner.

The transformations performed by the subfunctions of this FIT SERIALISE program can be illustrated by using the input imposition P R O L O G, corresponding to PROLOG's input list (p.r.o.l.o.g.nil):



As a preparatory first step, SERIALISE applies STAR to all expressions of its input imposition for "*"-encoding them into legitimate variables in a lexical-order preserving manner, before they are assigned to the argument IMP, by composing >IMP with #STAR. [Instead of the STAR encoding a simple NCONS encoding could also be used; inputs, like the above, containing ordinary identifiers only, would require no encoding at all.] Second SERIALISE calls SORT for sorting the encoded imposition >IMP as a list ∂(<IMP) in lexicographical order and with no duplicates [the "∂" instantiates the list contents]. Third it calls VARPOS to generate variable-position bindings from the sorted list elements and their position numbers. Fourth it uses the LOCAL primitive to evaluate the encoded imposition IMP in the binding environment created by VARPOS.

```
(>(SERIALISE >IMPo#STAR)
 (LOCAL (VARPOS 1 (SORT ∂(<IMP) LEXORDER NODUPS)) : IMP))
```

VARPOS generates a variable assignment from each list element X and the number N corresponding to its list position [initially, 1]; hence the recursive VARPOS call both removes one element X and increments N by 1. If the list becomes empty, VARPOS returns the empty imposition.

```
(>(VARPOS ?N (?X >Y)) (><X <N) (VARPOS (ADD1 <N) ∂(<Y)))
(>(VARPOS ID NIL))
```

STAR encodes its argument into a "*"-variable by simply CONS-catenating an asterisk to its argument X, with LISP primitives for transforming an atom to a list of its characters [EXPLODE] and vice versa [READLIST]. [Alternatively, STAR could just LIST the asterisk with its argument X.]

```
(>(STAR ?X) (READLIST (CONS * (EXPLODE <X))))
```

The sorting is done here by the predefined general LISP function SORT. Should this not be available, it could also be defined as a FIT version, e.g. on the basis of QUICKSORT [three-imposition LOCALs are read (LOCAL condition : then-part : else-part); NOTH is FIT's analogue to LISP's NOT; the LT-EQ-GT workhorse function builds on that in (Friedman & Wise 1978); two NEQ calls make it independent from whether the COMPAREFN is <-like or $\leq$-like]:

```
(>(SORT ?L ?COMPAREFN ?NODUPS) (QUICKSORT <L))

(>(QUICKSORT ?L)
 (LOCAL (NULL <L)
        :
        NIL
        :
        (APPEND ( QUICKSORT
                  ID
                  QUICKSORT
                  :
                  (LT-EQ-GT (CDR <L) (CAR <L))))))

(>(LT-EQ-GT ?L ?V)
 (LOCAL (NULL <L)
        :
        NIL
        (LIST <V)
        NIL
        :
        (LOCAL (<COMPAREFN (CAR <L) <V)
               (NEQ (CAR <L) <V)
               :
               ( (TRAFO ?X (CONS (CAR <L) <X))
                 ID
                 ID
                 :
                 (LT-EQ-GT (CDR <L) <V))
               :
               (LOCAL (NOTH (<COMPAREFN (CAR <L) <V))
                      (NEQ (CAR <L) <V)
                      :
                      ( ID
                        ID
                        (TRAFO ?X (CONS (CAR <L) <X))
                        :
                        (LT-EQ-GT (CDR <L) <V))
                      :
                      ( ID
                        (TRAFO ?X
                               (LOCAL (NULL <NODUPS)
                                      :
                                      (CONS (CAR <L) <X)
                                      :
                                      <X))
                        ID
                        :
                        (LT-EQ-GT (CDR <L) <V))))))
```

8.2 McDermott's QUADRAT Program

The author of the QUADRAT PROLOG program for finding the real roots of a quadratic equation characterizes it thus: "It does not show PROLOG at its best (or its worst), but it does make it easy to compare with more traditional languages" (McDermott 1980). Since we feel that this program also suggests a comparison with other non-traditional languages

and a discussion of various other matters, we reproduce it here in a slightly corrected form:

```
quadrat(A,B,C,Realroots) :-
   discrim(A,B,C,D), quadrat1(A,B,D,Realroots).

discrim(A,B,C,D) :-
   mult(B,B,Bsquared), mult(A,C,P1),
   mult(4,P1,P2), add(Bsquared,D,P2).

quadrat1(A,B,D,[]) :- D<0.

quadrat1(A,B,D,[R]) :-
   D=0, add(B,MinusB,0), mult(2,A,TwoA),
   mult(R,TwoA,MinusB).

quadrat1(A,B,D,[R1,R2]) :-
   D>0, add(B,MinusB,0), sqrt(D,SqrtD),
   add(MinusB,SqrtD,Num1),
   add(Num2,SqrtD,MinusB), mult(2,A,TwoA),
   mult(TwoA,R1,Num1), mult(TwoA,R2,Num2).
```

[Like McDermott, we omit the cuts that should follow after D<0 and D=0 and a cut that might redundantly follow after D>0.]

Apparently to illustrate relational programming through all levels, McDermott generously presupposes primitive add and mult relations, although these are not primitive relations but function-like one-directional operators in PROLOG.

It appears to be an inherent problem of relational programming that such [arithmetic] primitives cannot be easily defined as relations: Implementation is unsatisfactory with software and probably even more difficult with digital hardware [perhaps analogical hardware, like the circuits suggested by CONSTRAINTS (Sussman & Steele 1980), is better suited than normal arithmetic units for realizing multiple relation use]. For example, (Clocksin & Mellish 1981) introduce arithmetic operations under the misleading heading of built-in predicates [that a special "is" primitive must be used for evaluating arithmetic expressions, other types of expressions being not evaluable at all, makes things even more inconsistent], (Colmerauer 1983) even attempts to enumerate successor relations extensionally, and (Chikayama 1983) states with regard to the Japanese Fifth-Generation Kernel Language: "Arithmetical operations in KL0 are not bi-directional: Addition and subtraction should be effected by individual operations". While micro-PROLOG is a notable exception in that it does have arithmetic relations, it also illustrates the problem because it restricts their use to at most one unknown argument by simulating the underlying extensional relations imperfectly only (Clark et al. 1982).

It is also obvious that such relational primitives are less readable than their functional counterparts. For example, to decipher the relation call mult(R,TwoA,MinusB) in the second quadrat1 clause, one first has to check which variables will be instantiated at the time of the call, finally finding TwoA and MinusB; only then can one determine which use should be made of the relation by transforming its original product form R * TwoA = MinusB into the quotient form R = MinusB / TwoA.

But now let us assume the add and mult relations were predefined and readable. Then another problem arises when looking at the next higher level of the square and square root operations: Why is square performed by mult(B,B,Bsquared) while square root is performed by sqrt(D,SqrtD), i.e. why isn't there a single relation for both operations? Now, you may notice that sqrt should already be that single relation because in relational programming it should also be readable from right to left, so that mult(B,B,Bsquared) should be replaceable by sqrt(Bsquared,B). However, there would be problems with such a square-sqrt combination. Less importantly, since the range of square is non-negative numbers, the domain of its square root inverse is restricted to these. Therefore, while a relation call with negative second argument like sqrt(Ans,-3) would yield Ans=9 a relation call with negative first argument like sqrt(-9,Ans) would be undefined. More importantly, while the algorithm for square [PROLOG's nonvar primitive is used to ensure that the argument to square is fixed] is trivial,

```
sqrt(Ans,Tosquare) :- nonvar(Tosquare), mult(Tosquare,Tosquare,Ans).
```

the one for sqrt [the argument to square root must be fixed] is not,

```
sqrt(Tosqrt,Ans) :- nonvar(Tosqrt), ... Newton's method ...
```

The point is that there are quite different algorithms for the two uses of the sqrt relation, and incorporating them both into a single [nonvar-less] relation definition would neither be easy nor meaningful. This becomes even more obvious when noticing that in relational programming even the supposed primitive mult should be usable for taking square roots, so that sqrt(D,SqrtD) should be replaceable by mult(sqrtD,sqrtD,D). This should work inversely to mult(B,B,Bsquared) by finding a number sqrtD whose product with itself is D [this relation use is unusual in that one output variable occurs twice to divide the input into two equal factors]. If we hesitated to incorporate the primitive product and quotient functions into a single mult relation, we may be even more concerned about the square-sqrt combination, not to mention a mult integration of the non-primitive square root function. Perhaps McDermott took two completely different relations for square and square root because otherwise "... in PROLOG you have the problem of how to keep straight two separate versions of a relation, for different constellations of inputs" (McDermott 1980). In the last subsection we will see this problem further aggravated.

McDermott comments on his quadrat version: "The first thing to note is that clauses do not contain LISPy deeply-nested function calls, but instead a sequence of relation calls" (McDermott 1980). This is the well-known 'flatness' of PROLOG, normally disliked by functional [for example, LISP] programmers but liked by imperative [for example, PASCAL] programmers.

Now, the "$" [ESCVAL] operator defined in section 3.1 can be used to introduce some nesting into this program making it more concise and more readable:

```
quadrat(A,B,C,Realroots) :-
   quadrat1(A,B,discrim(A,B,C,$D),Realroots).

discrim(A,B,C,D) :-
   add(mult(B,B,$Bsquared),D,mult(4,mult(A,C,$P1),$P2)).
```

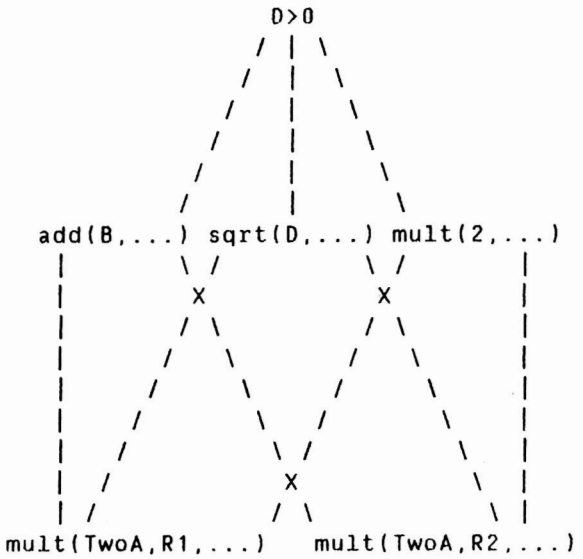
```
quadrat1(A,B,D,[]) :- D<0.

quadrat1(A,B,D,[R]) :-
   D=0, mult(R,mult(2,A,$TwoA),add(B,$MinusB,0)).

quadrat1(A,B,D,[R1,R2]) :-
   D>0, add(B,MinusB,0), sqrt(D,SqrtD), mult(2,A,TwoA),
   mult(TwoA,R1,add(MinusB,SqrtD,$Num1)),
   mult(TwoA,R2,add($Num2,SqrtD,MinusB)).
```

Notice that an expression with embedded ESCVAL expressions can be easily understood in a top-down manner by first abstractly viewing each ESCVAL expression as the ESCVAL variable it will produce. When we 'x off' the functors and arguments thus abstracted away, the top-level of the discrim clause body, for instance, is abstractly viewable as add(x(x,x,$Bsquared),D,x(x,x,$P2)), corresponding to the last conjunct add(Bsquared,D,P2) in the original clause. While in certain clause bodies the entire conjunction is joined to a single relation nesting [cf. the quadrat and discrim clauses], in other ones the conjunction becomes at least smaller by joining some of its conjuncts to relation nestings [cf. the last two quadrat1 clauses].

A corresponding ESCVAL-enriched FIT version of QUADRAT is the following:

```
(>(QUADRAT ?A ?B ?C ?REALROOTS)
 (QUADRAT1 <A <B (DISCRIM <A <B <C $|?D) <REALROOTS))

(>(DISCRIM ?A ?B ?C ?D)
 (ADD (MULT <B <B $|?BSQUARED) <D (MULT 4 (MULT <A <C $|?P1) $|?P2)))

(>(QUADRAT1 ?A ?B ?D ()) (LESSP <D 0))

(>(QUADRAT1 ?A ?B ?D (?R))
 (LOCAL (EQ <D 0)
        :
        (MULT <R (MULT 2 <A $|?TWOA) (ADD <B $|?MINUSB 0))))

(>(QUADRAT1 ?A ?B ?D (?R1 ?R2))
 (LOCAL (GREATERP <D 0)
        :
        (LOCAL (ADD <B |?MINUSB 0)
               (SQRT <D |?SQRTD)
               (MULT 2 <A |?TWOA)
               :
               (MULT <TWOA <R1 (ADD <MINUSB <SQRTD $|?NUM1))
               (MULT <TWOA <R2 (ADD $|?NUM2 <SQRTD <MINUSB)))))
```

The expression (LOCAL (EQ <D 0) : (MULT ...)) in the second QUADRAT1 clause reflects the real 'if then' meaning of the conjunction in the corresponding PROLOG clause, namely if D=0 then mult(...). Similarly, the outer LOCAL of the last QUADRAT1 clause is best viewed as an 'if then' condition [control flow]. Its inner LOCAL is best viewed as a generalized LET expression which introduces the variables MINUSB, SQRTD, and TWOA through relation calls [data flow]. The partial order of the data and control flow of that clause's conjuncts is



This cannot be seen in PROLOG's corresponding clause, where, for example, D>0 precedes add(B,...), as required by the control flow, add(B,...) precedes mult(TwoA,R1,...), as required by data flow, but add(B,...) also precedes sqrt(D,...), which is required neither by control nor data flow [this is what (Leavenworth & Sammet 1974) call "arbitrary sequencing" whose elimination is prerequisite for a language to be non-procedural]. The nested FIT LOCALs of the last QUADRAT1 clause, on the other hand, directly reflect the control and data flow [the textual order inside LOCAL is immaterial apart from the relationship 'left of colon'/'right of colon'].

A completely functional version of QUADRAT in FIT can be defined thus:

```
(>(QUADRAT ?A ?B ?C) (QUADRAT1 <A <B (DISCRIM <A <B <C)))

(>(DISCRIM ?A ?B ?C)
 (DIFFERENCE (SQUARE <B) (TIMES 4 <A <C)))

(>(QUADRAT1 ?A ?B 0)
 (QUOTIENT (MINUS <B) (TIMES 2 <A)))

(>(QUADRAT1 ?A ?B x?POSINT)
 (LOCAL (>SQRTD (SQRT <POSINT))
        :
        (QUOTIENT (PLUS (MINUS <B) ((BREADTH PLUS MINUS) <SQRTD))
                  (TIMES 2 <A))))

(>(POSINT ?N) (GREATERP <N 0))
```

This is the first version which directly reflects the problem specification because it "finds the real roots of a quadratic" (McDermott 1980) and does not appear to solve further problems like finding the infinitely many quadratics with given real roots. Perhaps it is also the first version which is readable without much pondering.

Instead of yielding the empty list if there is no real solution, a one-element list if there is one solution, and a two-element list if there are two, the program exploits the non-determinism of FIT and

yields jU, a single value, and a BREADTH of two values in these respective cases. Indeed we regard quadratics as a nice example for the explicit specification of non-determinism: The caller of a quadratics program should receive a failure if there is no solution for the given arguments, so that, e.g., other arguments may be tried automatically; the caller should receive just a single value if there is exactly one solution, so it can proceed deterministically, not even noticing the principal possibilities of failure and ambiguity; and the caller should receive a BREADTH of two equal-right values if there are two solutions. Note that the explicit non-deterministic branch (BREADTH PLUS MINUS) in the last QUADRAT1 clause corresponds exactly to the use of $\pm$ in mathematics. This is not possible with PROLOG's implicit depth-oriented non-determinism. Our use of non-deterministic instead of listified root results also has another advantage: It allows us to get rid of the first quadrat1 clause because for D<0, no other clause being applicable, jU is yielded automatically. The empty list could not be yielded in such an automatic manner. Although it would not have been necessary during the relational-functional translation, we replaced the condition D=0 by a constant 0 and replaced the condition D>0 by a typed variable x?POSINT in the invocation pattern. The typed variable is built from the generally useful predicate POSINT for positive integers.

The last QUADRAT1 clause may be further shortened to finally obtain the usual mathematical form of the quadratic algorithm:

```
(>(QUADRAT1 ?A ?B x?POSINT)
 (QUOTIENT (PLUS (MINUS <B) ((BREADTH PLUS MINUS) (SQRT <POSINT)))
           (TIMES 2 <A)))
```

However, the earlier clause precomputing the SQRT of POSINT in a LOCAL is preferable for efficiency reasons because under FIT-1's evaluation strategy ((BREADTH PLUS MINUS) (SQRT <POSINT)) would immediately normalize to (BREADTH (PLUS (SQRT <POSINT)) (MINUS (SQRT <POSINT))), so that (SQRT <POSINT) would be evaluated twice.

8.3 Fermat's Last Theorem

The FERMAT example shows that for some relations there is no known algorithm which uses them in one way, whereas there is an algorithm which uses them in another way. Let us begin with a trivial example often used to illustrate relational programming (Kowalski 1979) and constraint systems (Sussman and Steele 1980), namely the equation

X + Y = Z

which in PROLOG is written as a relation

```
plus(X,Y,Z).
```

Since this equation can be regarded as X^1 + Y^1 = Z^1, Fermat's equation might seem to be just a little bit more general. It is

X^N + Y^N = Z^N

and is considered as a relation

```
fermat(X,Y,Z,N).
```

To simplify the following discussion we presuppose that X, Y, Z, as well as N, are non-negative integers [not all PROLOGs have negative integers]. The relation call fermat(4,3,5,2), for instance, should succeed because 4^2 + 3^2 = 5^2. But what about calls with request variables like fermat(4,3,5,N) and fermat(X,Y,Z,2)? Although both may look harmless, in general we can only define the former use of the fermat relation [X, Y, and Z are fixed -- N is open], not the latter use [X, Y, and Z are open -- N is fixed]. In other words, if we split the fermat relation into two functions ferm and ferm-I [here, X, Y, Z, and N denote the set of non-negative integers; the empty set, {}, denotes explicit failure]

```
ferm: X x Y x Z -> N U { {} }
ferm-I: N -> powerset(X x Y x Z)
```

these can be called with specific arguments as in ferm(4,3,5) = 2 and ferm-I(2) = {(4,3,5), ...} [for the argument N=2 ferm-I is infinitely non-deterministic; cf. section 3.1]. In general, however, we know only that ferm is computable, but don't know whether ferm-I is. The former is demonstrated below; the latter is the case because there still is no known proof or disproof of "Fermat's last theorem", stating that for an integer N>2 the equation

```
X^N + Y^N = Z^N
```

has no solution in integers all different from 0 (Ribenboim 1979), i.e. no 'non-null solution'. So the relational representation of the Fermat equation will lead to a severe problem [a not generally usable relation fermat must be introduced], not arising in its functional representation [a generally usable function ferm can be introduced without at the same time introducing a not generally usable function ferm-I].

To find N or to yield a failure if none exists for arbitrary given X, Y, and Z a relational PROLOG program can be defined.

For this we first construct the underlying algorithm which relies on the following observations. Since for $X \geq Z$ or $Y \geq Z$ there clearly can be no non-null solution, we can presuppose $X<Z$ and $Y<Z$. Now we can show two facts.

1. If a Z exponentiation once became greater than the sum of the X and Y exponentiations, it will remain greater for all higher exponents, i.e.

```
if Z^N > X^N + Y^N then Z^N+1 > X^N+1 + Y^N+1 for all N
```

This can be seen very easily. Assuming $Z^N > X^N + Y^N$ and multiplying it with Z we get

```
Z^N+1 = Z*Z^N > Z*[X^N + Y^N] = Z*X^N + Z*Y^N
```

Since Z>X we get

```
Z*X^N + Z*Y^N > X*X^N + Z*Y^N
```

Similarly, since Z>Y we get

$$X*X^N + Z*Y^N > X*X^N + Y*Y^N = X^{N+1} + Y^{N+1} .$$

2. Z^N grows faster with N than $X^N + Y^N$ does, i.e.

there exists an integer N' such that $Z^N > X^N + Y^N$ for all N>N'.

This can be shown by the following elementary transformations. We can assume without loss of generality that $X \geq Y$. Z^N can be rewritten as $[X + D]^N$ with $D \geq 1$ because Z>X. The binomial theorem gives us [the binomial coefficients are defined by binco(N,K) := N*[N-1]*[N-2]*...*[N-K+1] / 1*2*3...*K]

$$[X + D]^N = X^N + binco(N,1)*X^{[N-1]}*D + binco(N,2)*X^{[N-2]}*D^2 + ... + binco(N,N-1)*X*D^{[N-1]} + D^N$$

If we omit the terms of the sum from $binco(N,2)*X^{[N-2]}*D^2$ we get

$$Z^N = [X + D]^N > X^N + binco(N,1)*X^{[N-1]}*D = X^N + N*X^{[N-1]}*D$$

Since $D \geq 1$ we get

$$X^N + N*X^{[N-1]}*D \geq X^N + N*X^{N-1}$$

If we set N'=X then for all N>N'

$$Z^N > X^N + N*X^{N-1} > X^N + X*X^{N-1} = X^N + X^N$$

Since $X \geq Y$

$$X^N + X^N \geq X^N + Y^N .$$

Using these facts we get the following concise but inefficient algorithm in ALGOL-like notation.

```
if X≥Z or Y≥Z then fail ;
N := 1 ;
while X^N + Y^N > Z^N do N := N+1 ;
if X^N + Y^N = Z^N then N else fail
```

Fact 2 ensures termination of the while loop.
Fact 1 permits the fail in the else case, i.e. if $X^N + Y^N < Z^N$.

Now, the algorithm can be rewritten into a more efficient PROLOG program, which accumulates exponentiations instead of recomputing them.

```
fermat(X,Y,Z,N) :-
   nonvar(X), nonvar(Y), nonvar(Z), X<Z, Y<Z,
   XY is X+Y, fermat2(X,Y,Z,X,Y,XY,Z,1,N).

fermat2(X,Y,Z,XX,YY,XXYY,ZZ,M,N) :-
   XXYY=ZZ, N is M.

fermat2(X,Y,Z,XX,YY,XXYY,ZZ,M,N) :-
   XXYY<ZZ, fail.
```

```
fermat2(X,Y,Z,XX,YY,XXYY,ZZ,M,N) :-
   XXYY>ZZ, XXX is XX*X, YYY is YY*Y, XXXYYY is XXX+YYY, ZZZ is ZZ*Z,
   M1 is M+1, fermat2(X,Y,Z,XXX,YYY,XXXYYY,ZZZ,M1,N).
```

If none of X, Y, and Z is an [open] variable and both X and Y are less than Z the fermat program calls the auxiliary tail-recursive relation fermat2. The arguments of the fermat2 program are the original variables X, Y, and Z, variables XX and YY for accumulating X and Y exponentiations, a variable XXYY for storing the sum of XX and YY, a variable ZZ for accumulating the Z exponentiations, a variable M for holding the current exponent, and the original variable N for handing the found exponent back to fermat. The initial fermat2 call essentially uses XXYY=XY=X+Y and M=1. The use of its nine arguments reduces the task of fermat2 to a simple case analysis on the relationship between XXYY and ZZ.

If XXYY=ZZ then the current value of M is the exponent sought for [obvious] and is assigned to N.

If XXYY<ZZ then this relationship would also hold for all subsequent recursions with higher exponents [fact 1] and a failure can be generated.

If XXYY>ZZ then XXYY will become equal to or less than ZZ for some higher exponent [fact 2] and fermat2 is called recursively. This call, apart from the original X, Y, and Z variables, could use the variables XX:=XX*X, YY:=YY*Y, XXYY:=XX+YY, ZZ:=ZZ*Z, and M:=M+1, if PROLOG's single-assignment property wouldn't enforce the use of new intermediate variables XXX, YYY, XXXYYY, ZZZ, and M1, respectively.

To find X, Y, and Z or to yield a failure if none exist for arbitrary given N no PROLOG program is known, however.

```
fermat(X,Y,Z,N) :- nonvar(N), ... unknown method ...
```

A functional FIT program that finds N or yields jF if none exists for given X, Y, and Z can be defined thus:

```
(>(FERM ?X ?Y ?Z)
 (LOCAL (LESSP <X <Z)
        (LESSP <Y <Z)
        :
        (FERM2 <X <Y <Z <X <Y (PLUS <X <Y) <Z 1)))

(v(FERM2 ?X ?Y ?Z ?XX ?YY ?XXYY ?ZZ ?N)
 (LOCAL u(EQ <XXYY <ZZ) : <N))

(v(FERM2 ?X ?Y ?Z ?XX ?YY ?XXYY ?ZZ ?N)
 (LOCAL u(LESSP <XXYY <ZZ) : jF))

(v(FERM2 ?X ?Y ?Z ?XX ?YY ?XXYY ?ZZ ?N)
 (LOCAL u(GREATERP <XXYY <ZZ)
        :
        (LOCAL (>XXX (TIMES <XX <X))
               (>YYY (TIMES <YY <Y))
               :
               (FERM2 <X <Y <Z <XXX <YYY (PLUS <XXX <YYY)
                        (TIMES <ZZ <Z) (ADD1 <N)))))
```

This works like the corresponding PROLOG fermat program, except for the following differences. The FIT FERM program directly nests (PLUS <X <Y) into its FERM2 call instead of first introducing an intermediate variable XY to transport X+Y into the call as done in PROLOG. Also, FERM needs no M variable because N, not being used for holding a request variable, can itself be used for exponentiation accumulation. Then, in the case XXYY=ZZ FERM2 returns N instead of assigning M to N. For XXYY<ZZ it yields jF to signal that no N exists [here FIT's jF is clearer than PROLOG's fail, which could also mean, like jU, that it is unknown whether an N exists]. If XXYY>ZZ only two additional variables XXX and YYY are used instead of five in the PROLOG version [in FIT even these are only for efficiency, avoiding two additional multiplications, whereas in PROLOG three further variables are necessary because nestings like fermat2(X,Y,Z,XXX,YYY,XXX+YYY,ZZ*Z,M+1,N) are not allowed].

The above case analysis by EQ, LESSP, and GREATERP calls in LOCAL bodies corresponds to clauses with constraints on the FERM2 invocation pattern [cf. section 6.3]. In FIT, such constraints can also be put directly into an invocation adapter, here constructed by putting the functions EQ, LESSP, and GREATERP into the invocation pattern. In this way, the FERM2 definitions can be shortened to

```
(v(FERM2 ?X ?Y ?Z ?XX ?YY EQ ?N) <N)

(v(FERM2 ?X ?Y ?Z ?XX ?YY LESSP ?N) jF)

(v(FERM2 ?X ?Y ?Z ?XX ?YY (COMPOSE GREATERP ?XXYY ?ZZ) ?N)
 (LOCAL (>XXX (TIMES <XX <X))
        (>YYY (TIMES <YY <Y))
        :
        (FERM2 <X <Y <Z <XXX <YYY (PLUS <XXX <YYY)
                 (TIMES <ZZ <Z) (ADD1 <N))))
```

The GREATERP function is composed with the original variables XXYY and ZZ because the value of ZZ is needed in the body.

A functional FIT program that finds X, Y, and Z for given N would be something completely separate from the above FERM function [namely the non-deterministic inverse function FERM-I]. That FERM-I cannot be defined doesn't restrict the applicability of the FERM function, whereas the non-definability of the corresponding relation use does restrict the applicability of the fermat relation.

It has often been pointed out in the PROLOG literature that the cut operator (Clocksin & Mellish 1981) and the execution order (Kowalski 1983) obstruct the multiple useability of relations; what seems to be less well known is the fact that even without any cut and with any conceivable execution strategy some relations cannot be used in a multiple manner. In the latter case the problem resides in the relational formulation [in the 'logic'] itself, not in a particular deduction procedure [in a 'control'] working on it. Let us further reformulate our point in Kowalski's terminology: Not only in PROLOG but even in logic programming [which is more pure because it is cut-less and non-sequential], there are programs for which invertibility, as defined by "This characteristic of logic programs, that it is possible to find any individual in a relationship with other individuals, is called <u>invertibility</u>." (Kowalski 1983), <u>cannot</u> be achieved.

The original source of the fermat problem can be traced back to the fact that in PROLOG Fermat's equation, like every top-level assertion, can only be formulated as a relation, fermat, not as a function, ferm; an illegitimate ferm-I use of this relation could only be prevented by superimposed "mode declarations" (Warren et al. 1977) [normally used for enhancing compiler efficiency], which are extraneous to the relational formalism. The fermat example is thus a signal cautioning against indiscriminate relational programming. This specializes the original interpretation of Fermat's last theorem for specification languages, namely that "there will never be a "solution" to the automatic programming problem" (Feldman 1972), also adopted in (Leavenworth & Sammet 1974).

9 REFERENCES

Aida, H. & Tanaka, H. & Moto-oka, T.: A Prolog extension for handling negative knowledge. New Generation Computing 1(1), 1983, 87-91.

Backus, J.: Function-level computing. IEEE spectrum, August 1982, 22-27.

Bellia, M. & Degano, P. & Levi, G.: The call by name semantics of a clause language with functions. In: (Clark & Taernlund 1982), 281-295.

Bendl, J. & Koeves, P. & Szeredi, P.: The MPROLOG system. In: Taernlund, S.-A. [Ed.]: Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980, 201-210.

Bibel, W.: Knowledge representation from a deductive point of view. Technische Universitaet Muenchen, Institut fuer Informatik, Projekt Beweisverfahren, Bericht ATP-19-V-83, May 1983. Also in: Proc. IFAC Symposium on Artificial Intelligence, Leningrad, Oct. 1983, Pergamon Press, to appear.

Boley, H.: Processing directed recursive labelnode hypergraphs with FIT programs. Univ. Hamburg, FB Informatik, IFI-HH-M-81/80, Sept. 1980.

Boley, H.: Artificial intelligence languages and machines. Univ. Hamburg, FB Informatik, IFI-HH-B-94/82, Dec. 1982. Final version in: Technology and Science of Informatics 2(3), May-June 1983.

Boley, H.: From pattern-directed to adapter-driven computation via function-applying matching. Univ. Kaiserslautern, FB Informatik, Interner Bericht 81/83, MEMO SEKI-83-06. Also in: GI - 13. Jahrestagung, Hamburg, Oct. 1983, Springer 1983.

Bowen, K.: Programing with full first-order logic. Machine Intelligence 10, 1982, 421-440.

Chikayama, T.: ESP - extended self-contained PROLOG - as a preliminary kernel language of fifth generation computers. New Generation Computing 1(1), 1983, 11-24.

Clark, K. & Ennals, J. & McCabe, F.: A micro-PROLOG primer. Logic Programming Associates Ltd., 36 Gorst Rd., London SW11 6JE, England, April 1982.

Clark, K. & Gregory, S.: A relational language for parallel programming. Proc. of the Conference on Functional Programming Languages and Computer Architecture, ACM, October 1981, 171-178.

Clark, K. & Taernlund, S.-A. [Eds.]: Logic programming. Academic Press, London, 1982.

Clocksin, W. & Mellish, C.: Programming in Prolog. Springer-Verlag, Berlin Heidelberg New York, 1981.

Colmerauer, A: Prolog in 10 figures. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, 487-499.

Conery, J. & Kibler, D.: Parallel interpretation of logic programs. Proc. of the Conference on Functional Programming Languages and Computer Architecture, ACM, October 1981, 163-170.

Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. CACM 18(8), Aug. 1975, 453-457.

Eggert, P. & Schorre, D.: Logic enhancement: a method for extending logic programming languages. Conference Record of the 1982 ACM Symposium on LISP and Functional Programming. Pittsburgh, Penn., August 1982, 74-80.

Feigenbaum, E. & McCorduck, P.: The fifth generation: Artificial intelligence and Japan's computer challenge to the world. Addison-Wesley, Reading, 1983.

Feldman, J.: Automatic programming. Stanford University, Computer Science Department, CS-255, Feb. 1972.

Feldman, J. & Low, J. & Swinehart, D., Taylor, R.: Recent developments in SAIL - An ALGOL-based language for artificial intelligence. Proc. AFIPS 1972 FJCC 41, 1972, 1193-1202.

Friedman, D. & Wise, D.: Functional combination. Computer languages, Vol. 3, 31-35, 1978.

Fuchi, K.: Aiming for knowledge information processing systems. In: Moto-oka, T. [Ed.]: Proceedings of the international conference on fifth generation computer systems. Tokyo, October 1981. North-Holland 1982, 101-114.

Fuhlrott, O.: PROLOG als Datenbank- und Programmiersprache. Univ. Hamburg, FB Inform., Oberseminar Datenbanken und Informationssysteme, Nov. 1982.

Fuhlrott, O.: personal communication. Hamburg, December 1983.

Fuhlrott, O.: A personal bibliography on logic programming, PROLOG, databases. O.Fuhlrott, Bekassinenau 92, D-2000 Hamburg 73, W.Germany, 1984.

Hansson, A. & Haridi, S. & Taernlund, S.-A.: Properties of a logic programming language. In: (Clark & Taernlund 1982), 267-280.

Henderson, P.: Functional programming. Application and implementation. Prentice-Hall International, London 1980.

Hewitt, C. & de Jong, P.: Analyzing the roles of descriptions and actions in open systems. Proc. AAAI-83, Washington, Aug. 1983, 162-167.

Hilbert, D. & Bernays, P.: Grundlagen der Mathematik II. Springer-Verlag, Berlin Heidelberg 1939, Zweite Auflage 1970.

Hussmann, M.: personal communication. Hamburg, November 1983.

Jaffar, J. & Lassez, J.-L. & Lloyd, J.: Completeness of the negation as failure rule. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, 500-506.

Kahn, K.: Unique features of Lisp Machine Prolog. Uppsala Programming Methodology and Artificial Intelligence Laboratory, UPMAIL Technical Report No. 15, 1983-02-14.

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

Kowalski, R.: Logic for problem solving. North-Holland, 1979.

Kowalski, R.: Logic programming. Proc. IFIP 83, Paris, 1983, 133-145.

Kurokawa, T.: LOGIC PROGRAMMING -- What does it bring to the software engineering? In: VanCaneghem, M. [Ed.]: Proceedings of the First International Logic Programming Conference. Marseille, Sept. 1982, 134-138.

Landin, P.: A correspondence between ALGOL60 and Curch's lambda-notation: Part I. CACM Vol. 8, No. 2, Febr. 1965, 89-101.

Leavenworth, B. [Ed.]: ACM SIGPLAN symposium on very high level languages. March 1974, Santa Monica, Ca., SIGPLAN Notices 9(4).

Leavenworth, B. & Sammet, J.: An overview of nonprocedural languages. In: (Leavenworth 1974).

McDermott, D.: The PROLOG phenomenon. SIGART Newsletter, No. 72, July 1980, 16-20.

McDermott, J. & Forgy, C.: Production system conflict resolution strategies. In: (Waterman & Hayes-Roth 1978).

Nakashima, H. & Suzuki, N.: Data abstraction in Prolog/KR. New Generation Computing 1(1), 1983, 49-62.

Nebel, B.: personal communication. Hamburg, September 1983.

Newell, A.: Production systems: Models of control structures. In: Chase, W. [Ed.]: Visual information processing. Academic Press, 1973, 463-526.

O'Keefe, R.: PROLOG compared with LISP? SIGPLAN Notices 18(5), May 1983, 46-56.

Ribenboim, P.: 13 lectures on Fermat's last theorem. Springer-Verlag, New York Heidelberg Berlin, 1979.

Robinson, J. & Sibert, E.: The LOGLISP user's manual. School of Computer and Information Science, Syracuse University, December 1981.

Robinson, J. & Sibert, E.: LOGLISP: an alternative to PROLOG. Machine Intelligence 10, 1982, 399-419.

Rulifson, J. & Derksen, J. & Waldinger, R.: QA4: A procedural calculus for intuitive reasoning. Stanford Research Institute, AI Center, Technical Note 73, Nov 1972.

Sato, M. & Sakurai, T.: Qute: A Prolog/Lisp type language for logic programming. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, 507-513.

Shapiro, E.: Methodology of logic programming research. Logic Programming Workshop, Portugal, 1983.

Shapiro, E. & Takeuchi, A.: Object oriented programming in Concurrent Prolog. New Generation Computing 1(1), 1983, 25-48.

Siekmann, J. & Szabo, P.: Universal unification. In: Wahlster, W. [Ed.]: GWAI-82, Bad Honnef, Sept. 1982, Informatik-Fachberichte 58, Springer-Verlag, 102-141.

Stefik, M. & Bobrow, D. & Mittal, S. & Conway, L.: Knowledge programming in LOOPS: Report on an experimental course. The AI Magazine 4(3), Fall 1983, 3-13.

Sussman, G. & Steele, G.: CONSTRAINTS - A language for expressing almost-hierarchical descriptions. Artificial Intelligence 14, 1980, 1-39.

VanEmden, M.: McDermott on Prolog: A rejoinder. SIGART Newsletter 73, October 1980, 19-20.

Warren, D.: Higher-order extensions to PROLOG: are they needed? Machine Intelligence 10, 1982, 441-454.

Warren, D. & Pereira, L. & Pereira, F.: PROLOG - The language and its implementation compared with LISP. Proc. Symposium on Artificial Intelligence and Programming Languages. SIGPLAN Notices 12(8), Special Issue, August 1977, 109-115.

Waterman, D. & Hayes-Roth, F. [Eds.]: Pattern-directed inference systems. Academic Press, 1978.

Weinreb, D. & Moon, D. & Stallman, R.: LISP machine manual. Fifth edition. MIT, AI Lab., Jan. 1983.

Winston, P. & Horn, B.: LISP. Addison-Wesley, Reading, 1981.