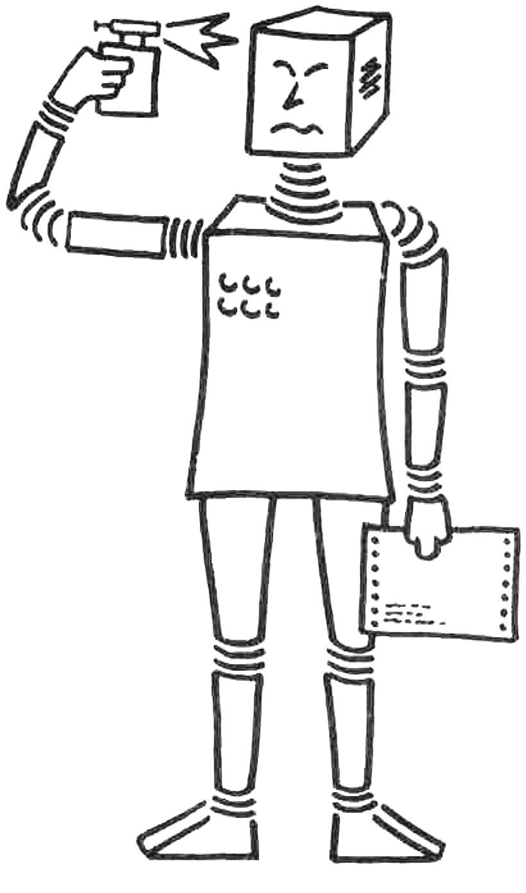
SEKI-PROJEKT

SEKI MEMO

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



FROM PATTERN-DIRECTED TO ADAPTER-DRIVEN COMPUTATION VIA FUNCTION-APPLYING MATCHING

Harold Boley

MEMO SEKI-83-06

FROM PATTERN-DIRECTED TO ADAPTER-DRIVEN COMPUTATION VIA FUNCTION-APPLYING MATCHING°

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

Abstract

The generalization of pattern matching to adapter fitting, as implemented in the programming language FIT, is described semantically. Adapters are like patterns that contain functions which during fitting are applied to corresponding arguments contained in data instances. They are more concise, easier to read, and more efficiently implementable than equivalent LAMBDA expressions and pattern-action rules, because they can analyse data, like patterns, and manipulate them, like functions, in one sweep. Variable settings created by pairing adapter elements with data elements are treated as expressions obeying a consistent-assignment rule, generalizing the usual single-assignment. While pattern-directed computation invokes transformation rules by matching their patterns to an expression and then performing the associated actions, adapter-driven computation only requires adapter/expression fittings. This permits a new representation of AI data bases, LISP functions, hypergraph operations, inference rules (incl. Wang's algorithm), Woods' RTNs, and Turing machines, showing that adapter-driven computation provides an AI-oriented general computational base. The efficiency of pattern-directed and adapter-driven computation is enhanced by introducing the SECURE operator as a functional alternative to PROLOG's cut.

1 INTRODUCTION

The notion of matching has become one of the most important concepts in artificial intelligence (AI), because of the recognition problems that arise on all levels of AI systems. The corresponding AI language feature of pattern matching is a central link between other important very-high-level AI language features like associative data bases, pattern-action rules, and non-determinism (Boley 1983). Moreover, the basic form, which uses a list pattern as a template to test and decompose a list instance, has been generalized along the nearly independent axes of unification, non-list matching, partial matching, and fitting. For a recent survey dealing with the first three generalizations see (Siekmann & Szabo 1982).

Fitting, as first presented at the Hamburg AISB/GI conference in 1978, synthesizes pattern matching (recognition) and function application (action) instead of performing recognition only. Below we will concentrate on list fitting (with a hint on hypergraph fitting), which is implemented in the form of a definitional interpreter of the AI programming language FIT (Boley 1979). Another language using a version of this generalization, albeit a SNOBOL-string-oriented one, is POPLAR (Morris 1982).

The fitting concept of FIT leads to a new view of computational processes, called <u>adapter-driven computation</u>, which permits more concise, easier to read, and more efficiently implementable programs than pattern-directed computation. Unlike some other approaches toward

° To appear in: GI - 13. Jahrestagung, Hamburg, Springer '83

a generalized matcher serving as a framework for formalized reasoning, notably "forced matches" in MERLIN and KRL (Bobrow & Winograd 1977), this matching/computation synthesis is a general yet simple and formal computational base. It can be introduced in three steps.

Pattern-directed computation: Computation is directed by patterns for invoking pattern-action rules or transformers, well-known from production systems (Waterman & Hayes-Roth 1978), AI languages (Hewitt 1972), and logic programming (Clark & Taernlund 1982). Left-hand-side patterns of transformers stored in a data base are matched to the expression to be evaluated and their instantiated right-hand-side actions are used as new expressions to be evaluated by the same process.

Function-applying matching or adapter fitting: Matching is enriched by applying functions inside generalized patterns to arguments inside instances. Elements of a pattern enriched by functions (then called an 'adapter') are paired with corresponding elements of an instance, the pairing results (which may be function applications) are evaluated using the full computational power of the general evaluator, and their values are reconstructed.

Adapter-driven computation: Computation is driven completely by adapters. Adapters stored in a data base are fitted to the expression to be evaluated and the successfully reconstructed expressions are directly used as new expressions to be evaluated by the same process.

Note that we understand the unqualified term 'rule' in its general sense, not necessarily implying a left/right division. Indeed, we are making a case for monolithic adaptation rules (adapters) as a complement to dichotomized transformation rules (transformers). The FIT version of Wang's algorithm shown in the appendix is typical for the mixture of adapters and transformers suited to practical AI programs.

Below we will discuss adapter fitting and adapter-driven computation through comparisons with previous work, systematically developed examples, and evaluation traces that highlight their operational semantics and interpreter realization. FIT programs will use the syntax of the present UCI LISP implementation FIT-1, running on a PDP-10, in which all examples in this paper have been tested. We will simplify internal trace steps (for instance, we omit certain prefix characters where no confusion can arise, in particular quotes) and only touch on the BREADTH/DEPTH expression handling of non-determinism (Boley 1979). For the presentation of the FIT formalism we will adopt an informal style, even when discussing abstract automata, so as to avoid additional meta formalisms that would deter many application-oriented readers.

2 THE CONCEPT OF FITTING

By synthesizing the concepts of applying a function to arguments and of matching a pattern to an instance (in particular, of setting a variable to values), we will arrive at the generalized concept of fitting a 'fitter' to 'fittees'. The syntactical form for evoking fitting, called 'fitment', is (fitter fittee1 ... fitteeN), like that of LISP function application. However, fitter may expand not only to transformer (incl. its named form, function) but also to adapter (incl. the classical pattern (incl. the degenerate forms, variable and constant)); the

_fittee_I's simply denote data for the _fitter_. In order to illustrate the semantics of this abstracted concept, we will begin by discussing simple examples of symbolic computation, involving LISP's 1+Z-ary (Z≥0) integer division function QUOTIENT, defined by (QUOTIENT _n_ _d_1 ... _d_Z) = _n_/(_d_1*...*_d_Z) for Z>0 and (QUOTIENT _n_) = _n_ for Z=0.

The synthesis starts with _predicate functions and patterns_. Consider the recognition of 0 denominators for preventing division by 0 in QUOTIENT calls. In LISP we can define it as a predicate function

```
ZEROD = (LAMBDA (X) (AND (EQ (CAR X) 'QUOTIENT) (MEMBER 0 (CDDR X)))).
```

In FIT we can alternatively define it as a structure-reflecting expression consisting of constants and variables, i.e. as a pattern

```
ZEROD = (QUOTIENT ?N >DL 0 >DR),
```

where ?-variables bind exactly one element and >-variables bind an arbitrary number of elements (incl. zero). A fitment like

```
(ZEROD '(QUOTIENT 256 4 0 8 2))
```

evaluates to a function application ((LAMBDA (X)...) '(QUOTIENT ...)) or to a pattern match

```
('(QUOTIENT ?N >DL 0 >DR) '(QUOTIENT 256 4 0 8 2)),
```

depending on whether ZEROD denotes the predicate function or the pattern. Thus the fitter name ZEROD abstracts from its underlying implementation. When typed to the FIT-1 system, the above pattern match pairs to the expression

```
(LIST (QUOTIENT QUOTIENT) (?N 256) (>DL 4) (0 0) (>DR 8 2)).
```

The pairing results are again fitments to be further evaluated inside the LIST call: (QUOTIENT QUOTIENT) and (0 0) are constant-constant matches returning QUOTIENT and 0, respectively; (?N 256), (>DL 4), and (>DR 8 2) are variable settings yielding the bindings N=256, DL=4, DR=8 2 and returning 256, 4, 8 2, respectively. After the evaluation of these derived fitments, the list (QUOTIENT 256 4 0 8 2) is successfully reconstructed. In general, successful pattern matches return the expression originally used as the fittee (truth-equivalent with 'true') and possibly yield binding side-effects. Failing matches like predicates in FIT yield the failure signal jF (jump 'false').

By analogical reasoning we may now find another fitter subconcept that relates to general functions as patterns relate to predicate functions. These _adapters_ are expressions containing functions (transformers) besides constants and variables. We call adapters using only predicate functions _predicate adapters_ and call other ones _general adapters_.

The synthesis is then completed with _general functions and adapters_. Varying the previous example, consider the transformation of 1 denominators into denominators without the 1 for simplifying QUOTIENT calls. In LISP we can write a general function (presupposing that (REMOVE _x_ _l_) has been defined to remove the element _x_ from the list _l_)

```
ONED = (LAMBDA (X)
        (COND ((AND (EQ (CAR X) 'QUOTIENT) (CDR X))
               (CONS (CAR X) (CONS (CADR X) (REMOVE 1 (CDDR X))))))).
```

In production systems, QLISP, PROLOG, FIT, and many other AI languages, we can use more general, pictorial, and concise pattern-action rules, in FIT represented as TRAFO (<u>tra</u>ns<u>fo</u>rmer) expressions, which are generalized LAMBDA expressions having a pattern in place of a simple list of LAMBDA variables. Using such a transformer the example becomes

```
ONED = (TRAFO (QUOTIENT ?N >DL 1 >DR) (QUOTIENT <N <DL <DR)),
```

where <-variables return their values. However, this example also illustrates some shortcomings with transformer solutions:

1. Transformers are not really concise because many of the constants and variables occur on both sides of such fitters, often even in identical order.

2. Transformers are not optimally readable because the correspondence between elements of both sides of a transformer is difficult to establish: In ONED above, one might overlook the fact that the 1 is elided on the right-hand side; in larger transformers like (TRAFO (T T F T T T F F T T T T F T T T) (T T F T T T F T T T T F F T T T)) this becomes even more of a problem.

3. Transformers entail some inefficiencies because an interpreter for transformers must generate variable bindings during invocation matches of their left-hand sides, somehow transport these bindings to their right-hand sides, and evaluate the right-hand sides in this environment, thus being committed to three mostly sequential computation steps.

FIT's method of overcoming these drawbacks is the use of general adapters, which combine the left-hand sides and right-hand sides of transformers into a new kind of monolithic very-high-level "aggregate operator" in the sense of (Leavenworth & Sammet 1974). In the example this leads to

```
ONED = (QUOTIENT ?N >DL ABo1 >DR),
```

where ABo1 can be regarded as a function primitive that returns the empty sequence if its argument is 1 and fails otherwise. ABo1 is actually composed (using o as the composition infix operator for fitters) of AB = (TRAFO ?X), a function which absorbs (transforms to the empty sequence) one argument ?X, and of 1, a constant which successfully matches, hence returns, only another 1.

If ONED has been bound to this general adapter, the following 7-step fitment evaluation results:

```
1  (ONED '(QUOTIENT 256 4 1 8 2))
2  ('(QUOTIENT ?N >DL ABo1 >DR) '(QUOTIENT 256 4 1 8 2))
3  (LIST (QUOTIENT QUOTIENT) (?N 256) (>DL 4) (ABo1 1) (>DR 8 2))
4a (LIST QUOTIENT 256 4 (AB (1 1)) 8 2)  with N=256, DL=4, DR=8 2
5a (LIST QUOTIENT 256 4 (AB 1) 8 2)                - " -
6a (LIST QUOTIENT 256 4 8 2)                       - " -
7a (QUOTIENT 256 4 8 2)                            - " -
```

The example shows that the shortcomings of transformers disappear in adapter solutions:

1. Adapters are very concise.

2. Adapters are optimally readable, because they eliminate the element correspondence problem: In the ONED adapter, one sees that the 1 will be absorbed; the 16-bit transformer becomes the adapter (T T F T T T F NOToF T T T NOToT F T T T). Furthermore, adapters make it possible to get rid of single-occurrence variables altogether (cf. section 5). This is done by replacing ?-variables by ID and >-variables by #ID, e.g. obtaining ONED = (QUOTIENT ID #ID ABo1 #ID). ID is a function which like the identity returns its single argument but fails for any other number of arguments. # is a prefix operator for fitter repetition (zero or more) related to LISP's MAPCAR by (MAPCAR f l) = ('(#f) l), where f is a function without side-effects and l is a list.

3. Adapters avoid invocation inefficiencies because they require few (theoretically, as shown in section 5, no) variable bindings during invocation fittings and need not transport any bindings to an expression to be instantiated, thus being able to fully exploit the parallelism of the invocation computation.

Like FP programs (Backus 1982) adapters are directly constructed by applying "program-forming operations" (mainly generalized parallel "combination" in the sense of Brainerd and Landweber, a version of which was introduced to programming languages in (Friedman & Wise 1978)) to existing programs (fitters) instead of specifying transformations on "object variables" as done by LAMBDA and TRAFO expressions.

3 GLOBAL EXPRESSIONS AND THE CONSISTENT-ASSIGNMENT RULE

In most LISP-based pattern matchers successful matches return T and yield variable-binding side-effects or they return a list with T in its CAR and an a-list of variable-value pairs in its CDR. In FIT we attempt to combine the handiness of yielding bindings as side-effects with the functional pureness of returning bindings as values. This leads to a new, inherently parallel, semantic foundation of pattern matching (and adapter fitting), completely integrated with functional expression evaluation. For this we allow a successful fitting to return a GLOBAL expression of the form (GLOBAL (b1 ... bM) v1 ... vN), where the bj's are bindings of the form (>variablej valuej,1 ... valuej,Kj) and the vj's are returned values proper. For example, the assignment (>DR 8 2) returns (GLOBAL ((>DR 8 2)) 8 2), where M=1 and N=K1=2. Fittings not performing any variable bindings can be viewed as returning a GLOBAL expression (GLOBAL () v1 ... vN) with an empty binding list which, however, simplifies to the values proper v1 ... vN.

The evaluation of the ONED fitment from step 4 can now be shown more precisely in the manner of pure FIT's semantic FEVAL function:

```
4b (LIST QUOTIENT
         (GLOBAL ((>N 256)) 256)
         (GLOBAL ((>DL 4)) 4)
         (AB (1 1))
         (GLOBAL ((>DR 8 2)) 8 2))
5b (GLOBAL ((>N 256) (>DL 4) (>DR 8 2))
           (LIST QUOTIENT 256 4 (AB 1) 8 2))
6b (GLOBAL ((>N 256) (>DL 4) (>DR 8 2)) (LIST QUOTIENT 256 4 8 2))
7b (GLOBAL ((>N 256) (>DL 4) (>DR 8 2)) (QUOTIENT 256 4 8 2))
```

The evaluation trace shows that GLOBAL expressions which are returned inside another expression migrate out of that expression and on their way up the expression tree are combined (uniting their bindings) until only one GLOBAL remains at the tree's root (5b). The evaluation continues in this GLOBAL's scope and pure FEVAL returns the evaluated GLOBAL (here M=3, N=K1=K2=1, and K3=2) as its value (7b), which impure FIT-1 then splits into binding effects and values proper (7a). The GLOBAL migration and combination can be described by the axiom

```
(... (GLOBAL (B1) V1) . . . (GLOBAL (BZ) VZ) ...) =
(GLOBAL (ubc B1 . . . BZ) (... V1 . . . VZ ...)).
```

Here the Bj's and Vj's are binding and value sequences of the respecitve forms bj,1 ... bj,Mj and vj,1 ... vj,Nj; ubc is a function for uniting bindings consistently (all occurrences of a multiple-occurrence variable require the same value), causing the entire GLOBAL expression to fail (to be reduced to jF) when any inconsistency is detected. This allows context-sensitive checks as in

```
('(QUOTIENT ?X ?X) '(QUOTIENT 8 4)) =>
(LIST (QUOTIENT QUOTIENT) (?X 8) (?X 4)) =>
(LIST QUOTIENT (GLOBAL ((>X 8)) 8) (GLOBAL ((>X 4)) 4)) =>
(GLOBAL (ubc (>X 8) (>X 4)) (LIST QUOTIENT 8 4)) =>
jF.
```

Our integration of pattern matching into a functional framework leads to a generalization of "single-assignment languages" (Tesler & Enea 1968) which may be called 'consistent-assignment languages'. (Interest in single-assignment languages stems from their equivalence with purely functional languages (Backus 1982) and their executability on parallel data-flow machines.) The requirement that only a single assignment to a variable is permitted becomes relaxed to the requirement that all assignments to a variable must be consistent. Single-assignment is a special case of consistent-assignment because, if there is only one assignment to every variable, no inconsistency can occur. Although the consistent-assignment rule allows assignments to a variable already bound to a value, these cannot change the old value because a new value differing from the old one causes the entire evaluation to be aborted with a failure signal jF. Therefore all statements (subexpressions) whose data values are available can be executed simultaneously. In FIT this means that expressions (incl. assignments) inside fitments can be executed using "AND parallelism" (Clark & Taernlund 1982), like the arguments of function applications in pure LISP, resulting in high speed-ups on parallel hardware (Boley 1983).

Using the ˆ-prefix for fetching variable values as soon as they are available, we may have the following evaluation:

```
1  (LIST (LIST (ADD1 ˆX) (0 0) (SUB1 3)) (?X (SUB1 2)) (?X 1))
2  (GLOBAL ((>X 1)) (LIST (LIST (ADD1 ˆX) 0 2) (?X 1) 1))
3  (GLOBAL ((>X 1)) (LIST (LIST (ADD1 1) 0 2) 1 1))
4  (GLOBAL ((>X 1)) ((2 0 2) 1 1))
```

In step 1 the subexpressions (0 0), (SUB1 3), (SUB1 2), and (?X 1) can be evaluated in parallel, the latter generating an X value. In step 2 ˆX and another (?X 1) can be evaluated in parallel, the former using the old value of X and the latter generating a new value for X which happens to be consistent with the old one. Note that the binding generated in the 'upper right' of the expression is used in its 'lower left'. This is possible only because of the parallel subexpression

evaluation which contrasts with the left-to-right EVLIS evaluation performed by ordinary LISP interpreters.

While we regarded the LIST application in step 1 as the initial expression of the above evaluation, it can also be viewed as the result of an adapter fitment in a step 0 (the embedded adapter (ADD1 0 SUB1) causes the embedded LIST application):

```
0  ('((ADD1 0 SUB1) ?XoSUB1 ?X) '((^X 0 3) 2 1))
```

This shows that "output" variables of single-assignment languages such as COMPEL (Tesler & Enea 1968) correspond to "temporary" variables in AI languages such as PLANNER (Hewitt 1972) or FIT, the latter using a PULLTEMPORARY (^) prefix to fetch the temporary variable values being generated during fitment evaluation, which become permanent only if the fitment succeeds; in contrast, the ordinary PULL (<) prefix is used to fetch permanent variable values. (Whereas PLANNER's MATCHLESS marks variables as _receiving_ a temporary or a permanent value, FIT gives all variables temporary values during matches, but allows us to _fetch_ their temporary or their previous permanent value. This is because no reason could be seen why a variable should permanently keep a value it received in a failing match.)

Apart from its application to the semantic foundation and implementation of pattern matching, there are many other interesting uses for the consistent-assignment rule, of which we mention only _conjunctive retrievals_, whose |?-variables, like match variables, are bound temporarily first (the |-prefix allows variables to retrieve their values from the data base without needing an explicit FETCH or GOAL statement). If the assertions (BIG TABLE), (BIG CHAIR), and (RED CHAIR) have been stored in the data base (see section 4 for how to do this), the consistent-assignment rule ensures that an expression containing conjunctive requests like

```
(APPEND '(THE) (BIG |?THING) '(IS A) (RED |?THING))
```

fails with one combination of bindings,

```
(APPEND '(THE)  (GLOBAL ((>THING TABLE)) (BIG TABLE))
        '(IS A) (GLOBAL ((>THING CHAIR)) (RED CHAIR))),
```

but succeeds with the other,

```
(APPEND '(THE)  (GLOBAL ((>THING CHAIR)) (BIG CHAIR))
        '(IS A) (GLOBAL ((>THING CHAIR)) (RED CHAIR))),
```

returning (GLOBAL ((>THING CHAIR)) (THE BIG CHAIR IS A RED CHAIR)).

4 IMPLICIT FITTERS AND ADAPTER-DRIVEN COMPUTATION

Besides explicitly fitting a fitter _fr_ (then called an 'explicit fitter') to a fittee _fe_ using a fitment (_fr_ _fe_), one can also store _fr_ (then called an 'implicit fitter') in a data base and allow it to be implicitly fitted to _fe_ by simply writing _fe_ inside the scope of that data base. If more than one implicit fitter is fittable to a fittee, the partial order of the specifity of their invocation adapters, represented by a nesting of DEPTH and BREADTH expressions, is used as the principal information for conflict-resolution. The resulting

bindings remain local to such implicit fittings. To store M fitters fr1, ..., frM globally, a direct GLOBAL call (GLOBAL (fr1 ... frM)) with an empty returned value can be used.

The explicit/implicit duality holds for all kinds of fitters, i.e. both for transformers and for adapters. For example, transformers such as the ONED TRAFO can not only be used explicitly in fitments like ((TRAFO (QUOTIENT ?N >DL 1 >DR) (QUOTIENT <N <DL <DR)) '(QUOTIENT 256 4 1 8 2)) but can also be stored in the global data base. To do this one might use (GLOBAL ((TRAFO (QUOTIENT ...) (QUOTIENT ...)))), but currently the form of a variable setting (FIT permits arbitrary lists to act as variables), (GLOBAL ((>(QUOTIENT ...) (QUOTIENT ...)))), is still prefered. This has the same effect as the GLOBAL generated by the setting (>(QUOTIENT ?N >DL 1 >DR) '(QUOTIENT <N <DL <DR)) itself. If expressions like (QUOTIENT 256 4 1 8 2) are now typed in the scope of the global data base, the implicit fitter is fitted to these fittees and transforms them in the well-known pattern-directed manner.

What is new, however, is our transfer of this duality to adapters. In particular, the ZEROD pattern can either be used explicitly in fitments like ('(QUOTIENT ?N >DL 0 >DR) '(QUOTIENT 256 4 0 8 2)), as discussed above, or it can be stored in the global data base using (GLOBAL ((QUOTIENT ?N >DL 0 >DR))) and be implicitly matched to fittees like (QUOTIENT 256 4 0 8 2) by simply typing these expressions. (Some 'parenthesis sugaring' and renaming of such special GLOBAL uses would yield the more usual (ASSERT (QUOTIENT ?N >DL 0 >DR)).) Once implicit, the adapter (QUOTIENT ?N >DL 0 >DR) can be viewed as a definition for a function QUOTIENT applied to formal arguments containing a zero divisor. The sample fittee (QUOTIENT 256 4 0 8 2) can be viewed as an application of the QUOTIENT function to actual arguments containing a zero divisor. Like the explicit match, the implicit match returns the fittee; in other words, this QUOTIENT application returns itself, indicating that it contains a zero divisor and preventing the erroneous use of the numeric definition of QUOTIENT. This QUOTIENT definition can also be regarded as an assertion containing variables, as allowed in AI languages like QLISP and PROLOG. When the definition (KNOWS ?X PRESIDENT) has been established, the call (KNOWS FRED PRESIDENT) returns itself to indicate that FRED knows the PRESIDENT, in the same way the QUOTIENT call returns itself to indicate that 256, 4, 0, 8, 2 contains a zero divisor. By using such implicit patterns, predicate functions can be specified in a very-high-level manner. In particular, definitions of list predicates often become more declarative than their LISP and PROLOG correlates. For example, the definition of the MEMBER predicate, which still involves recursion in both LISP and PROLOG, in FIT reduces to the implicit pattern (MEMBER ?X (>L ?X >R)), pictorially showing an ?X-element in the list (>L ?X >R).

Examples of implicit predicate adapters are (COLOR ELEPHANT GREY), (CLOLOR FIRE-HYDRANT RED), etc. (where ELEPHANT, FIRE-HYDRANT, etc. are embedded predicate functions), abbreviating the implicit transformers (Charniak 1981) (TRAFO (COLOR ?ELEPHANT GREY) (ELEPHANT <ELEPHANT)), (TRAFO (COLOR ?FIRE-HYDRANT RED) (FIRE-HYDRANT <FIRE-HYDRANT)), etc. Charniak's request (COLOR CLYDE |?WHAT) is only fitted by the first adapter whose predicate application (ELEPHANT CLYDE) succeeds, not by the many other adapters whose predicate applications (FIRE-HYDRANT CLYDE) etc. fail. In the naive transformer implementation only the right-hand sides discover these failures. Charniak's discrimination net indexing uses ?ELEPHANT, ?FIRE-HYDRANT, etc. as typed variables, so that CLYDE only invokes the ELEPHANT transformer. This, however, makes

the right-hand sides vacuous, showing that predicate adapters, viewable as assertions containing predicates and indexable like transformer patterns, are a natural representation here. Predicate adapters also permit the definition of predicates by recursively using the predicate to be defined in its own invocation. A very concise definition of a PALINDROME predicate consists of a principal definition (%PALINDROME ?X PALINDROME ?X) specifying this kind of invocation recursion and of two definitions (%PALINDROME ?X) and (%PALINDROME) for recursion termination (the %-prefix in front of the 'defined' PALINDROME occurrences restricts them to being constants in contrast to the recursively 'defining' PALINDROME occurrence which acts as a function). Successful evaluations (embedded recursive calls are written inside 'evaluates to' arrows, thus =[...]=>) like

```
(PALINDROME M A D A M) =[ (PALINDROME A D A) =[ (PALINDROME D) ]=>
                          (PALINDROME A (PALINDROME D) A) ]=>
(PALINDROME M (PALINDROME A (PALINDROME D) A) M)
```

do not just return 'true' but give a parse of the recursive palindrome structure.

The ONED general adapter can also be stored in the data base, using (GLOBAL ((QUOTIENT ?N >DL ABo1 >DR))), and be implicitly fitted to fittees like (QUOTIENT 256 4 1 8 2) by typing these expressions. Like the corresponding explicit fitting, this example returns the list (QUOTIENT 256 4 8 2). More generally, a QUOTIENT application to arguments containing a 1 denominator returns this QUOTIENT call as a data value without the 1. This is probably not exactly what is wanted, because, after the simplification of our QUOTIENT application, we would normally like the evaluation to continue, eventually returning the final result 4. For enabling this, the REVA (r) prefix operator is introduced which, applied to an arbitrary fitter, reevaluates the result of its fitting. Thus we can modify the ONED adapter to r(QUOTIENT ?N >DL ABo1 >DR) which, when stored in the data base and used implicitly, drives the desired evaluation (QUOTIENT 256 4 1 8 2) => (QUOTIENT 256 4 8 2) => 4. The evaluation continuing after an implicit fitting specified by a REVA-adapter may, of course, use the same REVA-adapter again, so that a kind of tail-recursion or iteration with varying numbers of arguments emerges. For example, the modified ONED definition can be used iteratively to eliminate an arbitrary number of 1 denominators in QUOTIENT calls as in the adapter-driven computation

```
(QUOTIENT 160 1 5 1 1 4) =>
(QUOTIENT 160 5 1 1 4) =>
(QUOTIENT 160 5 1 4) =>
(QUOTIENT 160 5 4) =>
8.
```

While it is in principle possible to specify all kinds of computation using REVA-adapters for recursion and constant-adapters (i.e. adapters without an r-prefix) for termination (see section 5), it is often convenient to use implicit REVA-adapters for making a function application its own iteration loop and using an implicit transformer for detecting a termination criterion and also returning the result accumulated in one of the arguments.

A simple numeric example of applying this method is an alternate definition of a LISP-like PLUS function (for non-negative arguments) on the basis of ADD1 and SUB1. The definition can be made by typing

(GLOBAL (r(PLUS SUB1 ADD1) (>(PLUS 0 ?X) <X))), where r(PLUS SUB1 ADD1) specifies an iteration loop for simultaneously decrementing the first and incrementing the second argument, while (>(PLUS 0 ?X) <X) detects when the first argument is 0 and the second argument can be returned as the result. A sample evaluation first using the adapter twice and then using the transformer once is (PLUS 2 8) => (PLUS 1 9) => (PLUS 0 10) => 10.

An example involving numbers and lists is the NTH function for selecting the nth list element. It consists of r(NTH SUB1 CDR) and (>(NTH 1 (?X >Y)) <X), enabling evaluations like (NTH 3 '(A B C D)) => (NTH 2 '(B C D)) => (NTH 1 '(C D)) => C.

Finally, a purely non-numeric example is the BOXES operation for transforming directed recursive labelnode hypergraphs into set-like DR<u>L</u>Hs containing all the original labelnode boxes but none of their hyperarc arrows (Boley 1980). The adapter r(BOXES (DRLH >X (TRAFO (TUPLE >Y) <Y) >Z)) uses an embedded TRAFO expression to iteratively replace TUPLEs representing the arrows of DR<u>L</u>Hs by their box contents. When this is no longer applicable, only boxes remain and the less specific transformer (>(BOXES (DRLH >L)) (DRLH <L)) initiates a DRLH collection normalization. This permits evaluations like

```
(BOXES '(DRLH B (TUPLE C D E A) (TUPLE D A D))) =>
(BOXES '(DRLH B C D E A (TUPLE D A D))) =>
(BOXES '(DRLH B C D E A D A D)) =>
(DRLH B C D E A D A D) =>
(DRLH A B C D E).
```

5 ADAPTERS AS A COMPUTATIONAL BASE

Special adapter definitions using the form r(<u>fct</u> <u>fct</u>1 ... <u>fct</u>N) are equivalent to tail-recursive function definitions of the form (>(<u>fct</u> ?<u>arg</u>1 ... ?<u>arg</u>N) (<u>fct</u> (<u>fct</u>1 <<u>arg</u>1) ... (<u>fct</u>N <<u>arg</u>N))), where the <u>arg</u>I's are variables and the <u>fct</u>I's are unary functions (incl. the identity) or function compositions. Thus, making the adapter r(PLUS SUB1 ADD1) implicit is equivalent to defining the function (>(PLUS ?X ?Y) (PLUS (SUB1 <X) (ADD1 <Y))).

There are, however, adapters which cannot be so directly transformed into functions, for instance those performing context-sensitive checks by using multiple-occurrence variables (e.g. FIT's PALINDROME; it <u>can</u> be translated differently of course). On the other hand there are functions which cannot be easily transformed into adapters, e.g. tail-recursive functions using their arguments in different orders in their pattern and in their body (e.g. LISP's REVERSE; it <u>can</u> be translated using PULLTEMPORARY variables inside implicit adapters for example). Instead of considering more of these special transformations we would like to pose the general question whether adapters can be used to represent partial recursive functions or some other universal computational base. (The Church/Turing thesis implies that arbitrary adapters can be represented by partial recursive functions; FIT's FEVAL function shows how this may be done in LISP.) We can answer it affirmatively by showing that adapters <u>without variables and with very simple functions</u> can be used to simulate Turing machines. This will build on a uniform tail-recursive representation of Chomsky-hierarchy abstract automata, including Woods' recursive transition networks.

The tape of each automaton m can be represented as a variable-length sequence of actual arguments to a tail-recursive function defined by adapters. The function name is "scattered" (Boley 1979); more precisely, it consists of a principal prefix m and a subordinate infix s. The state and the head position of an automaton are represented by the naming of s and by its position between the arguments, respectively, where the automaton's head is understood to scan the argument (for RTNs, the arguments) immediately to the right of s. Essentially, the transition function f of an automaton becomes a function f which is applied to state infixes s1 and arguments a1 to their right, returning successor state infixes s2 and possibly new arguments a2. It is called inside an implicit REVA-apdapter r(m ... f #ID) for defining m; the 'repeated identity', #ID, is used to avoid single-occurrence >-variables (in our automata representation there is no need for multiple-occurrence variables). For each final state sz a constant-adapter definition (m ... sz ...) terminates the computation. To initialize m with start state sa, this start state can be used as an additional function name defined by the adapter r((TRAFO : m) sa #ID), where (TRAFO : m) generates m from the empty sequence.

If m is a finite automaton (FA), the REVA-adapter is r(m f #ID) and the constant-adapters are (m sz). If m's transition function f for state s1 and symbol a returns state s2, i.e. f(s1,a)=s2, then an f definition becomes (>(f s1 a) s2).

If m is a recursive transition network (RTN) (Woods 1970), the adapters are the same as in FAs and the f definitions are extended as follows. If m's transition function f for state s1 and a successful embedded call to a subnetwork with initial state sa1 returns state s2, then an f definition becomes (>(f s1 sa1) s2). Thus, the invocation adapter of f's definition contains a call to sa1, which acts as a predicate checking f's applicability (truly recursive calls, i.e. sa1=sa are allowed). A simple example suggestive of the context-free power of such invocation recursions is an RTN for recognizing A↑N B↑N for N≥1. The Woods-like notation of this RTN shown on the left below becomes the FIT definitions on the right, using m=RTN, f=FANBN, sa=ANBN, sz=AN-1BN-1.

```
                ANBN                         r((TRAFO : RTN) ANBN #ID)
                /--\                         r(RTN FANBN #ID)
                |  |                         (RTN AN-1BN-1)
        A       |  v      B                  (>(FANBN ANBN A) AN-1BN)
ANBN ------>  AN-1BN  ------> AN-1BN-1       (>(FANBN AN-1BN ANBN) AN-1BN)
(start)                        (final)       (>(FANBN AN-1BN B) AN-1BN-1)
```

A sample evaluation is (recursion level is reflected by indentation)

```
(ANBN A A A B B B) =>
(RTN ANBN A A A B B B) =>
(RTN AN-1BN A A B B B) =[ (ANBN A A B B) =>
                          (RTN ANBN A A B B) =>
                          (RTN AN-1BN A B B) =[ (ANBN A B) =>
                                                (RTN ANBN A B) =>
                                                (RTN AN-1BN B) =>
                                                (RTN AN-1BN-1) ]=>
                          (RTN AN-1BN B) =>
                          (RTN AN-1BN-1) ]=>
(RTN AN-1BN B) =>
(RTN AN-1BN-1).
```

By embedding RTNs inside patterns, obtaining adapters such as (>W ANBN >W), many context-sensitive languages can be accepted, which can be recognized neither by patterns nor by RTNs alone (thus adapters also synthesize matching and parsing, as desired by Simmons, Wilks, and others). In this way, the use of registers and tests in augmented transition networks (ATNs) can often be avoided. In cases where this is not possible, such features could still be added to this RTN representation to obtain full ATNs (arbitrary test predicates on arcs are implicit in our formalization).

If m is a linear bounded automaton (LBA), the REVA-adapter is r(m #ID f #ID) and the constant-adapters are (m #ID sz #ID). If m's transition function f for state s1 and symbol a1 returns state s2, print symbol a2, and a right move, i.e. f(s1,a1)=(s2,a2,RIGHT), then an f definition is (>(f s1 a1) a2 s2). Analogously, f(s1,a1)=(s2,a2,LEFT) becomes (>(f ?X s1 a1) s2 <X a2).

If m is a Turing machine, the adapter and f definitions are as in LBAs with one addition. There is another REVA-adapter definition r(m #ID g) which applies a function g at the right end of the tape. Thus, m's transition function f remains restricted to a function reading proper symbols. At the tape end, instead of letting f read the BLANK symbol (virtually padded to the right of the proper symbols until actual infinity), another function g is applied to a state argument only, from which it generates a new proper symbol (thus sticking to potentially infinite argument sequences). If m's transition function f for state s1 and the special symbol BLANK returns state s2, print symbol a2, and a right move, i.e. f(s1,BLANK)=(s2,a2,RIGHT), then a g definition is (>(g s1) a2 s2). Analogously, f(s1,BLANK)=(s2,a2,LEFT) becomes (>(g ?X s1) s2 <X a2).

6 FITTER EFFICIENCY AND THE SECURE OPERATOR

Because of the inherent parallelism of fitting, parallel AI architectures (Boley 1983), rather than sequential computers, are ideal adapter machines (cf. section 3). However, although the FIT-1 implementation is written in a purely functional subset of LISP to emphasize semantic clarity, the response times of the fully compiled UCI LISP version running on a von Neumann computer are already sufficient for serious experimentation. Still there are many machine-independent possibilities for further improving the efficiency of adapter-driven computation, even without exploiting imperative LISP.

First, we have studied the 'compilation' of adapters from FIT into LAMBDA expressions of its present implementation language LISP, thus avoiding their repeated FIT interpretation (Boley 1979).

Recent experiments have led us to devise and implement the SECURE operator in the FIT-1 system. This can be viewed as a purely functional alternative to the rule-choice-confirming use of PROLOG's cut operator. Other than cut, SECURE applies to an unordered set of rules, thus keeping the original modularity advantage of production systems, i.e. a programmer can decide whether a new rule is SECURE independently of the data base of rules into which it is to be added. A SECURE (\) mark specifies the safeness of a rule definition (implicit transformer or adapter) such that a use of it can be prioritized to uses of other matching definitions and to other uses of itself. Thus the SECURE prioritization applies to non-determinism arising 1. from several

matching definitions and 2. from several possibilities for matching one definition. (For this also separate SECURE1 and SECURE2 prioritizations could be introduced.) SECURE performs its prioritization during conflict-resolution by changing the order inside priority-ordered DEPTH expressions or changing unordered BREADTH expressions to more efficient DEPTH expressions. Such SECURE-generated DEPTH expressions are a very transparent means for contolling combinatorial explosion in AI programs. The two applications of SECURE's priority assignment are refined as follows.

1a. A SECUREd definition is prioritized to any non-SECUREd definition, independent of its specifity value. It can thus be used to override specifity orderings. For example, (GLOBAL (\(KNOWS ?X PRESIDENT) (KNOWS JOHN MARY))) defines a less specific adapter meaning "Everybody knows the president" with a SECURE mark and a more specific adapter meaning "John knows Mary" without a SECURE mark. After this, (KNOWS JOHN |?WHOM) yields

```
(DEPTH (GLOBAL ((>WHOM PRESIDENT)) (KNOWS JOHN PRESIDENT))
       (GLOBAL ((>WHOM MARY)) (KNOWS JOHN MARY))),
```

i.e. it returns (KNOWS JOHN PRESIDENT) and binds WHOM to PRESIDENT before it returns (KNOWS JOHN MARY) and binds WHOM to MARY. Thus the usual specifity order is inverted, enforcing the early use of less specific, yet explicitly 'privileged', definitions.

1b. A SECUREd definition may become prioritized to other equally specific SECUREd definitions: since all these SECUREd definitions are safe they can be tried in an arbitrary DEPTH order. Continuing the example, after the additional definition (GLOBAL (\(KNOWS ?X POPE))) the localized request (LOCAL (KNOWS JOHN |?WHOM) : <WHOM) through SECURE becomes arbitrarily either (DEPTH PRESIDENT POPE MARY) or (DEPTH POPE PRESIDENT MARY) instead of (DEPTH MARY (BREADTH POPE PRESIDENT)). Similarly, using the rules P3b and P4a in the appendix, (WANG '(AND P Q) ARROW '(OR P Q)) arbitrarily chooses one of the two definitions, say the OR dropping definition P4a, and yields the efficient

```
(DEPTH (WANG '(AND P Q) ARROW P Q)  suspension-for-dropping-the-AND)
```

instead of the inefficient

```
(BREADTH (WANG P Q ARROW '(OR P Q)) (WANG '(AND P Q) ARROW P Q)).
```

2. An invocation match possibility of a SECUREd definition may become prioritized to other equally context-sensitive invocation match possibilities: since all these invocation possibilities of a SECUREd definition are safe they can be tried in an arbitrary DEPTH order. For example, using rule P3b in the appendix, (WANG '(AND P Q) '(AND R S) ARROW P S) arbitrarily chooses one of the two invocation fitting possibilities, say that one dropping the first AND, and yields the efficient

```
(DEPTH (WANG P Q '(AND R S) ARROW P S)
       suspension-for-dropping-the-second-AND)
```

instead of the inefficient

```
(BREADTH (WANG P Q '(AND R S) ARROW P S)
         (WANG '(AND P Q) R S ARROW P S)).
```

As a final means for improving efficiency, domain-dependent knowledge may be applied in the form of "meta rules" (Davis 1980) to make more 'reasonable' use of stored adapters. Like "function-level" reasoning (Backus 1982) adapter-level reasoning is easier than transformer-level reasoning, since meta rules, like programmers, can read from an adapter both when it will be used and what it will do, at one glance. There is no need for a separate examination of a "content" (Davis 1980) part supporting the matching of the pattern part, because form (pattern) and content (action) are united. For example, in the domain of natural numbers, the convergence-seeking meta rule "Use rules applying the predecessor function SUB1 before rules applying the successor function ADD1" can compare adapters of the form

```
r(... SUB1 ...) and
r(... ADD1 ...)
```

more easily than transformers of the form

```
(>(... ?X ...) (... (SUB1 <X) ...)) and
(>(... ?X ...) (... (ADD1 <X) ...)).
```

7 CONCLUSIONS

Although adapters are a general base for computation, they are not intended to replace functions or transformers. Indeed, by their very definition they depend on transformers; furthermore, implicit adapters can be used to define functions in a particularly concise manner. However, adapters give programmers a new dimension of trade-off: For many problems, one can specify transformers that rewrite an entire expression or one can specify structure-reflecting adapters performing localized parallel rewritings in the globally recognized context of an expression.

Valuable insights about this trade-off could be gained while working with the FIT-1 system. Functions which leave the top-level form of data largely intact but change arbitrary parts of their content, as typical for many inference rules (such as about half of the rules in Wang's algorithm) and for operations on DRLHs, frames, and other complex structures, are most naturally expressible as adapters. These are more concise, easier to read, and more efficiently interpretable than corresponding transformer definitions. Often mixtures of adapters (e.g. for recursion) and transformers (e.g. for initialization and termination) seem the best method of defining a function.

Of course, more work will be necessary for further exploring the potentials of adapters. The documented 60 K words of UCI LISP source programs implementing FIT-1 are available from the author. In order to transfer FIT-1 to the VAX-11, it will be transcribed into FRANZ LISP.

8 REFERENCES

Backus, J.: Function-level computing. IEEE spectrum, August 1982, 22-27.

Bobrow, D. & Winograd, T.: An overview of KRL, a knowledge representation language. Cognitive Science 1(1), 1977.

Boley, H.: Five views of FIT programming. Univ. Hamburg, FB Inform., IFI-HH-B-57/79, Sept. 1979.

Boley, H.: Processing directed recursive labelnode hypergraphs with FIT programs. Univ. Hamburg, FB Inform., IFI-HH-M-81/80, Sept. 1980.

Boley, H.: Artificial intelligence languages and machines. Univ. Hamburg, FB Inform., IFI-HH-B-94/82, Dec. 1982. Also in: Technique and Science of Informatics 2(3), May-June 1983.

Charniak, E.: A common representation for problem-solving and language-comprehension information. AI 16, 1981, 225-255.

Clark, K. & Taernlund, S.-A. (Eds.): Logic programming. Academic Press, New York, 1982.

Davis, R.: Content reference: Reasoning about rules. AI 15, 1980, 223-239.

Friedman, D. & Wise, D.: Functional combination. Computer languages, Vol. 3, 31-35, 1978.

Hewitt, C.: Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT, AI-TR-258, April 1972.

Leavenworth, B. & Sammet, J.: An overview of nonprocedural languages. In: B. Leavenworth (Ed.): ACM SIGPLAN symposium on very high level languages. March 1974, Santa Monica, Ca., SIGPLAN Notices 9(4), 1-12.

Morris, J.H.: Real programming in functional languages. In: J. Darlington, P. Henderson & D. Turner (Eds.): Functional programming and its applications. Cambridge University Press, 1982.

Siekmann, J. & Szabo, P.: Universal unification. In: W. Wahlster (Ed.): GWAI-82, Bad Honnef, Sept. 1982, Informatik-Fachberichte 58, Springer-Verlag, 102-141.

Tesler, L.G. & Enea, H.J.: A language design for concurrent processes. AFIPS Conference Proceedings, SJCC, Vol. 32, 1968, 403-408.

Wang, H.: Toward mechanical mathematics. IBM Journal of Research and Development 4(1), Jan. 1960, 2-22.

Waterman, D. & Hayes-Roth, F. (Eds.): Pattern-directed inference systems. Academic Press, New York, 1978.

Woods, W.A.: Transition network grammars for natural language analysis. CACM 13(10), October 1970, 591-606.

9 APPENDIX: WANG'S ALGORITHM IN FIT

"Sequents" <u>antecedent</u> -> <u>consequent</u> (Wang 1960) are represented as expressions of the form (WANG <u>antecedent</u> ARROW <u>consequent</u>), where <u>antecedent</u> (<u>consequent</u>) is a sequence of implicitly conjuncted (disjuncted) formulas. Wang's original rule labels are used for easy comparison; without loss of generality the rules P5a-P6b are omitted. His 'and' is represented by ANDTH, FIT's version of LISP's AND. Exploiting a Church-Rosser property, Wang always eliminates leftmost connectives while our SECURE (\) marks do this by allowing the rules to eliminate arbitrary connectives (see section 6).

```
P1. \(WANG #ID ?X #ID ARROW #ID ?X #ID); ?X on both ARROW sides: proven
P2a. r\(WANG (TRAFO : ^X) #ID ARROW #ID ABo(NOT ?X) #ID); ?X inside NOT
P2b. r\(WANG #ID ABo(NOT ?X) #ID ARROW #ID (TRAFO : ^X)); crosses ARROW
P3a. (>\(WANG >L ARROW >R1 (AND ?X ?Y) >R2); consequent AND gives ANDTH
       (ANDTH (WANG <L ARROW <R1 <X <R2) (WANG <L ARROW <R1 <Y <R2)))
P3b. r\(WANG #ID (TRAFO (AND ?X ?Y) <X <Y) #ID ARROW #ID); drop ant.AND
P4a. r\(WANG #ID ARROW #ID (TRAFO (OR ?X ?Y) <X <Y) #ID); drop conse.OR
P4b. (>\(WANG >L1 (OR ?X ?Y) >L2 ARROW >R);  antecedent OR gives ANDTH
       (ANDTH (WANG <L1 <X <L2 ARROW <R) (WANG <L1 <Y <L2 ARROW <R)))
```