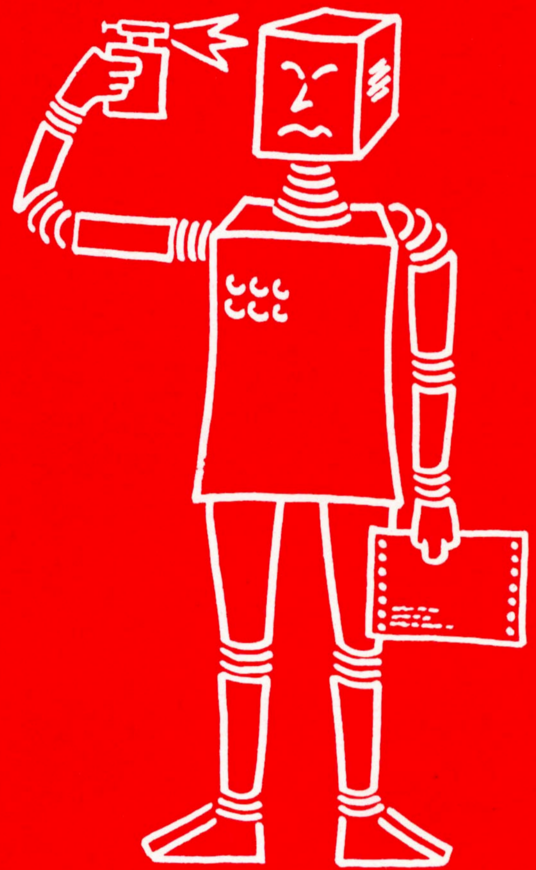
SEKI-PROJEKT

SEKI MEMO

Institut für Informatik III
Universität Bonn
Bertha-von-Suttner-Platz 6
D 5300 Bonn 1, W. Germany

Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1, W. Germany



MEMO SEKI-BN-81-01

APE: An Expert System for Automatic Programming from Abstract Specifications of Data Types and Algorithms

by

U. Bartels, W. Olthoff, and P. Raulefs

APE: An Expert System for Automatic Programming from Abstract Specifications of Data Types and Algorithms

Ulrich Bartels, Walter Olthoff, and Peter Raulefs

SEKI-PROJEKT
Institut für Informatik III
Universität Bonn
Postfach 2220
D-5300 Bonn 1, West Germany

and

Institut für Software-Technologie
Gesellschaft für Mathematik und Datenverarbeitung mbH
Schloss Birlinghoven, D-5205 St.Augustin 1, West Germany

Abstract

The APE (Automatic Programming Expert) system constructs executable and efficient programs from

- algebraic specifications of abstract data types, and
- abstract algorithms given as conditional term-rewrite-rule-systems with terms built up from operation symbols of the abstract data types involved.

The APE is an experimental system devised to develop methods for codifying a rather broad extent of programming knowledge required to construct implementations of data types and algorithms.

For data type specifications, the APE admits hidden operations, conditional axioms, and parameterized data types. The APE automatically implements algebraic specifications of all commonly known data types in terms of clusters of INTERLISP-functions. The APE constructs executable implementations of a variety of sorting and searching algorithms.

As an experimental prototype, the APE demonstrates that a **knowledge-based programming paradigm** provides a useful tool for partially automating an important phase of software development.

Keywords and Phrases. Abstract data types, automatic programming, codification of programming knowledge, expert systems, knowledge representation, production systems.

1. Introduction

A standard paradigm of software design is to start with abstract specifications, and to gradually expand, refine, and transform them to an appropriate level to obtain efficient implementations. This approach factors software design into the constructions of abstract specifications, and the development of more concrete representations. Such a factorization entails a corresponding structuring of design, validation, adaption, and maintenance of software systems.

Algebraic specifications [ADJ 76, HR 80] of abstract data types (ADTs) allow to specify data types by characterizing the behavior of data objects under characteristic operations without resorting to particular representations. Hence, developing algorithms at the level of manipulating objects of algebraically specified ADTs provides an optimum of freedom from specific specifications, facilitates the manipulation of algorithms (program transformations), and allows proving properties of algorithms independent of representation details. The remaining task consists in implementing such **abstract algorithms** in terms of appropriate representations supported at the machine level.

Implementing an ADT-specification requires

- to design **concrete representations** for encoding abstract data objects in them.
- to design algorithms **implementing** abstract operations on concrete representations.

For an ADT-specification, (1) a concrete representation and (2) algorithms implementing all abstract operations on concrete representations together form a **concrete impleuentation** of ADT-specifications. To be run on computers, concrete representations must be expressed in terms of an executable programming language. Hence, concrete representations must be designed in terms of the data structures supported by the programming language.

Given a programming language, developing a concrete implementation from an abstract algorithm usually consists of a multitude of only partially interrelated design decisions based on insight, experience, and training. Many such decisions are isolated in that they are taken from different domains of knowledge, i.e. there are no well-structured transformation algorithms.

We call knowledge contributing to develop concrete implementations from abstract algorithms **programming knowledge.** The nature of programming knowledge suggests to encode it in terms of **production systems** [DK 77]. A **production rule** encodes an isolated piece of knowledge. A production system consists of a **production base** containing production rules, and a **production control** for sequencing the application of production rules. Systems of production systems are an appropriate tool for both encoding and utilizing loosely interrelated pieces of knowledge.

Production systems are particularly well suited for being developed experimentally: Start with an initial set of productions, and try to apply them to typical examples. If the initial production system does not work, analyze what goes wrong, and add resp. change production rules, or adjust the production control. A resulting production system may the incorporate all domain-specific knowledge of those who have developed it.

Among other approaches to automatic programming, two systems have been developed along a similar paradigm:

(1) DAEDALUS [MW 79] is a rule-based system for deriving programs from descriptions of input/output-relations, giving no informations about algorithms to be employed. The main contribution of DAEDALUS are formation-rules for control-structures (conditional-, recursion-, procedure-formation), and a generalization rule (for related results, see [BI 79]).

(2) PECOS [BA 78a,b], the coding expert of the PSI-system [GR 76], generates Lisp-programs from algorithms given as programs in a high-level language admitting some reasonably "abstract" data structures (such as sets). In the PSI-system, PECOS is meant to be supported by an efficiency expert [KA 79] embodying knowledge about the efficiency of various representations of data structures.

The problem of generating concrete code from algebraic ADT-specifications and abstract algorithms has not been attacked before. Compared with what is achieved by PECOS, this is a much more challenging task as we start from the most abstract level (axiomatizations) of specifying data types.

We present the APE system (Automatic Programming Expert) which consists of two subsystems:

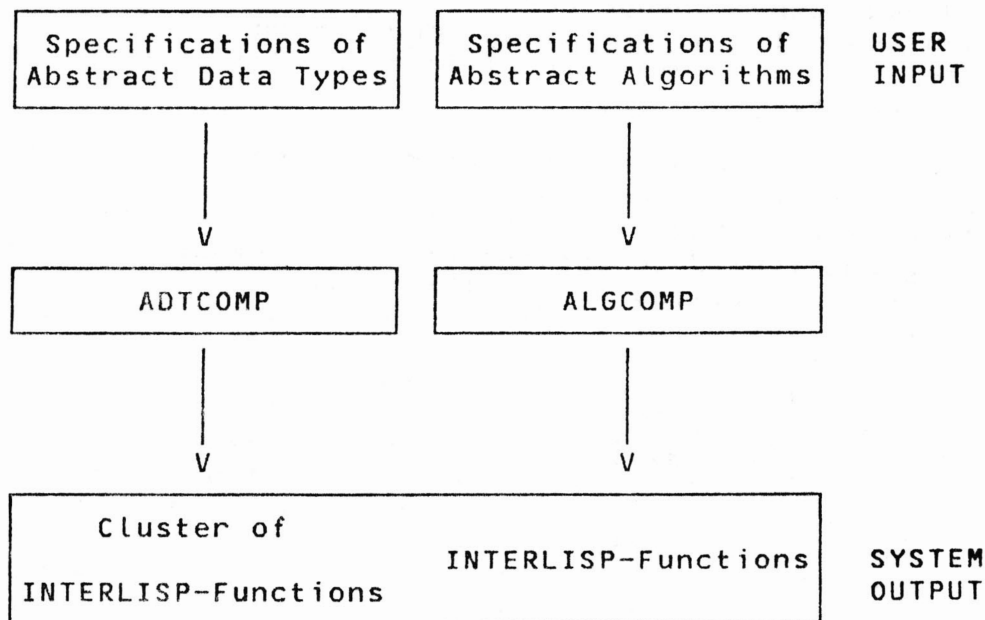
- ADTCOMP codes algebraic specifications of abstract data types as executable INTERLISP-programs.
- ALGCOMP constructs executable INTERLISP-programs from abstract algorithms given in terms of conditional term-rewrite rules with terms made up from operation symbols of abstract data types being implemented by ADTCOMP.

Section 2 surveys the structure of the APE. The specification language for abstract data types and abstract algorithms is outlined in Section 3. The way the APE works is explained in Sections 4 and 5, illustrated by detailed walks through the implementation of an abstract data type specification and an abstract program. The accomplishments of the system in its present state of development are summarized in Section 6.

Further details on the ADTCOMP subsystem are given in [EKRT 80a,80b]. The APE is implemented in INTERLISP and is fully operational. The system with detailed documentation (in German) is available from the third author.

2. System Structure

The structure of the APE is summarized in the following illustration:



In both ADTCOMP and ALGCOMP, programming knowledge is codified in terms of production rules. Both subsystems contain **rule manipulation packages** providing an extensive interface for interactive experiments and modifications of the production bases. As has been experienced in the development of other expert systems, knowledge acquisition is the most serious problem. As we do not see a feasible way of acquiring programming knowledge automatically, an extensive dialogue and editing support turned out to be of crucial importance in the evolution of our system.

A **production system** consists of

- a **production base** containing **production rules**
- a **production control** guiding the selection of production rules.
- a **data base** to which production rules are applied.

Production rules operate on a data base. A production rule is of the form <test> -> <action>.
<test> is evaluated to a predicate which is satisfied or not by the data base. If the <test> is satisfied by the data base, the rule is **applied** by executing <action> on the data base. The effect of <action > is to possibly alter the data base. In many production systems , production rules may also change the production base (e.g. by adding/deleting production rules), or the production control. In APE, however, production rules only affect the data base.

The production control selects production rules to be checked for applicability and subsequent application. In the simplest

form of a production system, the production base is just a list of production rules. and the production control moves through the list from one rule to the next. Initially, our production control used **agendas.** An agenda is a control program for sequencing the order of considering production rules, e.g. a regular expression of names of production rules and intermediate tests for branching. In the evolution of our system, however, more and more "control knowledge" in agendas was transferred to the organization of production bases, and our present system does not use agendas at all any more.

3.Specification Language

Specifications to the APE consists of

- **algebraic specifications** of all data types involved, and
- a set of production systems each of which specify an **abstract algorithm** .

The production rules of an abstract algorithm consist of terms and assertions built from operation symbols and sorted variables of the algebraic data type specification together with operation symbols denoting the algorithm and auxiliary algorithms being specified.

3.1. Algebraic Data Type Specifications

Abstract data type specifications are based on the algebraic specification technique with initial algebra semantics as pioneered by the ADJ-group [ADJ 76]. In its simplest form, an algebraic specification consists of a **signature** (S,Σ) and a set E of **equations** . A signature (S,Σ) establishes a syntax for denoting objects of the data type, and consists of

- a set S of **sorts** . Sorts are names for sets of objects belonging to the data type.
  Example: S = {olist,int,atom} is a set of three sorts for the open list abstraction. olist is a name for the set of all open lists consisting of atoms, and int, atom are names for the sets of all natural numbers resp. atoms.
- a set Σ of **operation symbols** . Taking each sort $s \in S$ to be a name of a set D.s of data objects, an operation symbol in Σ is the name of a function among appropriate sets of data objects. To specify domain and range for each operation symbol $f \in \Sigma$, an **arity** is ascribed to f: $arity(f) = (s_1 \ldots s_n, s)$ for sorts $s_1,\ldots,s_n,s \in S$ iff f denotes a function $f.D\colon D.s_1 \times \ldots \times D.s \rightarrow D.s$. This is written as $f\colon(s_1 \ldots s_n,s)$. A nullary operation symbol $c\colon(\Lambda,s)$ denotes a value in D.s.

Example: Σ = { add: (olist nat atom,olist),
remove: (olist nat, olist),
read: (olist nat,atom),
constr: (olist atom,olist),
EMPTY: olist,
UNDEF: olist,atom }
is a set of operation symbols for the open list data abstraction.

Given a signature (S,Σ), we require that for each sort $s \in S$ there is an infinite set X.s of variable symbols s.t. for different sorts s_1 and s_2 in S, $X.s_1$ and $X.s_2$ are disjoint. From operation and variable symbols we can construct **(S,Σ)-terms** by observing arities:

The set $T.\Sigma$ of all (S,Σ)-terms is defined to be the least family $T.\Sigma = (T.\Sigma s | s \in S)$ s.t.

(1) $\forall s \in S.\ X.s \cup \Sigma.s \subset T.\Sigma s$
(Σs stands for constants of sort s).
(2) $\forall n \in NAT.\ \forall s_1,\ldots,s_n \in S.\ \forall t_1 \in T.\Sigma s_1,\ldots,t_n \in T.\Sigma s_n$.
$\forall f:(s_1 \ldots s_n,s) \in \Sigma.\ f(t_1,\ldots,t_n) \in T.\Sigma s$.

An **(S,Σ)-equation** p=q is a pair of (S,Σ)-terms of same sort.
A **conditional (S,Σ)-equation**

q = **if** p = p' **then** r_1 **else** r_2 **fi**

consists of a condition p = p' and alternatives r_1 and r_2 s.t. p,p' and q, r_1,r_2 are terms of same sorts, respectively.

```
Example: E = { add(empty,N,X)        = if N=1
                                       then constr(EMPTY,X)
                                       else UNDEF fi ,
               add(constr(L,X),N,Y)  = if N=1
                                       then constr(constr(L,X),Y)
                                       else constr(add(L,N-1,X),Y) fi ,
               remove(EMPTY,X)       = UNDEF,
               remove(constr(L,X),N) = if N=1
                                       then L
                                       else constr(remove(L,N-1),X) fi ,
               read(EMPTY,X)         = UNDEF ,
               read(constr(L,X),N)   = if N=1
                                       then X
                                       else read(L,N-1) fi }
```
is a set of (S,Σ)-equations for the open list data abstraction.

This specification is an example of a specification containing a **hidden operation** constr which is not accessible to a user, but serves definitional purposes (establishing a congruence) only.

A data type specification as an input to the APE consists of the three sets S,Σ and E of sorts, operations (with arities), and equations, with an indication which operations are hidden. Such a specification is given a name, e.g. OLIST for our above example. Then, the command (IMPLEMENT 'OLIST) causes the APE

to construct a bundle of LISP-functions implementing the algebraic data type specification (see Sect.4).
A parameterized data type specification PSPEC is a pair PSPEC = (SPEC0,SPEC1) s.t. PSPEC transforms any SPEC0-algebra A to a SPEC1-algebra containing A as a subalgebra (see e.g. [ADJ 80]). In fact, most data types are parameterized. The command (PARAMETERIZE 'OLIST 'SPEC) causes the APE to construct a LISP-implementation of the open list abstraction s.t. the atoms stored in an open list are LISP-implementations belonging to the data type specified by SPEC.

3.2. Specification of Abstract Algorithms

Abstract algorithms are specified in terms of the following syntax:

```
<abstract-algorithm> ::= <header><prod-system><comment>
<header>             ::= abstract algorithm
                         <fcn-name><formal-params-dcl>
                         { relations <relation> }
                         { preconditions <assertion> }
                         { external <fcn-name> }
<prod-system>        ::= prodsystem
                         <fcn-name><formal-params-list> =
                          set-of-<production-rule>
<production-rule>    ::= <assertion> -> <term>
<comment>            ::= (some explanatory text)
```

<header> introduces

- the name of the algorithm (a function name)
- declarations for formal parameters consisting of a sort name, an identifier, and an optional initial value. Formal parameters are partitioned into
  - user accessible parameter positions (must be supplied with actual parameters)
  - internal parameter positions (for internal recursive calls) which must not be supplied with actual parameters in a user call (internal initialization)
- preconditions on actual parameters (for user accessible parameter positions only)
- references to the external algorithms invoked in this algorithm.

A <prod-system> is a production system consisting of a set of production rules, i.e. test/action-pairs where tests are boolean terms, and actions are terms s.t. all terms are constructed from formal parameters and operation/function symbols of any of the abstract data types involved, the name of the abstract algorithm (a recursive application) and names of external algorithms.

Example:

```
abstract algorithm BUBBLE-SORT;

parameters          user      OPENLIST[INT] OL;
                    internal  INT INDEX := 1,
                              BOOL BUBBLE? := FALSE;

external            CHANGE;

prodsystem BUBBLE-SORT (OL,INDEX,BUBBLE?) =
(1) { read (OL,INDEX+1) ≠ UNDEF and
      read (OL,INDEX) ≤ read (OL,INDEX+1)
    ——> BUBBLE-SORT (OL,INDEX+1,BUBBLE?);
(2)   read (OL,INDEX+1) ≠ UNDEF and
      read (OL,INDEX) > read (OL,INDEX+1)
    ——> BUBBLE-SORT(CHANGE(OL,INDEX,INDEX+1),INDEX,TRUE);
(3)   read (OL,INDEX+1) = UNDEF and BUBBLE? = TRUE
    ——> BUBBLE-SORT(OL,1,FALSE);
(4)   read (OL,INDEX+1) = UNDEF and BUBBLE? = FALSE
    ——> OL }

comment BUBBLE-SORT(OL) replaces the open list bound to OL
        with its sorted version!!
```

BUBBLE-SORT invokes the auxiliary algorithm CHANGE which is specified as follows:

```
abstract algorithm CHANGE;

parameters          user OPENLIST OL,
                         INTEGER INDEX1,
                         INTEGER INDEX2;

preconditions       read(OL,INDEX1) ≠ UNDEF,
                    read(OL,INDEX2) ≠ UNDEF;

prodsystem CHANGE (OL,INDEX1,INDEX2) =
(1) { INDEX1 ≤ INDEX2
    ——>  let OL := add(OL,INDEX1,read(OL,INDEX2)) in
         let OL := add(OL,INDEX2+1,read(OL,INDEX1+1)) in
         let OL := remove(OL,INDEX1+1) in
         let OL := remove(OL,INDEX2+1) in OL;
(2)   INDE[illegible]   INDEX2
    ——> CHANGE(OL,INDEX2,INDEX1) }

comment CHANGE(OL,INDEX1,INDEX2) exchanges the INDEX1st and
        INDEX2nd element of the open list bound to OL!!
```

4. Automatic Coding of Algebraic Data Type Specifications

4.1. System Structure

Upon encountering an abstract data type specification (possibly parameterized), APE works through seven phases that successively construct a LISP-implementation of the data type. Except for the Initialization- and LISP-phase, each phase incorporates specific programming knowledge and refines the intermediate constructions obtained from the previous phase.

These seven phases are:

(1) Initialization (I-) Phase. The I-phase converts the input specification into a LISP-representation.
(2) Functionality (->-) Phase. The ->-phase considers the signature of the input specification only. Conclusions from the operations' functionalities are drawn, and the structure of LISP-functions implementing these operations is established.
(3) Axiom (=-) Phase. The =-phase considers the axioms of the input specification only, and make inferences from observations on the axioms.
After the =-phase, the input specification is no longer considered.
(4) Representation (R-) Phase. The R-phase combines conclusions of the previous phases to determine the data structure in which objects of the data type are represented.
(5) Compile (C-) Phase. Using all information obtained so far, the C-phase constructs actual LISP-code to make optimizations and improvements.
(6) Cleanup Phase. The Cleanup-phase refines and/or rearranges the LISP-code obtained from the C-phase to make optimizations and improvements.
(7) LISP Phase. The LISP-phase constructs the final LISP-code. As LISP does not support a module-construct such as SIMULA-class or ALPHARD-form, a corresponding mechanism is built up in the LISP-phase.

Phases (2)-(6), which actually apply programming knowledge, are production systems. Application of production rules is controlled by meta-rules and attention focussing. These mechanisms and pattern matching are not discussed in this paper.

4.2. A Walk through the Automatic Implementation of an Abstract Data Type Specification

To explain how the APE works out an implementation, we present a walk through the implementation of the algebraic specification of the data type OLIST (open list) as given in section 3.1. First, we show what the APE is doing in response to the command (IMPLEMENT 'OLIST), and then we briefly explain how

the data type open list is implemented as a parameterized data type.

Initialization (I-) Phase. The input specification is converted into an internal LISP-representation, and the LISP-atoms D,X,N,ADD,REMOVE,READ, and CONSTR are associated with the following properties:

D	NAME	OLIST	{D is the internal data type name bound to the external name OLIST}
ATOM	PARAMTYPE	T	{ATOM and NAT are parameter types}
NAT	PARAMTYPE	T	
X	SORT	ATOM	{X and N are identifiers of sort ATOM resp. NAT}
N	SORT	NAT	
ADD	OP	(IMPLEM)	{ADD,REMOVE, and READ are accessible operations to be implemented, and CONSTR is a hidden operation which needs not be implemented}
REMOVE	OP	(IMPLEM)	
READ	OP	(IMPLEM)	
CONSTR	OP	(HIDDEN)	

Functionality (->-) Phase. ->-rules check functionalities of operations to make useful observations. In our example, rules ->01,->02A,->03, and ->08 apply:

```
->01.    if *OP1 maps a D-object and possibly a
            *Y-object to a D-object
            and *Y is not D
         then *OP1 is a singly continuable operation
```

```
->01:Result.  ADD     CONTINUABLE  (SINGLE)
              REMOVE  CONTINUABLE  (SINGLE)
              CONSTR  CONTINUABLE  (SINGLE)
```

Note. Match variables begin with an asterisk.
An operation op is **n-fold continuable** iff op constructs from n D-objects another D-object.

```
->02A.   if *OP1 maps D-objects and possibly other
            arguments to a parameter sort *S
         then *OP1 reads contents out of D-objects
```

```
->02A:Result. READ READS
```

```
->03.    if *OP1 works on D-objects using
            another parameter of sort *S≠D
       then *S is a sort of the elements of which
            serve as selection criteria .
---------------------------------------------------
->03:Result.     ADD SELECT-CRIT (NAT)
              REMOVE SELECT-CRIT (NAT)
                READ SELECT-CRIT (NAT)
                   D SELECT-CRIT (NAT)
```

Note. If by rule ->03 a selection criterion is applied to D-objects, ->03 not only ascribes this information as a property to the relevant operations, but to D, too. This is because the occurrence of a selection criterion heavily influences the choice of the representation of D-objects in the R-phase.

Rule ->08 establishes the structure of the LISP-functions that will be refined to implement the operations, and results in:

```
->08:Result.     ADD FUNCT ((LAMBDA (D N X) FORM D))
              REMOVE FUNCT ((LAMBDA (D N) FORM D))
                READ FUNCT ((LAMBDA (D N) FORM ))
```

Axiom (=-) Phase. =-rules analyse the axioms of a specificaton.

The first rule detects the elementary construction operator and determines which operations extract constituents from terms built up by the elementary construction operator.

```
=01.    if *OP1 is a read operation extracting
           elementary information from a term constructed
           with operation *OP2
      then *OP2 writes elementary objects into objects
           *OP1 reads from which is noted by binding
           *OP2 to *E.
---------------------------------------------------
=01:Result.    READ READS    (IF(EQ N 1))
             CONSTR WRITES   (ELEM)
                 *E DENOTES  (CONSTR)
```

=03 detects all error exceptions and results in:

```
=03:Result.     ADD UNDEF (IF(EQ D DO) and (NEQ A 1))
             REMOVE UNDEF (IF(EQ D DO))
               READ UNDEF (IF(EQ D DO))
```

=04 detects operations which work recursively through the

structure of data objects.

```
=04.   if the elementary construction operation is
          singly continuable,
          and *OP1 moves inside *E-terms,
          and *OP1 is not an elementary instruction opera-
              tion
       then *OP1 works recursively through D-objects
```

```
=04:Result.    ADD WORKSRECTHRU (IF(NEQ N 1))
            REMOVE WORKSRECTHRU (IF(NEQ N 1))
              READ WORKSRECTHRU (IF(NEQ N 1))
```

Rule =06 observes which operations write and remove elements from D-objects, and results in:

```
=06:Result.    ADD WRITES   (IF (EQ N 1))
            REMOVE DELETES  (IF (EQ N 1))
```

Rule =10 detects selection criteria and observes how operations use selection criteria.

```
=10:Result.   N SELECTCRIT (COUNTER)
            ADD COUNTS      (IF (NEQ A 1))
```

Representation (R-) Phase. There is a wealth of conclusions which can be obtained from the results of the previous phase. First, rule R01 decides to implement data types in terms of lists if their elements are constructed by a singly continuable construction operation:

```
R01:Result.   D  LIST
```

For operations working recursively through data objects and simultaneously keeping counts, rule R06 establishes that their action affects the NEWEST element which had been added by the elementary construction operation.

```
R06.    if *OP WORKSRECTHRU IF *CONDITION
           and *OP COUNTS IF *CONDITION
       then *OP COUNTS NEWEST IF *CONDITION
```

```
R06:Result. REMOVE COUNTS (NEWEST IF (NEQ N 1))
               ADD COUNTS (NEWEST IF (NEQ N 1))
              READ COUNTS (NEWEST IF (NEQ N 1))
```

Although R01 has decided that open lists should be implemented as lists, it is still open what particular kind of list structure (such as singly or doubly linked lists) should be selected. For example, the doubly linked list structure is chosen when reading is done at one "end" of data objects, and a counting operation recursively works through the list from the other end. To prepare this decision, we have to find out whether operations work at the front rsp. backend.

```
R11B.    if D is to be implemented as a list but
            no direction has been established yet,
            and *OP is an operation to be implemented,
            and *OP COUNTS the NEWEST (resp. OLDEST)
                element of D-objects
                (possibly under *CONDITION)
       then *OP COUNTS at the FRONT end of D-objects,
            and lists representing D-objects
                are directed now.

R11B:Result.    D DIRECTED
             READ COUNTS   (FRONT IF (NEQ N 1))
```

Rule R13B establishes the position at which the elementary construction operation bound to *E inserts new elements, and R15B works simularly for counting operations:

```
R13B:Result. CONSTR WRITES (FRONT)
R15B:Result. REMOVE COUNTS (FRONT IF (NEQ N 1))
                ADD COUNTS (FRONT IF (NEQ N 1))
```

We have now collected all information which makes the next rule choose the singly linked list structure to represent D-objects:

```
R21.    if D-objects are represented as lists
           and there is no operation to be implemented
               which deletes or counts at the back end
               of lists
      then D-objects are represented as singly linked lists.

R21:Result.    D  LIST  (SL)
```

Finally, R29 establishes the LISP-representation of the basic D-object DO (the empty open list):

```
R29:Result.  DO  FUNCT (NLAMBDA (FRONT)
                        (SETQ FRONT (CONS))
                        (RPLACA FRONT FRONT))
```

Compile (C-) Phase. C-rules work on the FUNCT-property of operation names which is a pattern of the LISP-function to implement the respective operation. All strings in such a FUNCT-pattern that are not LISP-atoms are regarded as keywords to be **expanded** to LISP-code by C-rules. C-rules work from front to back ends of FUNCT-patterns. A C-rule inspects the next keyword after the current position. The ***-sign indicates the current position pointer. We demonstrate the Compile-phase by showing how the operation ADD is implemented.
Note that the Functionality Phase has already established the FUNCT-pattern (LAMBDA (D N X) FORM D) for ADD, and the current position pointer is now initialized to stand immediately behind the formal LAMBDA-parameters.

```
C02.    if D is a list and *OP does
           *ACTION∈{READS,DELETES,WRITES} at the
           *POSITION∈{FRONT,BACK}end,
           and *OP does not *ACTION *PLACE,
            or *OP ACTION IF *CONDITION1,
            or *OP DELETES IF *CONDITION2,
            or *OP WORKSRECTHRU IF *CONDITION3,
            or *OP COUNTS FRONT IF *CONDITION4
           {there are more conditions on logical
            connections between *CONDITIONs}
      then in the FUNCT-pattern of *OP,
           FORM is replaced with
           (COND UNDETERMINED (*CONDITION1 *ACTION FRONT)
                              (CONDITION2 DELETES FRONT)
                              (T (APPLY* *OP FRONT PARAMS)))

C02:Result. ADD FUNCT((LAMBDA (D N X) ***
                       (COND UNDETERMINED
                          ((EQ N 1)WRITES FRONT)
                          (T (APPLY* ADD FRONT PARAMS)))D))
```

Seventeen applications of C-rules yield the final result of the Compile-Phase for ADD:

```
ADD FUNCT <LAMBDA(D N X)
            (COND((AND (NOT (CDR D))(NOT (EQ N 1)))
                     (RETURN (QUOTE UNDEF)))
                 ((NOT (CDR D))
                   (REPLACE (CDR D)(CONS X (CDR D)))
                   (AND (EQ (CAR URD) D)
                        (RPLACA URD (CDR D))) D)
                 ((EQ N 1)
                   (REPLACE (CDR D)(CONS X (CDR D)))
                   (AND (EQ (CAR URD) D)
                        (RPLACA URD (CDR D))) D)
                 (T (APPLY* ADD (CDR D) N X))) D>
```

Cleanup (CLEAN-) Phase. The Cleanup Phase again works through the FUNCT-property of operation names, and makes optimizations and final expansions. In our example, the following rules

apply for ADD:

```
CLEAN02. Replace occurrences matching
         (REPLACE(CDR *EXPR)) with (RPLACD *EXPR)

CLEAN07.   if *OP COUNTS accdg.to selection
              criterion *SCRIT,
              and the FUNCT-property of *OP contains
                  a sub-expression matching
                  (APPLY* *OP ... *SCRIT ... )
         then *SCRIT is replaced with (SUB1 *SCRIT)
```

```
CLEAN02,07:Result.
ADD FUNCT ((LAMBDA (D N X)
             (COND((AND(NOT(CDR D))(NOT((EQ N 1)))
                         (RETURN 'UNDEF))
                   ((NOT(CDR D))(RPLACD D(CONS X (CDR D)))
                   (AND(EQ(CAR URD)D)(RPLACA URD(CDR D)))D)
                   ((EQ N 1)(RPLACD D(CONS X (CDR D)))
                   (AND(EQ(CAR URD)D)(RPLACA URD(CDR D)))D)
                   (T(APPLY* ADD(CDR D)(SUB1 N)X)))D))
```

LISP-Phase. Simulating a construct for operation-clusters in LISP, the LISP-Phase generates the following LISP-code:

```
<OPENLIST
  <NLAMBDA (OP . DLISTE)
    (SETQ ARGUMENTE (CDR DLISTE))
    (SETQ DATSTRUK (CAR DLISTE))
    (SETQ DLISTE (APPEND (LIST (EVAL (CAR DLISTE)))
                         (CDR DLISTE)))
    (PROG ((URD (CAR DLISTE)))
          (COND
            ((ASSOC OP OPENLIST)
             NIL)
            (T (PRINTL "***ERROR:" OP "IS NO"
                        (QUOTE OPENLIST) "OPERATION!")
               (TERPRI)
               (RETURN)))
          (RETURN (EVALA (QUOTE (APPLY (EVAL OP)
                                       DLISTE))
                         OPENLIST>>
  (RPAQQ OPENLIST ((DO NLAMBDA (OBEN)
                                (SETQ OBEN (CONS))
                                (RPLACA OBEN OBEN))
          <READ LAMBDA (D A)
                (COND
                  ((NOT (CDR D))
                    (RETURN (QUOTE UNDEF)))
                  ((EQ A 1)
                    (CAR (CDR D)))
                  (T (APPLY* READ (CDR D)
                             (SUB1 A>
```

```
(REMOVE LAMBDA (D A)
         <COND
           ((NOT (CDR D))
             (RETURN (QUOTE UNDEF)))
           <(EQ A 1)
             (AND (EQ (CAR URD)
                      (CDR D))
                  (RPLACA URD D))
             (RPLACD D (CDR (CDR D>
           (T (APPLY* REMOVE (CDR D)
                      (SUB1 A>
         D)
(ADD LAMBDA (D A X)
       (COND
         ((AND (NOT (CDR D))
               (NOT (EQ A 1)))
           (RETURN (QUOTE UNDEF)))
         ((NOT (CDR D))
           (RPLACD D (CONS X (CDR D)))
           (AND (EQ (CAR URD)
                    D)
                (RPLACA URD (CDR D)))
           D)
         ((EQ A 1)
           (RPLACD D (CONS X (CDR D)))
           (AND (EQ (CAR URD)
                    D)
                (RPLACA URD (CDR D)))
           D)
         (T (APPLY* ADD (CDR D)
                    (SUB1 A)
                    X)))
       D)))

(PUTPROPS OPENLIST STRUCTURE OPENLIST PARAMETER NIL)

THE FOLLOWING RULES HAVE BEEN APPLIED:
      -%>01 -%>02A -%>03 -%>07A -%>08
      =01 =03 =04 =06A =06B =06C =10
      &13B &15B &21 &29 &32
      I02 I08 I08B I33 I10 I15 I06 I23 I13 I36
      B02 B07
```

5. The Automatic Implementation of Abstract Algorithms

The second subsystem of the APE is called ALGCOMP, and constructs LISP-implementations of abstract algorithms given in terms of a syntax as outlined in section 3.2. Just like ADTCOMP, ALGCOMP is an expert system with programming knowledge being codified in terms of production systems. In this section, we give an overview of the structure of the system, and then explain how ALGCOMP works by giving a detailed walk through the implementation of the selection sort algorithm. Note that abstract algorithms are production systems using terms constructed from operation symbols of abstract data types. Implementations generated by ALGCOMP are based on implementations of all abstract data types involved, as produced by ADTCOMP.

5.1. System Structure

An abstract algorithm input is processed by ALGCOMP in three phases:

<u>INPUT-Phase</u>: The abstract algorithm is parsed and translated into an internal form. A user is supported in debugging syntax errors.

<u>IMPLEMENTATION-Phase</u>: This phase constructs a LISP-LAMBDA-form implementing the abstract algorithm.

<u>LISP-Phase</u>: The output of the previous phase is extended by an interface according to the data given in the <header>-section of the abstract algorithm specification. The result is a definition of a LISP-function which can be invoked by supplying data to the user accessible parameter positions.

Only the IMPLEMENTATION-Phase incorporates specialized programming knowledge in terms of production systems. This phase proceeds in six successive steps:

<u>STEP-1</u>: The test-expressions constitute the left-hand-sides of the productions of the abstract algorithm. Each test-expression is decomposed to a disjunction or conjunction of literals, i.e. atomic tests.

<u>STEP-2</u>: Each literal is converted to a LISP-form.

<u>STEP-3</u>: The outcomes of **STEP-1** and **STEP-2** are combined to produce LISP-forms for all left-hand-sides.

The remaining STEP-4 to STEP-6 apply to the right-hand-sides (actions) of the production rules.

<u>STEP-4</u>: General observations about the structure of right-hand-sides are made. Particularly, all **copying** decisions are made. This is necessary because arguments in different parameter positions are independent of each other in the abstract specification. For example, if POP(s) and s (with s denoting a stack) occur

in different parameter positions, applying the operation POP to s has no influence on the stack s at the other parameter position. If, however, s is implemented as a specific stack object, POPping this object may result in encountering the POPped stack object, when coming across the object implemented for s at the second parameter position. One possibility to avoid such unwanted side-effects consists in maintaining different copies of data objects at different positions. To avoid unnecessary copies, the necessity for making copies, resp. working with references without copying is established in **STEP-4**.

STEP-5: Recursive calls to the algorithm are processed. In particular, this involves substitutions and making copies as established in **STEP-4**.

STEP-6: The results of all previous steps are assembled to construct a LAMBDA-form implementing the algorithm.

5.2. A Walk through the Automatic Implementation of the Selection-Sort Algorithm

We consider the following input specification of the Selection-Sort algorithm:

```
abstract algorithm SELECTION-SORT;
parameters          user OPENLIST[INT] OL;
                    internal OPENLIST[INT] OL1 :=
                             (INIT OL1 OPENLIST[INT]),
                             OPENLIST[INT] OL2 :=
                             (INIT OL2 OPENLIST[INT]);
external            MAXLIST, APPEND;
prodsystem SELECTION-SORT(OL,OL1,OL2) =
(1) { read(OL,1) = UNDEF ---> OL2;
(2)    read(OL,1) ≠ MAXLIST(OL)
     --->  SELECTION-SORT(remove(OL,1),
                         add(OL1,1,read(OL,1)),OL2);
(3)    read(OL,1) = MAXLIST(OL)
     --->  SELECTION-SORT(APPEND(OL1,remove(OL,1)),
                         EMPTY-OBJECT,
                           add(OL2,1,read(OL,1))) }
comment SELECTION-SORT obtains the maximum of
        OL, adds this value to OL2 and removes
        it from OL. Then, SELECTION-SORT is
        recursively applied to OL until OL is empty.
        OL2 then contains the resulting sorted list!!
```

Omitting the conversion of the abstract algorithm to an internal representation in the **INPUT-phase**, we describe how the **IMPLEMENTATION-phase** generates a LAMBDA-form implementing SELECTION-SORT.

STEP-1. This step applies to left-hand-sides (lhs) of productions only. There is only one applicable rule which observes

that all left-hand-sides are literal boolean expressions. It is also noted that the lhs of rule (2) contains a ≠-relation.

STEP-2. First, each lhs-expression is decomposed and searched for applications of external algorithms:

```
S2-1.    if lhs matches <*REL,*LEFT,*RIGHT>,
            and *LEFT invokes the agorithm
                being implemented,
            and *RIGHT invokes an external algorithm
         then
            L PREDICATE
                       ((REL *REL)(LEFT *LEFT)(RIGHT *RIGHT)
```

```
S2-1:Result. L2 PREDICATE ((REL ≠)
                           (LEFT read(OL,1))
                           (RIGHT MAXLIST(OL)))
```

A second rule S2-9 determines that ≠ is implemented as NEQ in LISP, yielding

```
S2-9:Result. L2 PREDICATE (FORM.(NEQ read(OL,1)
             MAXLIST(OL)))
```

Notation. Li resp. Ri denote the left- resp. right-hand-side of rule i.

STEP-3. (No significance in this example).

The subsequent steps apply to the rhs of the productions of the algorithm only.

STEP-4.
In its simplest form, a rhs is just an atom. This is detected by rule 4-2:

```
S4-2.    if rhs matches <*ATOM>
            and *ATOM is a formal parameter
         then invokation of the algorithm evaluates to *ATOM.
```

```
S4-2:Result. R1 FORM      (OL2)
             R1 EVALUATES (PARAMETER)
```

Rule S4-1 observes that (2) and (3) contain recursive calls to the algorithm:

```
S4-1.    if rhs matches <*ALGNAME(...)>
       then rhs contains a recursive call to the algorithm.
```

```
S4-1:Result. R2 EVALUATES (RECURSIVELY)
             R3 EVALUATES (RECURSIVELY)
```

ALGCOMP attempts to convert recursive to iterative structures. When modelling a recursive call by an iteration, values corresponding to parameters being passed in a call have to be determined. Very often, data objects representing such values need to be copied to prevent unwanted side-effects. In our example, in (2) and (3) a copy of OL must be supplied as an argument of the operation read, as the remove operation changes OL.

```
S4-51.    if a subexpression of rhs matches
             <*PARAM ... (*PARAM ... >
             and *PARAM is a formal parameter
          then replace the second occurrence of *PARAM
             with a copy.
```

```
S4-51:Result. R2 COPIES (OL)
              R3 COPIES (OL)
```

Finally, rule S4-9 prepares arguments of recursive calls for STEP-5:

```
S4-9:Result. R2 ARG1     (EXPR.(remove(OL,1))
             R2 ARG2     (EXPR.(add (OL1,1,read(OL,1))))
             R3 ARG3     (EXPR.(OL2))
             R2 ARGLIST  (ARG1 ARG2 ARG3)
```

<u>STEP-5</u>. We now consider the arguments of recursive calls. Rule S5-1 applies to the simplest case that such an argument is an atom, and this is true for ARG3. The action of rule S5-1 consists in attacking the form (SETQ *FP *ATOM) to the corresponding argument, where *FP matches the appropriate formal parameter, and *ATOM the argument itself.

```
S5-1:Result. R2 ARG3 (EXPR.(SETQ OL2 OL2))
```

However,in our case S5-1 results in a dummy assignment, and there is a rule S5-2 deleting such assignments:

```
S5-2:Result. R2 ARG3 (FORM.)
```

Rule S5-1 also applies if the specification requires assigning the empty object of the corresponding data structure. Then, rule S5-9 obtains the empty data object of the data structure, i.e. (OPENLIST DO) in our example:

```
S5-1:Result. R3 ARG2 (EXPR.(SETQ OL1 EMPTY-OBJECT))
S5-9:Result. R3 ARG2 (FORM.(SETQ OL1(OPENLIST[INT] DO)))
```

Productions (2) and (3) contain recursive calls to SELECTION-SORT which would generate procedure instances if the algorithm were implemented as a recursive procedure. This is detected by rule S5-31 and results in

```
S5-31:Result. R2 ARG1 (EXPR.(SETQ OL remove(OL,1)))
              R3 ARG3 (EXPR.
                         (SETQ OL2 add(OL2,1,read(X3,1))))
```

However, the recursive calls change OL resp. OL2 so that no assignment is necessary . S5-8 removes the assignment introduced by S5-31:

```
S5-8:Result. R2 ARG1 (FORM.remove(OL,1))
             R3 ARG1 (FORM.add(OL2,1,read(X3,1)))
```

Considering the other arguments, rules S5-31 and S5-8 also apply to ARG2 of R2:

```
S5-31.    if call corresponds to a recursive instantiation
             and rhs has property COPIES with value *P
        then replace *P with X2.
```

```
S5-31:Result. R2 ARG2
             (EXPR.(SETQ OL1 add(OL1,1,read(X2,1))))
```

```
S5-8:Result. R2 ARG2 (FORM.add(OL1,1,read(X2,1)))
```

Calls to external algorithms are rewritten by rule S5-4:

```
S5-4:Result. R3 ARG1
            (EXPR.(SETQ OL (APPEND OL1 remove(OL1,1))))
```

STEP-6. The right-hand-sides are now assembled according to the result of **STEPs-4,5.** For (1), **STEP-4** has observed that R1 is just an atom , so R1 is kept unchanged:

```
S6-2:Result. R1 FINAL (OL2)
```

For R2, and R3, forms for evaluating the arguments have been generated already. In addition, the decissions about copying arguments need to be implemented now, and recursive calls to the algorithm are coded by rule S6-1:

```
S6-1:Result.
R2 FINAL ((COPY OL X2)(remove(OL,1))
                      (add(OL1,1,read(X2,1)))
                      (SELECTION-SORT OL OL1 OL2))
R3 FINAL ((COPY OL X3)(SETQ (APPEND OL1 remove(OL,1)))
                      (SETQ OL1 (OPENLIST[INT] DO))
                      (add(OL2,1,read(X3,1)))
                      (SELECTION-SORT OL OL1 OL2))
```

Finally, left-and-right-hand-sides are compiled to an LAMBDA-form:

LISP-Phase. The LISP-Phase establishes an interface for calling the implementation of the algorithm, following data and requirements given in the <header>-section. In particular, the implementation successively clears LISP-stacks s.t. garbage copies of parameters are deleted. The result is that at run-time the function stack of the LISP-system contains a top-level call to the algorithm implementation, together with just one copy of the COND-form independent of the extent of the file to be sorted and the corresponding number of recursive calls.

Notation: For reasons of easier internal use calls to data type operators are converted into a LISP-executable form, e.g. read(OL,1) is converted to (OL READ 1).

Result of the LISP-Phase:

```
<SELECTION-SORT
 <LAMBDA (OL OL1 OL2)
   (COND
    ((EQ (QUOTE OPENLIST[INT])
         (GETP OL (QUOTE PARAMETER)))
     (COND
      ((GETD (QUOTE APPEND))
       (COND
        ((GETD (QUOTE MAXLIST))
         (COND
          ((GETD (QUOTE COPY))
```

```
          (INIT OL1 OPENLIST[INT])
          (INIT OL2 OPENLIST[INT])
          <RPLACA (QUOTE SELECTION-SORT)
           (DSUBST OL (QUOTE OL)
            (SUBST <QUOTE (RETEVAL
                            (STKPOS (QUOTE SELECTION-SORT) -1)
                            (QUOTE (SELECTION-SORT NIL>
                   (QUOTE (SELECTION-SORT OL OL1 OL2))
                   (GETP (QUOTE SELECTION-SORT)
                         (QUOTE ALGORITHM>
           (APPLY SELECTION-SORT NIL))
       (T (PRINT "***ERROR*** AUXILIARY FCN" (QUOTE COPY)
                 "UNDEFINED")))
      (T (PRINT "***ERROR*** EXTERNAL" (QUOTE MAXLIST)
                "UNDEFINED")))
     (T (PRINT "***ERROR*** EXTERNAL" (QUOTE APPEND)
               "UNDEFINED"))))
    (T (PRINT "***ERROR***" OL "NOT OF TYPE OPENLIST[INT]")
         NIL>>
```

Result of automatic coding:

```
    <LAMBDA NIL
           (COND
             ((EQ (OL READ 1)
                  (QUOTE UNDEF))
               OL2)
             ((NEQ (OL READ 1)
                   (MAXLIST OL))
               (COPY OL X2)
               (OL REMOVE 1)
               (OL1 ADD 1 (X2 READ 1))
               (SELECTION-SORT OL OL1 OL2))
             ((EQ (OL READ 1)
                  (MAXLIST OL))
               (COPY OL X3)
               (SETQ OL (APPEND OL1 (OL REMOVE 1)))
               (SETQ OL1 (OPENLIST[INT] DO))
               (OL2 ADD 1 (X3 READ 1))
               (SELECTION-SORT OL OL1 OL2))
             (T (ERROR 10>
```

THE FOLLOWING RULES HAVE BEEN APPLIED:
(S1-1 S2-1 S2-9 S3-1 S4-1 S4-2 S4-51 S4-52 S4-9 S5-1 S5-31
S5-4 S5-8 S5-9 S6-1 S6-2)

6. Accomplishments

In its present state of development, the programming expertise incorporated in the APE enables the system to automatically implement a large amount of algebraic data type specifications, and some of the standard algorithms in sorting and searching.

Data types. THE ADTCOMP-subsystem automatically implements all algebraic specifications we could get hold of from the literature, and many more in addition.
This includes

- open and bounded lists, queues, and stacks;
- sequential file structures;
- arrays and records of fixed and variable length resp. number of fields;
- tree structures.

In our experiments, ADTCOMP has successfully implemented 76 algebraic specifications of abstract data types. As illustrated in the open list example, ADTCOMP admits **hidden operations** , **conditional axioms** , and **parameterized data types.** However, there are two restrictions:

- ADTCOMP requires that all objects of a data type to be implemented are inductively constructed from elementary objects. For example, ADTCOMP cannot implement the ring abstraction.
- For parameterized data types, ADTCOMP does not check requirements imposed on actual parameter types (see [EH 81], [HR 81]).

Algorithms. So far the ALGCOMP-subsystem has successfully implemented the following sorting and searching algorithms (see [KN 73]):

- straight-selection sort;
- straight-insertion sort;
- bubble sort;
- merge sort;
- quick sort;
- Fibonaccian search;
- binary search;
- sequential search.

ALGCOMP is also successful in implementing some numerical algorithms such as the greatest-common-divisor, least-common-multiple, and factorial algorithm.

In applying this expertise, the APE is capable of generating systems of programs which are implementations of abstract specifications. successful example is a management system for processing files of personel data.

7. Final Remarks and Conclusions.

7.1 Interactive System Development and Theory Formation. The development of the APE has been evolutionary, driven by considering more and more examples, and feeding the programming knowledge necessary to implement the examples into the production bases. To support codifying and incorporating programming knowledge, APE has an extensive interactive user interface. This interface helps a user to find out why APE failed, coding new rules and experimenting with them, and finally changing meta-rules and the production control unit after inserting new rules into the production bases. The userinterface is extremly important for **upgrading** the production bases: whenever a new portion of programming knowledge has been fed to APE, it is usually in order to condense rules into fewer and more general rules. **Rule condensation** actually contributes to the formation of a **theory of programming** , and we expect such a theory to eventually evolve from our investigation.

7.2. Accomplishments. Previous systems start at a much more concrete level of data structures such as sets, collections, etc., and implement them at the somewhat more "concrete" level of lists, arrays, etc. APE generates code from representation-free **axiomatizations** of all structures previous systems could implement, and in addition succeeded for about 70 other axiomatizations. APE generates clusters of routines which may be mutually recursive. Previous systems could not introduce recursions.

7.3. A Successful Paradigm. Considering the amount of work which has been invested in its development so far, the APE is still in its infancy. Demonstrated accomplishments and experiments have shown that with incorporating additional programming knowledge, the APE will become a powerful tool in software development. Given the limited success of the program synthesis paradigms (e.g. [BI 79], [BS 77], [MW 79]) after a decade of work, we claim that the knowledge-based programming paradigm supported by the APE is the only approach which is likely to provide tools for substantially automating phases of the industrial software development cycle. In fact, we think that program synthesis and knowledge-based programming complement each other: Program synthesis (as in [BI 79], [MW 79]) should produce abstract algorithms from non-algorithmic specifications. Then, a system such as ours brings in the expertise to produce efficient programs.

8. References

[ADJ 76] Goguen, J.A., J.W. Thatcher, E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. IBM Research Rept. RC-6487 (1976).

[ADJ 80] Ehrig, H., H.-J.Kreowski, J.W. Thatcher, E.G. Wagner, J.B. Wright. Parameterized Data Types in Algebraic Specification Languages. Proc. 7th ICALP, 1980. Springer LNCS, Vol.85, pp.157-168

[BAR 78a] Barstow, D.R. Automatic construction of algorithms and data structures using a knowledge-base of programming rules.
Rept. STAN-CS-77-641(77), Stanford U.

[BAR 78b] Barstow, D.R. Experience with a refinement paradigm in a knowledge-based automatic programming system. Proc. AISB/GI-Conference on Artificial Intelligence 1978.

[BI 79] Bibel, W. Syntax-directed, semantics-supported program synthesis.
Proc. 4th Workshop on Automatic Deduction (Austin, Texas), 1979.

[BS 77] Biermann, A.W., D.R. Smith. The hierarchical synthesis of LISP-programs.
Proc. IFIP 77 (1977):41-45

[DK 77] Davis, R., I.King. An overview of production systems in Elcock, Michie (Eds.) Machine Representation of Knowledge.
Horwood Wylie, 1977.

[EH 81] Ehrig, H. Algebraic theory of parameterized specifications with requirements.
Proc. 6th CAAP, Genova 1981, Springer-Verlag (in press).

[EKRT 80a] Eigemeier, H., Ch.Knabe, P.Raulefs, K.Tramer. Automatic implementation of algebraic specifications of abstract data types.
Memo SEKI-BN-79 (Nov.79), Univ. Bonn, Inst. f. Informatik III and Proc. AISB-80 Conf. on Artificial Intelligence.

[EKTR 80b] Eigemeier, H., Ch.Knabe, P.Raulefs, K.Tramer. An expert system for automatic coding of abstract data type specifications.
Proc. 10th Annual GI-Conf., Springer Informatik-Fachberichte 33,1980, pp. 431-441.

[GR 76] Green, C.C. The design of the PSI program synthesis

system.
Proc. 2nd Conf. on Software Engineering (San Francisco, CA), 1976.

[HR 80] Hornung, G., P.Raulefs. Terminal algebra semantics and retractions for abstract data types.
Proc. 7th ICALP, 1980. Springer LNCS, Vol.85, pp. 310-323

[HR 81] Hornung, G., P.Raulefs. Initial and terminal algebra semantics of parameterized abstract data type specifications with inequalities.
Proc. 6th CAAP, Genova 1981, Springer-Verlag (in press).

[KA 79] Kant, E. Efficiency considerations in programming synthesis: a knowledge-based approach.
Tech. Rept. STAN-CS-79-755, Comp. Sci. Dept, Stanford Univ.

[KN 73] Knuth, D.E. The Art of Computer Programming.
Vol.3, Addison-Wesley, 1973

[MW 79] Manna, Z., R.Waldinger. Synthesis: Dreams => Programs.
IEEE-TSE: 5 (1979)294-327.