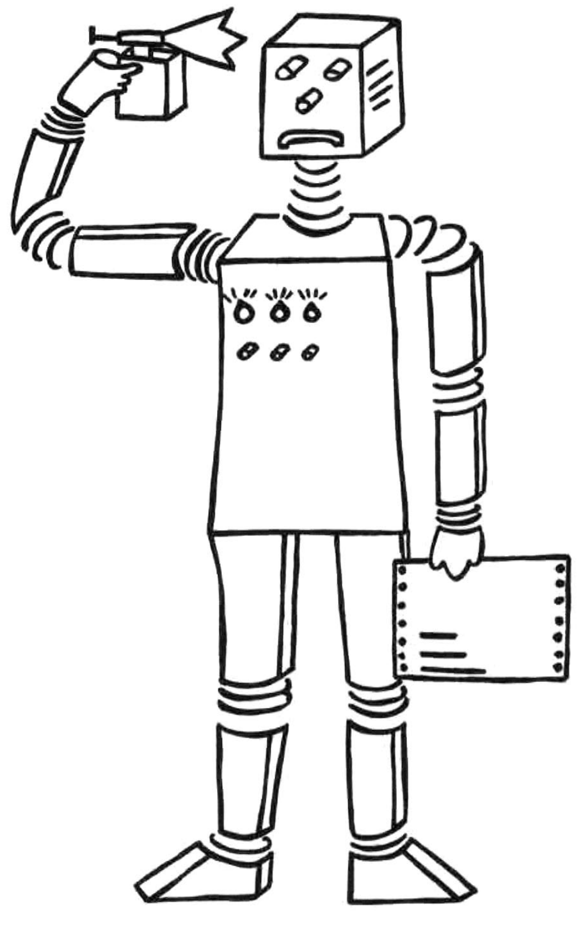
SEKI-REPORT

Artificial
Intelligence
Laboratories

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



Iconic-Declarative Programming and Adaptation Rules

Harold Boley

SEKI Report SR-88-04

ICONIC-DECLARATIVE PROGRAMMING AND ADAPTATION RULES

Harold Boley, FB Informatik, Univ. Kaiserslautern

Abstract: Functional and logical languages permit declarative programming, i.e. executable high-level problem specifications. However, to obtain optimum intelligibility, programs should also be iconic, i.e. directly model their domain, as illustrated imperatively by object-oriented languages. After a discussion of iconic aspects in pure LISP (call nestings) and pure PROLOG (invocation patterns), an advanced iconic-declarative technique in the functional/logical AI language FIT is presented: adaptation rules join left-hand side invocation patterns and right-hand side call nestings of transformation rules into single pictorial contexts that process data by global tests and direct local transformations. Their utility is exemplified in three domains (list processing, set normalization, and graph searching). The conclusions criticize the predominance of transformation rules and contrast iconic-imperative with iconic-declarative developments.

1. Introduction

Programming has been renowned for involving multiple layers of manual or automatic translation that mediate between the given problems and machines. So-called 'high-level' -- or even 'very-high-level' -- languages have attempted to shield application programmers from most of the lower, machine-oriented levels, and to retain -- in the ideal case -- the uppermost, 'problem-oriented' level only. Today, the equivalent attribute 'declarative' is mainly claimed by two such groups of languages, namely the -- purely -- functional and logical paradigms. However, their principal realizations, LISP and PROLOG, in real live -- AI and otherwise -- are often applied in a quite non-declarative, 'imperative' fashion.

On the other hand, the object-oriented paradigm is gaining momentum in AI via a third major group of programming languages. Their flagship, SMALLTALK, ubiquitously employs imperative features such as instance variables (hence a state concept) and PASCAL-like method definitions. In our opinion, however, the potential of object-oriented programming is not its imperative features themselves but a not necessarily related, more essential characteristics: The directness of its (often 1-to-1) modeling of real-world objects by language objects.

This direct modeling is typical for what we will call 'iconic' languages, which permit programs (representations) to mirror the structure of the data (world) in the 'pictorial' or 'analogical' manner of a homomorphic (in extreme cases, isomorphic) image.

Our emphasis in this paper will be on

(1) what we call deep icons as opposed to the usual surface icons, i.e. we are interested in internal, information-processing applications of direct modeling, not in external information-presentation applications via graphics, even though this latter, well-known sense of 'iconic' has contributed a lot to SMALLTALK's popularity;

(2) iconic programs as opposed to the more general and well-known concept of iconic representations, although this distinction is not very sharp since in SMALLTALK, for instance, representation of the world happens to be done using program-like objects.

An obvious advantage of iconicity -- also inherent in the 'Wysiwyg' ("What you see is what you get") principle for text processing -- is the self-explanatory 'meaning immanence' of iconic programs, not requiring extraneous, 'symbolic-denotational' interpretation mappings to reach the semantics.

Thus the present attempts at integrating functional/logical languages on the one hand with object-oriented languages on the other hand (hence, the three major AI paradigms) may be seen under the following double perspective:

(1) Can object-oriented languages be made more declarative or,

(2) Can functional/logical languages be made more iconic?

Coming from a functional/logical background [Boley 1986 1987], in the following sections we will concentrate on the second half of this question, i.e. on the issue of iconic-declarative languages. Our approach is to complement the above-discussed object-oriented (imperative) technique of data mirroring by a new functional/logical (declarative) pattern-derived kind of iconic programs. These 'adaptation rules' or 'adapters' match data structures like patterns do, and simultaneously process some of their parts by embedded functions.

Adaptation rules are the central construct of FIT, a LISP-implemented declarative AI language; in our experience they blend nicely with the more well-known, less iconic, transformation rules (including PROLOG's Horn rules) and with the widespread non-iconic functions (including LISP's lambda definitions).

Even though we want to proceed 'upward' the declarative axis, it is also possible to look 'downward', while keeping on the iconic side of computing: Instead of direct modeling on a (SMALLTALK-like) language level, also direct models down to the hardware level can be employed, as discussed from AI classics like [Minsky & Papert 1971] and [Sloman 1971] to many modern 'connectionist' papers since [Hillis 1985].

Extrapolating from all these occurrences of iconicity on arbitrary levels of declarativeness we suggest that the 'declarative'/'iconic' distinction is indeed an orthogonal one, as illustrated by a summary of this introductory section in the 'qualitative' diagram below.

```
                                   *declarative
                                   *
                                   d
        |--------------------|-----e--PROLOG|---------------FIT|
        |--------------LISP| |Horn c clauses| adaptation rules |
        |lambda expressions| |     l        |                  |
        |                  | |     a        |                  |
        |                  | |     r        |                  |
        |------------------|-|-----a--------|------------------|
non-iconic (symbolic)      | |     t        |                      iconic
********|******************|*|*iconIcity****|***************************
        |                  | |     v        |
        |      |-----------|-|-----e--------|---------SMALLTALK|
        |ordinary imperative |     n        | object-oriented  |
        |      programs    | |-----e--------|    programs      |
        |      |-----------|-------s---------------------------|
        |------------------|       s
                                   *
                                   *non-declarative (imperative)
```

2. Iconicity in functional and logical languages

In the sense conceived here, iconicity is not a yes/no property but constitutes a fairly continuous scale, ranging from completely non-iconic to partially iconic, to completely iconic languages. Moreover, there are several facets of iconicity, whose summarizing in a single dimension must be taken as a first approximation. This section compares two such facets, operator calls and definitions, for functional and logical languages. The next section will then proceed to the more iconic adaptation rules of FIT.

2.1. Call iconicity

An operator (function) call in functional languages takes n arguments and returns 1 value. Therefore, call nestings are permitted, i.e. n inner function calls can be written directly at the argument positions of an n-ary outer function call. We regard this standard method of nesting as a simple but important kind of iconicity, because the correct call correspondences are established automatically.

An analogous operator (predicate) call in logical languages takes n+1 arguments, where, say, the n first arguments are inputs and the last argument is an output variable. Instead of call nestings, sequential

call conjunctions must be employed, where the first n calls bind n auxiliary output variables and the last call uses their values as its respective inputs. We regard this method of conjunction as a good example of non-iconic programming, because only through the symbolic links of the n output variables to their 'second occurrences' as input variables are the correct call correspondences established.

For example, with LISP functions (-f) we can iconically use the nesting

```
(append-f (reverse-f '(1 2 3)) (delete-f 'b '(a b)))
```

while with PROLOG relations (-r) we have to introduce symbolic names like Aux, Auy, and Res:

```
reverse-r([1,2,3],Aux), delete-r(b,[a,b],Auy), append-r(Aux,Auy,Res).
```

We hope that the above discussion has made the 'flatness' critique of PROLOG in, e.g., [McDermott 1980] and [Boley 1987] more precise: As much as we estimate logical variables for non-ground functional programming [Boley 1986], we also criticize their nesting replacement as a typical non-iconic technique. We can even go one step further, contending that the sometimes vague scepticisms against the use of (subsets of) first-order logic in AI can be explicated by criticizing its dependence on lots of symbolic variables for establishing indirect, non-iconic, hard-to-read correspondences across (potentially large conjunctive) formulas. Higher-order logics may be used to eliminate many of these variables, e.g. replacing the first-order rules

```
uncle-r(Unc,Nep) :- brother-r(Unc,Aux), parent-r(Aux,Nep).
grandmother-r(Gma,Gch) :- mother-r(Gma,Aux), parent-r(Aux,Gch).
```

by the second-order facts

```
relation-product(brother-r,parent-r,uncle-r).
relation-product(mother-r,parent-r,grandmother-r).
```

but also introduce (a smaller number) of additional, higher-order variables, as in the abstracted second-order rule (PROLOGish syntax, RELFUN semantics)

```
Prod(Ind1,Ind3) :- relation-product(Rel1,Rel2,Prod),
                   Rel1(Ind1,Ind2), Rel2(Ind2,Ind3).
```

2.2. Definition iconicity

An operator (function) definition in functional languages associates the function name with an entire function expression consisting of the parameters and a call nesting, often written as a lambda expression.

This link is a paradigm of non-iconic techniques because an arbitrarily chosen symbolic name is defined to denote a lambda function as a whole.

An analogous operator (predicate) definition in logical languages associates invocation patterns (containing the predicate name followed by possible further fixed parts and the parameters) with call conjunctions (empty for facts), i.e. it is a system of Horn clauses. While these links themselves again are not iconic, each of the patterns is a very iconic construct because it pictorially mirrors the set of operator calls for which the associated call conjunction is applicable.

For example, let us consider a functional (-f) and a relational (-r) version of an operator selecting the nth list element. While in LISP we must define the operator nth-f non-iconically by one named lambda-expression like

```
(def nth-f (lambda (n l)
              (cond ((eq 1 n) (car l))
                    (t (nth-f (sub1 n) (cdr l))) )))
```

in PROLOG we can define the analogous operator nth-r iconically by two Horn clauses like

```
nth-r(1,[First|Rest],First).
nth-r(N,[First|Rest],Found) :- Aux is N-1, nth-r(Aux,Rest,Found).
```

Obviously, the two cond clauses of the LISP definition correspond to the two Horn clauses of the PROLOG definition, but the case analysis is performed differently in these languages:

In the base case, LISP employs the function call (eq 1 n) while PROLOG just uses the fixed pattern part 1 iconically in the position of the n parameter.

In the recursive case, LISP employs the always true constant t (better: a positive-integer check on n) and PROLOG uses the always matching variable N (better: a variable with positive-integer type).

Note also that in a more complete LISP definition the l parameter would require an additional non-emptiness check in a very first clause; in PROLOG this is implicit in the subpattern [First|Rest], which iconically shows a list of at least one element, First, and an arbitrary Rest. In addition to the more concise description of the case analysis, the use of iconic patterns also saves selectors in the associated goal conjunctions: in the first case the selector call (car l) becomes the second variable occurrence of First; in the second case the selector call (cdr l) becomes the second variable occurrence of Rest.

2.3. Call-and-definition iconicity

The advantages of the iconic techniques in LISP (nesting) and PROLOG (patterns) can be combined by associating invocation patterns with call nestings. This is not only done in programming languages like SASL, ML, EQLOG, and RELFUN, but already in the mathematical language of recursion equations, where a complete version of our nth example could be written as (list construction is viewed as a right-associative product operator "*")

```
nth(1,f*r) = f
nth(n,f*r) = nth(n-1,r) for n>1
```

Iconic 'pattern-match equations' like these provide a further piece of evidence for the fact that mathematicians have often thought of abstract entities like "the nth factor of a product" in concrete, visual ways (and without auxiliary 'logical variables' for operator results).

Let us end this section with an even more iconic mathematical nth definition, collapsing the earlier two equations into one:

```
nth(n,f1*...*fn*...*fz*nil) = fn
```

Here, the ellipsis operator "..." and variables (f) with variable indexes (n and z) employed in the nth pattern permit a 'visual random access' to the desired element (note that during a match the direct n occurrence and the index occurrence of n in fn must be bound consistently). Such extremely iconic, varying-length ellipsis constructs have only been rarely used in pattern-matching languages, although in many cases they can be implemented via a binary form like the f*r used earlier.

3. Adapters as rules without the 'symbolic link'

The most iconic kind of operator definition considered up to this point employs so-called transformation rules or transformers, each consisting of a pattern linked with a body: This well-known type of rule transforms data via a left-hand-side pattern match followed by a right-hand-side body instantiation. As mentioned earlier, the link itself is arbitrary, hence symbolic, in nature.

Let us now proceed to the issue of eliminating this remaining major non-iconicity from rules. If the structure of a rule pattern is similar to that of the rule body, it is possible to employ adaptation rules or adapters, consisting of a joined pattern/body expression: This new type of rule 'adapts' data via a generalized pattern match that directly applies the operators normally called only on the body side. Thus, the global, symbolic pattern-to-body link is abolished and a local, iconic

embedding of operators in the relevant positions of a pattern is enabled. Besides iconicity, this also enhances parallelization: Essentially, while a transformer links _in sequence_ a recognition part with a separate processing part, an adapter comprises _in parallel_ recognition plus processing aspects as a single unit. This indicates that iconicity (readability) and efficiency (parallelization) can go hand in hand: Both for human and machine interpreters -- rather than representing data by names that transport them from a left-hand side to their operators on a right-hand side -- it appears advantageous to install operators directly on a 'single-hand' side, where their data are expected.

The declarative AI language FIT can be regarded as an attempt to maximize the iconicity and parallelization of operator definitions, by using the non-standard adaptation rules where possible, while keeping the standard transformation rules for adapter-embedded processing and as a fall-back definition method.

_3.1._ _List adapters_

Resuming an earlier list-processing example, the functional version of the nth operator in FIT can be defined thus (the prefixes "?" and ">" mark variables to which exactly one and arbitrarily many values can be assigned, respectively, while "<" fetches variable values indiscriminately):

```
(>(NTH 1 (?FIRST >REST)) <FIRST)
 r(NTH SUB1 CDR)
```

Here, the base case employs a transformer -- written in the assignment form (>pattern body) -- because pattern and body are structurally dissimilar. On the other hand, the recursive case uses an adapter -- in the form of a so-called 'result-reevaluating' or 'reva' (r) adapter -- because the pattern and body of the equivalent transformer

```
(>(NTH       ?N       ?L)
  (NTH (SUB1 <N) (CDR <L)) )
```

are structurally similar, hence can be joined, as shown by the vertical alignment of the pattern variables ?N and ?L with the right-hand-side operator applications (SUB1 <N) and (CDR <L), respectively.

Iconicity is maximized here by eliminating symbolic variable links like ?N --> (SUB1 <N), and introducing direct adapter occurrences of operators like SUB1. Thus the adapter r(NTH SUB1 CDR) iconically mirrors the structure of, and processes on, NTH calls like (NTH 3 '(A B C D)). For this call the adapter matches NTH to itself and simultaneously applies SUB1 and CDR to 3 and (A B C D), respectively, yielding (NTH 2 '(B C D)); this "r" result is evaluated again by the adapter,

yielding (NTH 1 '(C D)), a base case which is then reduced to C by the transformer. Note that in FIT -- unlike in PROLOG -- the order in which rules are written is immaterial, because rule conflicts are resolved on the basis of the "most specific first" principle; in our example the transformer pattern is more specific than the adapter.

Instead of named functions like CDR also anonymous transformers like (TRAFO (?FIRST >REST) @(<REST)) can be embedded into adapters, as in the alternate NTH adapter r(NTH SUB1 (TRAFO (?FIRST >REST) @(<REST))), which only accepts a non-empty list (?FIRST >REST) for replacement by its "@"-instantiated REST list @(<REST).

3.2. Set adapters

A somewhat more complex example is the following definition of SET, a self-normalizing constructor for sets that eliminates duplicates and sorts the remaining elements into a canonical order (for other 'basic collections' such as BAG, HEAP, and STRING see [Boley 1987]).

```
 (SET #ID)
r(SET #ID ?X #ID (COMPOSE AB ?X) #ID)
r(SET #ID (TRAFO (COMPOSE GREATERP
                          ?X (COMPOSE #AB >M) ?Y)
                 <Y <M <X)
     #ID)
```

This definition employs adaptation rules for all three of its cases:

(1) Termination: A constant adapter -- written without an "r" prefix -- returns a SET call on zero or more (#) arbitrary arguments (ID) unchanged if none of the more specific other adapters applies.

(2) Idempotence: One reva adapter mirrors SET calls with a duplicated ?X occurrence in any context (#ID ?X #ID ?X #ID), and directly removes the second occurrence (the ABsorption function is COMPOSEd after the variable ?X).

(3) Commutativity: Another reva adapter mirrors SET calls with an ?X-GREATERP-?Y occurrence, in any context (#ID ?X >M ?Y #ID), and directly exchanges these occurrences (the embedded TRAnsFOrmer COMPOSEs the function GREATERP after the adapter ?X (COMPOSE #AB >M) ?Y in its left-hand side and uses <Y <M <X in its right-hand side).

Let us consider a sample set normalization corresponding to the equality {7,4,1,6,4,6,3} = {1,3,4,6,7}. The trace below shows the purely 'adapter-driven' SET computation, where the arrows "=i=>" mean "derives by rule i" and the set elements focused by the adapters are underlined:

(SET 7 $\underline{4}$ 1 6 $\underline{4}$ 6 3) =2=>
(SET 7 4 1 $\underline{6}$ $\underline{6}$ 3) =2=>
(SET $\underline{7}$ $\underline{4}$ $\underline{1}$ $\underline{6}$ $\underline{3}$) =3=>
(SET $\underline{3}$ $\underline{4}$ $\underline{1}$ 6 7) =3=>
(SET 1 $\underline{4}$ $\underline{3}$ 6 7) =3=>
(SET 1 3 4 6 7) =1=>
(SET 1 3 4 6 7)

It can be seen that in most derivation steps only selected pieces of the set are involved. Of course, the above trace is idealized with respect to non-determinism: in cases of multiple adapter applicability it chooses an optimal one, whereas the real FIT implementation tries them all in a breadth-oriented fashion.

3.3. Graph adapters

As a still more advanced application let us now discuss the definition of BIDSEARCH, a program conducting a bidirectional search on a directed graph. We represent a search problem as a call (BIDSEARCH (start) arc1 ... arcN (goal)), where (start) and (goal) represent the start and goal nodes of the search and the sequence arc1 ... arcN represents a graph with arcI = (fromI toI) representing a directed arc fromI--->toI. The idea is to reinterpret (start) and (goal) as length-one paths and to grow them together from both ends in parallel, until they meet.

```
r(BIDSEARCH (#ID ?X (TRAFO : ^Y))
            #ID ABo(?X ?Y) #ID ABo(?R ?S) #ID
            ((TRAFO : ^R) ?S #ID))

r(BIDSEARCH (#ID ?X (TRAFO : ^Y))
            #ID ABo(?R ?S) #ID ABo(?X ?Y) #ID
            ((TRAFO : ^R) ?S #ID))

(>(BIDSEARCH (>LPATH ?M) >LARCS (?M ?N) >RARCS (?N >RPATH))
 @(<LPATH <M <N <RPATH))

(>(BIDSEARCH (>LPATH ?M) >ARCS (?M >RPATH))
 @(<LPATH <M <RPATH))
```

This definition employs two adapters and two transformers:

(1) Path growing using serial arcs: The first adapter depicts a left path ending with ?X, a right path beginning with ?S, and an arc (?X ?Y) to the left of an arc (?R ?S), and immediately forward-expands the left path by ^Y (this 'temporary value' of Y, obtained through the arc (?X ?Y), is regenerated from the empty sequence by (TRAFO : ^Y)), backward-expands the right path by ^R (analogously regenerated by (TRAFO : ^R)), and absorbs the two used arcs (the function AB is "o"-infix-COMPOSEd with the arc patterns).

(2) Path growing using crossed arcs: The second adapter is similar, but with the arcs (?X ?Y) and (?R ?S) interchanged.

(3) Path meeting using an arc: The first transformer depicts a left path ending with ?M, a right path beginning with ?N, and an arc (?M ?N) acting as a 'missing link', and delivers the concatenated result path (via the instantiation prefix "@").

(4) Path meeting using no arc: The second transformer is similar, but requires no more intermediate arcs because the nodes ?M and ?N coincide.

We consider a sample bidirectional search of a directed path from A to B in the following directed graph:

$$\begin{tikzcd}
A \arrow[rr] & & E \arrow[dl] \\
& C \arrow[ul] \arrow[d, bend left] & \\
B & D \arrow[l] \arrow[u, bend left] &
\end{tikzcd}$$

The trace below shows the combined 'adapter/transformer-driven' BID-SEARCH computation, with the focused nodes underlined:

(BIDSEARCH '($\underline{\text{A}}$) '(C A) '(C D) '(D C) '(E C) '($\underline{\text{A}}$ E) '(D $\underline{\text{B}}$) '($\underline{\text{B}}$)) =1=>
(BIDSEARCH '(A $\underline{\text{E}}$) '(C A) '(C $\underline{\text{D}}$) '(D C) '($\underline{\text{E}}$ C) '($\underline{\text{D}}$ B)) =2=>
(BIDSEARCH '(A E $\underline{\text{C}}$) '(C A) '(D C) '($\underline{\text{C}}$ D B)) =4=>
(A E C D B)

Notice the conciseness of the three-step derivation, due to the bidirectional-parallel graph exploration. This sample problem permits no non-determinism (A and E have only one outgoing arc, B and D have only one incoming arc) except at the point where the two paths meet in C: in addition to the successful rule 4 the real FIT system also tries rule 1 here, but the resulting cycle-path call (BIDSEARCH '(A E C A) '(D C D B)) immediately fails, because no more rule applies. For larger graphs FIT's breadth-oriented search strategy would cause BID-SEARCH to enumerate all paths between the given nodes in the partial order of their lengths.

4. Conclusions

Let us reformulate the main technical points of this paper. Transformation rules as used in forward-chaining (OPS5) and backward-chaining (PROLOG) systems can be more intelligible than the function definitions of lambda-calculus (LISP) systems because the pattern sides of transformers iconically reflect their applicability conditions. However, in many cases iconicity can be further raised by putting the operators from the action side directly into the relevant positions of the pattern side, thus leaving a single recognize/act context.

This can be illustrated in terms of a motion-picture metaphor: Rule-based programming need not employ 'before'/'after' snapshots (transformers) but can also use 'during' exposures (adapters); while the former simulate motion by entirely replacing one picture by a second (even if only a tiny fraction of the picture is affected), the latter move focused parts of a single picture inside a fixed frame. On a closer look, transformation rules like animated cartoons seem to involve an unnatural, jerky technique for forcing change, historically enabled by the relative cheapness of the 'copy' operation in both celluloid and core memory. Adaptation rules are more like fixing a work piece on a workbench and applying tools where appropriate or like putting a liquid crystal into a display and applying electric currents for local restructuring, i.e. they seem to involve a much more natural, smoother technique for causing change.

The predominance of the 'cartoon-like' transformer concept in computer science is highlighted by text editors: As far as we know the string substitution command of all editors is transformer-style. Thus to replace occurrences of 'examplify' by 'exemplify', without wrongly changing 'example' to 'exemple', almost the entire word has to be retyped using a transformation command equivalent to the rule examplify -> exemplify. This is an error-prone procedure and we propose to extend editors like Emacs by an adaptation command permitting rules equivalent to ex(a->e)mplify. Moreover, the adapter concept of FIT, which we also implemented in a depth-first mini version called MUFIT, could be considered as a new, complementary option for existing and developing programming languages (as we envision it for RELFUN).

Not only in programming but also in psychology do we regard the transformer technique as a less plausible model of information processing than the adapter technique: It is evident that states of the mind are not entirely copied in a 'before'/'after' fashion but locally reorganized from moment to moment with most parts of the state keeping normally unaffected.

More generally, this paper has tried to argue that declarativeness is not the only decisive scale on which proposed techniques for complex information processing will be compared. In particular, functional/logical programs can be quite unintelligible if not also

taking good positions on another scale, here called iconicity. It might well happen that newer imperative techniques such as object-oriented programming and even quantitative/qualitative simulation will ultimately rate better on an overall account. The declarative programming community should not leave iconic programming techniques to these other communities alone, but actively look for ways of incorporating iconicity without loosing declarativeness.

Our proposal of adaptation rules is certainly not the only possibility of iconic-declarative programming. Other iconic extensions of functional/logical languages should be explored as well, e.g. via combinations with object-oriented languages (including graphic surface iconicity). Perhaps the present interest in attempts at introducing type hierarchies into functional/logical languages, as exemplified by [Goguen & Meseguer 1986], also derives partly from the iconicity dimension: a SMALLTALK 'class hierarchy', a KRYPTON 'T-box', and a functional/logical 'subsort tree' all act as a direct, central 'image' of the conceptual 'is-a' relationships rather than relying on their indirect, scattered 'encoding' in unary predicates.

References

[Boley 1986] H. Boley: RELFUN: A Relational/Functional Integration with Valued Clauses. SIGPLAN Notices 21(12), Dec. 1986, pp. 87-98

[Boley 1987] H. Boley: FIT: Declarative Programming as Transformer and Adapter Fitting. Univ. Hamburg, FB Informatik, Diss., Aug. 1987

[Goguen & Meseguer 1986] J. Goguen, J. Meseguer: EQLOG: Equality, Types, and Generic Modules for Logic Programming. In: D. DeGroot & G. Lindstrom (Eds.): Logic Programming - Functions, Relations, and Equations. Prentice-Hall, Englewood Cliffs, NJ, 1986, pp. 295-363

[Hillis 1985] W.D. Hillis: The Connection Machine. MIT Press, Cambridge, Mass., 1985

[McDermott 1980] D. McDermott: The PROLOG Phenomenon. SIGART Newsletter, No. 72, July 1980, pp. 16-20

[Minsky & Papert 1971] M. Minsky, S. Papert: On Some Associative, Parallel, and Analog Computations. In: Jacks, E. (Ed.): Associative Information Techniques. New York, 1971, pp. 27-47

[Sloman 1971] A. Sloman: Interactions between Philosophy and Artificial Intelligence: The Role of Intuition and Non-logical Reasoning in Intelligence. Artificial Intelligence 2, 1971, pp. 209-225