PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

AUTOMATIC IMPLEMENTATION OF ABSTRACT DATA TYPES SPECIFIED BY THE LOGIC PROGRAMMING LANGUAGE

Norbert Heck and Jürgen Avenhaus
University of Kaiserslautern
Computer Science Department
D-6750 Kaiserslautern
FR Germany

ABSTRACT

We present a specification method for abstract data types which is based on the logic programming language and propose a framework for deriving correct implementations of abstract data types from their specifications. Provided that several conditions are met, it is possible to decompose the generation of implementations into three phases. The first and second phase which are sufficient for deriving a correct implementation lend themselves for automation. The third phase which serves for optimizations and syntheses can be done interactively using a well suited set of transformation rules.

1 INTRODUCTION

It is known that (Horn clause) logic can be interpreted as a high level programming language (Kowalski 1974). Such a high level programming language is used to specify problems in an abstract, i.e. implementation-independent but formal and easy to understand way. This specification has finally to be implemented on a real computing device.

Normally the distance between the abstract specification and the computer level is very great, so one introduces more levels and performs the implementation in steps. A typical problem is then as follows: given an "abstract" data type, i.e. a specification of the input/output relation of some operations and given a "concrete" data type on a lower level, one has to implement the abstract data type in terms of the concrete one.

In this paper we show that logic programs can be used to specify abstract data types and that the implementation again can be done using the logic programming language. This outlines a unifying approach based on logic which leads from a specification of a problem to an implementation on a computer.

The data types defined by our specification method constitute an algebra: the elements which represent abstract data objects are terms, the operations act on these terms. Both, the set of terms representing data objects and the operations are defined by logic programs. So the semantic of an abstract data type is defined algorithmically. This contrasts with the algebraic specification method where the semantic is defined statically by equations (Goguen et al. 1978). Our method equals more the concept of (Loeckx 1981).

Next we study the problem of how to implement one data type by another, both data types given in terms of logic. Usually, after having specified the implementation type one has to prove its correctness against the specification. We will show that under reasonable conditions the construction of a correct implementation from a specification is mechanizable. This implementation however may not be very efficient. A collection of rules for program transformations can be given to derive a better implementation in an interactive manner. For program transformation systems see also (Burstall and Darlington 1977) and (Bauer and Wössner 1981).

Because of lack of space this paper does not contain all formal proofs, instead we give a simplified overview. For more details see (Heck 1984).

2 LOGIC PROGRAMS

In the following we assume that the reader is familiar with Horn clause logic as programming language (Kowalski 1974). So we repeat only the basic facts.

A <u>logic program</u> is a finite set of

Horn clauses $A \vee \neg B_1 \vee \ldots \vee \neg B_m$ written as

$$A \leftarrow B_1, \ldots, B_m \qquad m \geq o$$

where A and the B_i are atomic formulas in the sense of first order logic. A logic program R works on an input called goal statement

$$G \equiv \leftarrow A_1, \ldots, A_n \qquad n \geq 1$$

as follows. Suppose there is a clause $A \leftarrow B_1, \ldots, B_m$ in R and a j, $1 \leq j \leq n$, such that A_j and A are unifiable by a most general unifier σ. Then G derives

$$G' \equiv \leftarrow \sigma A_1, \ldots, \sigma A_{j-1}, \sigma B_1, \ldots, \sigma B_m, \sigma A_{j+1}, \ldots, \sigma A_n$$

A computation with input G is a sequence of pairs

$$(G_o, \sigma_o), (G_1, \sigma_1), \ldots, (G_k, \sigma_k)$$

such that $G_o \equiv G$, $\sigma_o = \emptyset$ is the empty substitution, G_i derives G_{i+1} with mgu σ and $\sigma_{i+1} = \sigma \cdot \sigma_i$. The computation is successful if G_k is the empty clause □.

Logic programs are used to compute relations (or functions). To explain this let TERM be the set of terms (over some set of variables) that can be built with the function symbols occurring in the program R and let H be the Herbrand universe of R, i.e. the set of variable free terms. If P is a n-place predicate symbol of R we define

$$\hat{D}_R(P) = \{(\sigma(t_1), \ldots, \sigma(t_n)) \mid t_1, \ldots, t_n \in \text{TERM}, \text{ there is a successful computation } (\leftarrow P(t_1, \ldots, t_n), \emptyset), \ldots, (\square, \sigma)\}$$

With the completeness result of (Clark 1979) it can be shown that

$(t_1, \ldots, t_n) \in \hat{D}_R(P)$ iff R implies $P(t_1, \ldots, t_n)$ in the sense of first order logic.

We want to associate a relation $D_R(P)$ over the set of ground terms H to P that reflects what is computed by R. To do so, we fix a set $I_R(P) \subseteq \{1, \ldots, n\}$ of input positions, say $I_R(P) = \{1, \ldots, n-1\}$ for simplicity, and define

$$D_R(P) = \{(t_1, \ldots, t_{n-1}, \sigma(x)) \mid t_1, \ldots, t_{n-1} \in H, \sigma(x) \in H, \text{ there is a successful computation } (\leftarrow P(t_1, \ldots, t_{n-1}, x), \emptyset), \ldots, (\square, \sigma)\}$$

We say R defines P as a total (partial) function if for all $t_1, \ldots, t_{n-1} \in H$ there is exactly (at most) one $t_n \in H$ such that $(t_1, \ldots, t_{n-1}, t_n) \in D_R(P)$. In (Heck 1984) one can find sufficient conditions on R such that

(i) $D_R(P) = \hat{D}_R(P) \cap H^n$

(ii) R defines P as a partial function

(iii) no goal clause $\leftarrow P(t_1, \ldots, t_{n-1}, x)$ admits an infinite computation.

In the rest of the paper all programs are so that (i) holds. As indicated above we will use the logic programming language in an uniform approach for specifying and reasoning about abstract data types.

3 THE SPECIFICATION METHOD

Abstract data types have turned out to be a useful and powerful concept for the design and description of data types while creating a software system. A well-known method for specifying abstract data types is the algebraic style of specification (Goguen et al. 1978).

In this paper we present an algorithmic specification method based on the logic programming language. Whereas algebraic specifications merely describe the desired properties of operations by equations and so define a class of algebras, in the specification method proposed here carriers and operations of an algebra are defined explicitly as sets and functions by Horn clauses.

3.1 Logic specifications

A logic specification of a data type s consists of

a) a signature, that is a list of operation symbols with their types of input and output

b) a term specification which defines the carrier set of s

c) a set of clauses which defines the operations of s declared in a). These clauses define the operations as functions (see Section 2).

We explain the method by an example.

Example Specification of type *queue*

type *queue*

relations
NEWQ(*queue*)
ADDQ(*queue*, *nat*, *queue*)
DELETEQ(*queue*, *queue*)
FRONTQ(*queue*, *nat*)
APPENDQ(*queue*, *queue*, *queue*)

```
terms
  IS-TERM_queue(newq) <-
  IS-TERM_queue(addq(x,i))
    <- IS-TERM_queue(x),IS-TERM_nat(i)
clauses
  NEWQ(newq) <-
  ADDQ(x,i,addq(x,i)) <-

  DELETEQ(newq,newq) <-
  DELETEQ(addq(newq,i),newq) <-
  DELETEQ(addq(addq(x,i),j),addq(x',j))
    <- DELETEQ(addq(x,i),x')

  FRONTQ(addq(newq,i),i) <-
  FRONTQ(addq(addq(x,i),j),i')
    <- FRONTQ(addq(x,i),i')

  APPENDQ(x,newq,x) <-
  APPENDQ(x,addq(x',i),addq(x",i))
    <- APPENDQ(x,x',x")
```

The carrier of the data type *queue* consists of the terms *newq* and *addq*(...(*addq*(*newq*, i_1),..., i_n), where i_j, j = 1,...,n, are elements of the type *nat* which is assumed to have been specified previously and is at a "hierarchically lower level". The function symbols *newq* and *addq* are called <u>term constructors</u> of type *queue*. NEWQ, ADDQ, FRONTQ, DELETEQ and APPENDQ are the operations of *queue*. They are indeed defined as functions, e.g. for any term t_1 of type *queue* and term t_2 of type *nat* there is at most one t_3 of type *queue* such that $(t_1,t_2,t_3) \in D(\text{ADDQ})$.

We call a set of logic specifications <u>hierarchical</u> if it is possible to order its elements to

$$M_1, M_2, \ldots, M_n$$

such that

- M_1 is the specification of type *boolean*
- the specification M_i depends only on types specified by $M_1,\ldots,M_{i-1}$ for all i, $1 \leq i \leq n$ that is, operations of other types which are used in M_i are defined in a specification M_j, $j<i$.

3.2 The data type defined

We now want to associate with a hierarchical set of logic specifications a heteregeneous algebra consisting of carrier sets (one for each type) and a collection of operations. As indicated in the *queue*-example, the carrier elements will be determined by the term specifications and the operations will be derived from the Horn clauses of the specifications.

<u>Definition</u>

Let M be a hierarchical set of logic specifications for types $S = \{s_1,\ldots,s_n\}$. M defines an algebra $\mathfrak{A}$, which consists of

(i) the set $\underline{s} = D(\text{IS-TERM}_s)$ defined by the IS-TERM clauses for each $s \in S$

(ii) the function P: $\underline{r}_1 \times \ldots \times \underline{r}_n \rightarrow \underline{r}_{n+1}$ for all operation symbols $P(r_1,\ldots,r_n,r_{n+1})$ in M, where P is given by
$P(t_1,\ldots,t_n) = t_{n+1}$, if
$(t_1,\ldots,t_n,t_{n+1}) \in D(P)$,
$t_i \in \underline{r}_i$ for all i, $1 \leq i \leq n+1$

Our specification method can be generalized:

- Sometimes we have to deal with data types s where not all terms of $\underline{s}$ represent data objects, i.e. we have to restrict the term language $\underline{s}$ (e.g. bounded types).
- Likewise, it is desirable to identify terms which are syntactically different but represent the same data element. The carrier set contains then equivalence classes as elements.

But for simplicity, we shall consider only logic specifications as in the definition above.

As a concluding remark of this section we want to emphasize that our interest in this paper is in tools for the specification and implementation of software. Introducing data types may also be viewed as an aid to write safer programs in a concrete programming language. This is common in PASCAL and other languages, where data types are used (among others) to prevent mixing objects of different types, e.g. a number and a list consisting only of that number. In our approach we declare for each predicate symbol the types of its arguments. This implies an implicit typing of the variables used in a clause. In (Heck 1984) it is shown that one can easily test whether a logic program is syntactically correct in the sense of correct use of types. A referee pointed out that types in logic have been dicussed earlier. For a different approach see (Mishra 1984).

4 IMPLEMENTATIONS OF ABSTRACT DATA TYPES

A hierarchical set of logic data type specifications is a formal description of an algebra which can be used as a standard in program development. Proposed implementations can be proved correct relative to its specification. We provide a framework in which an implementation is again de-

fined by a logic specification. Such an implementation can be considered as a new specification which needs further treatment towards a final realization.

The following situation is characteristic when implementing a type p: we want to implement p with the help of data types q' which are already available and allow for an efficient implementation of the operations of p. That is, we have to define a new type q, whose elements and operations, implementing the elements and operations of p, are expressed in terms of elements and operations of q'. To be uniform, q is described by a logic specification and its correctness is given by the corresponding algebra.

<u>Definition</u>

Let M be a hierarchical set of specifications for types $S = \{s_1,\ldots,s_n\}$, let M_p be the specification of a type p, $p \in S$, such that $M \cup \{M_p\}$ is hierarchical and defines the algebra $\mathcal{A}_p$, and let M_q be the specification of a type q, $q \in S$, such that $M \cup \{M_q\}$ is hierarchical and defines the algebra $\mathcal{A}_q$. q is said to be an <u>implementation</u> of p, if $\mathcal{A}_p$ and $\mathcal{A}_q$ are isomorphic.

In spite of its symmetric character this definition corresponds to the intuitive notion of an implementation as motivated above. An implementation of p is constructed with the help of data types specified in M.

<u>Example</u> Implementation of type *queue* in terms of type *clist* for circular list (Guttag et al. 1978a))

(i) The specification of the type *queue* is already given.

(ii) The specification of type *clist* is given as follows

```
type clist
relations
  CREATE(clist)
  INS(clist,nat,clist)
  DELETE(clist,clist)
  VALUE(clist,nat)
  RIGHT clist,clist)
  JOIN(clist,clist,clist)
terms
  IS-TERM_clist(create) ←
  IS-TERM_clist(ins(c,i))
    ← IS-TERM_clist(c),IS-TERM_nat(i)
clauses
  CREATE(create) ←
  INS(c,i,ins(c,i)) ←
  DELETE(create,create) ←
  DELETE(ins(c,i),c) ←
  VALUE(ins(c,i),i) ←
  RIGHT(create,create) ←
  RIGHT(ins(create,i),ins(create),i))←
  RIGHT(ins(ins(c,i),i_1),ins(c',i))
    ← RIGHT(ins(c,i_1),c')
  JOIN(c,create,c) ←
  JOIN(c,ins(c',i),ins(c",i))
    ← JOIN(c,c',c")
```

(iii) Let M be the set of specifications for {*boolean, nat, clist*} and let p be *queue*. We now present the specification for an implementation type q = *imqueue*. It refers to M by using the term language <u>clist</u> as carrier set for *imqueue* and by describing the *imqueue*-operations in terms of *clist*-operations.

```
type imqueue
relations
  IM.NEWQ(imqueue)
  IM.ADDQ(imqueue,nat,imqueue)
  IM.DELETEQ(imqueue,imqueue)
  IM.FRONTQ(imqueue,nat)
  IM.APPENDQ(imqueue,imqueue,imqueue)
terms
  IS-TERM_imqueue(t)
    ← IS-TERM_clist(t)
clauses
  IM.NEWQ(c) ← CREATE(c)
  IM.ADDQ(c,i,z)
    ← RIGHT(ins(c,i),z)
  IM.DELETEQ(c_1,c_2) ← DELETE(c_1,c_2)
  IM.FRONTQ(c,i) ← VALUE(c,i)
  IM.APPENDQ(c_1,c_2,z) ← JOIN(c_2,c_1,z)
```

We shall now look for conditions which imply that a type q is an implementation of type p. Clearly, the following theorem holds.

<u>Theorem</u>

Let M, M_p, M_q, $\mathcal{A}_p$ and $\mathcal{A}_q$ be as in the definition above.

Let RP be a relation over <u>q</u>×<u>p</u> definded by a Horn clause program where RP is called a <u>representation function</u>.

q is an implementation of p, if the following two <u>correctness conditions</u> are satisfied:

(i) RP: <u>q</u> → <u>p</u> is defined as a bijective function

(ii) to each function

$P: \underline{r}_1 \times \ldots \times \underline{r}_n \rightarrow \underline{r}_{n+1}$, $n \geq 0$

of p corresponds the function

$IM.P: \underline{r}'_1 \times \ldots \times \underline{r}'_n \rightarrow \underline{r}'_{n+1}$

of q with

$$r'_i = \begin{cases} q & r_i = p \\ r_i & \text{otherwise} \end{cases}, \quad i=1,\ldots,n+1,$$

such that for all $t'_i \in \underline{r}'_i$, $1 \leq i \leq n$:

$$\begin{cases} RP(IM.P(t'_1,\ldots,t'_n)) = P(t_1,\ldots,t_n) & r_{n+1} = p \\ IM.P(t'_1,\ldots,t'_n) = P(t_1,\ldots,t_n) & \text{otherwise} \end{cases}$$

where

$$t_i = \begin{cases} RP(t'_i) & r'_i = q \\ t'_i & \text{otherwise} \end{cases}, \quad i=1,\ldots,n$$

Example

With the help of RP defined by

```
RP(create,newq) ←
RP(ins(c1,j),addq(x,i))
  ← RIGHT(ins(c2,i),ins(c1,j)),RP(c2,x)
```

we can show that *imqueue* is an implementation of *queue* with RP being the isomorphism.

What we have done so far is the following. We have two data types p, q and a function RP: $\underline{q} \rightarrow p$, where p, q and RP are defined by logic programs. We have to prove that q is indeed an implementation of p by means of RP, i.e. the correctness conditions of the last theorem hold. To do so one has to develop tools which allow to prove statements on logic programs. Program transformations, discussed in the next section, are such tools.

Normally it is hard to prove for two given data types p,q that one is an implementation of the other. In Section 6 we try to find automatically an implementation q for p. This will lead in most cases to a very poor implementation, but subsequent program transformations may lead to better ones.

5 TRANSFORMATIONS

As indicated above, we need a concept of transformations for logic programs. Transformation rules which are guaranteed to preserve the semantics are tools for verification methods and development techniques.

Definition

Let PROG be the class of all logic programs.

a) A relation $T \subseteq PROG \times PROG$ is called a transformation rule.
b) Let T be a transformation rule and let P be a predicate symbol. T is called P-valid, if $D_{R_1}(P) = D_{R_2}(P)$ for all $(R_1,R_2) \in T$.
c) T is called valid, if T is P-valid for all predicate symbols P.

Note that P-valid transformation rules preserve the denotation of P.

Definition

Let $T,T_1,T_2,\ldots,T_n$ be valid transformation rules.

a) $R_1 \in PROG$ is T-transformable into $R_2 \in PROG$, if $(R_1,R_2) \in T$ or $(R_2,R_1) \in T$.
b) $R_1 \in PROG$ is $T_1 \cdot T_2 \cdot \ldots \cdot T_n$-transformable, $n \geq 2$, into $R_{n+1} \in PROG$, if there exists $R_2,\ldots,R_n \in PROG$ such that R_i is T_i-transformable into R_{i+1} for $i = 1,\ldots,n$. We write

R_1
↕ T
R_2

R_1
↕ * $T_1 \cdot T_2 \cdot \ldots \cdot T_n$
R_{n+1}

As an example we now consider a fundamental transformation rule which we shall use very often in the next chapters.

Lemma "Symbolic execution"

The following rule T_1 is valid

$T_1 = \{(R_1,R_2) \mid R_1$ contain clauses

$A \leftarrow B_1,\ldots,B_m$ and $B \leftarrow C_1,\ldots,C_l$

(with no variables in common) such that a substitution σ exists with $\sigma = mgu(B,B_i)$ for some i, $1 \leq i \leq m$;

$R_2 = R_1 \cup \{\sigma A \leftarrow \sigma B_1,\ldots,\sigma B_{i-1}, \sigma C_1,\ldots,\sigma C_l,\sigma B_{i+1},\ldots,\sigma B_m\}\}$

Note that T_1 executes procedure calls before runtime, it resembles the 'unfolding rule' of (Burstall and Darlington 1977).

Besides such an elementary transformation rule T_1 we need special rules which are well suited for our application domain. Rules for elimina-

ting recursion are such rules (see next section). As an example we present a rule T_2 for recursion removal. Opposite to the rule T_1 we refer in this rule to types and term constructors.

Lemma "Recursion removal"
Let p_3 be a type with a corresponding IS-TERM$_{p_3}$-clause

$$\text{IS-TERM}_{p_3}(g(t_1,t_2) \leftarrow \text{IS-TERM}_{p_1}(t_1), \text{IS-TERM}_{p_2}(t_2)$$

and let p_1 be a type with the term specification

$$\text{IS-TERM}_{p_1}(a) \leftarrow$$

$$\text{IS-TERM}_{p_1}(f(t_1,\ldots,t_n)) \leftarrow \text{IS-TERM}_{q_1}(t_1),\ldots,\text{IS-TERM}_{q_n}(t_n)$$

where $q_j = p_1$ for one j, $1 \le j \le n$.

Then the following transformation rule T_2 is G-valid:

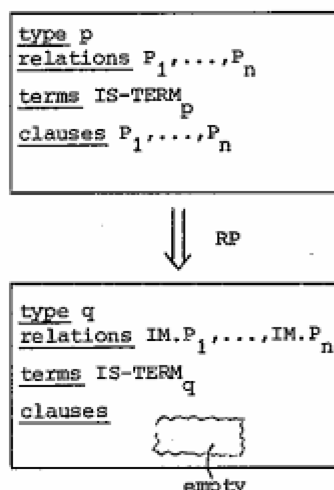
$$T_2 = \{(R_1,R_2) \mid R_1 = R \cup \{G(g(a,x),x) \leftarrow, G(g(f(x_1,\ldots,x_n),x),x') \leftarrow B_1,\ldots,B_m,G(g(z,x),x')\};$$

the two clauses denote G as a function G: $\underline{p}_3 \rightarrow \underline{p}_2$;
the literals $B_1,\ldots,B_m$ do not contain the predicate symbol G;
for all substitutions σ: $V \rightarrow H$ (with $V = \{x_1,\ldots,x_n,x\}$, H the Herbrand universe of R_1^n)
there is a successful computation $(\leftarrow\sigma B_1,\ldots,\sigma B_m,\emptyset),\ldots,(\square,\theta)$, such the "size" of the term $\theta\sigma(z)$ is less than the size of $\sigma(f(x_1,\ldots,x_n))$;
$R_2 = R \cup \{G(g(x,y),y)\leftarrow\}$ }

The proof is by induction on the size of the first g-argument. Note that in the recursive version of G the first g-argument is successively reduced. Since the result is known in advance, long computations can be avoided. This is done in R_2.

6 AUTOMATIC IMPLEMENTATIONS

In this section we shall study the problem of deriving correct implementations of abstract data types from specifications. Rather than to write an implementation by hand, being guided by the specification, and then to prove it satisfies the specifications, we want to generate correct implementations by transformations thus obviating the need for the verification step. The following diagram illustrates our concept:



The data type p which has to be implemented is completely specified while only the signature and the term specification of the implementation type is given. Additionally, a representation function RP is available which reflects the connection between the abstract objects of p and the "concrete objects" described by the IS-TERM$_q$ clauses. With the help of RP we shall try to derive clauses for the implementing operations IM.P_i which define IM.P_i as a correct implementation of P_i, $i = 1,\ldots,n$.

The idea to transform a program to another program by data structure mappings is already developed in (Burstall and Darlington 1977) and (Hansson and Tärnlund 1980). While these authors only treat examples we are looking for general techniques.

We shall define a certain "class of data types and representations" which lends itself to a constructive process of implementation. The data type which we want to implement and the used representation function have to meet some requirements. For the sake of simplicity we assume that the term structure of p is built by one constant a and one function symbol f only. If there are more function symbols the following construction has to be applied to all of them.

Definition
Let M_p be the specification of a data type p.
p is suited for an inductive implementation, if

- the term specification of M_p consists of the two clauses
  $\text{IS-TERM}_p(a) \leftarrow$

$IS\text{-}TERM_p(f(t_1,\ldots,t_n))$
$\leftarrow IS\text{-}TERM_{p_1}(t_1),\ldots,IS\text{-}TERM_{p_n}(t_n)$

that is, a and f are the term constructors of p.

- the relations-part of M_p contains besides other symbols the two operation symbols A(p) and $F(p_1,\ldots,p_n,p)$
- the clauses-part of M_p contains besides other clauses the two clauses

  $A(a) \leftarrow$
  $F(x_1,\ldots,x_n,f(x_1,\ldots,x_n)) \leftarrow$

  A and F are called constructor functions, which have to be distinguished from the term constructors a and f. Note that all terms of p can be constructed by application of A and F.

For example, unbounded linear and binary types can be defined in such a way that they are suited for an inductive implementation (e.g. lists, stacks, queues, trees).

Definition
Let M_p be the specification of type p.
M'_q is called a partial specification of a type q relative to p, if the following holds:

(i) to each operation symbol $P(r_1,\ldots,r_n)$ of M_p corresponds an operation symbol $IM.P(r'_1,\ldots,r'_n)$ of M'_q, where

$$r'_i = \begin{cases} q & r_i = p \\ r_i & \text{otherwise} \end{cases}, \quad i=1,\ldots,n$$

M'_q contains no other operation symbols.

(ii) M'_q contains a term specification defining q.

(iii) The clauses-part of M'_q is empty.

Definition
Let M_p be the data type specification of p suited for an inductive implementation with term constructors a and f. Let M'_q be a partial specification of q relative to p.
A relation RP over q×p is called an inductive representation function, if it is defined by the clauses

$RP(c,a) \leftarrow$
$RP(d,f(x_1,\ldots,x_n))$
$\leftarrow G_1,\ldots,G_m,\{RP(d_j,x_j)\}_{p_j = p}$

which define RP: q → p as a bijective function.

Remarks

1) $\{RP(d_j,x_j)\}_{p_j = p}$ is used as an abbreviation for $RP(d_{j_1},x_{j_1}),\ldots,RP(d_{j_k},x_{j_k})$ where the j_i-th component of $f(x_1,\ldots,x_n)$ is of type p for each $i = 1,\ldots,k$. The literals G_i, $i = 1,\ldots,m$, represent calls of operations, which are at the disposal of p. The elements c,d,d_j are terms containing only variables and term constructors of type q.
2) RP is defined by induction on the structure of p.
3) RP satisfies the correctness condition (i) of an implementation.

Example
Consider the example of Section 4 with p = *queue*, q = *imqueue*. Let M'_q be the *imqueue*-specification without the clauses-part. RP: imqueue → queue as defined in Section 4 is an inductive representation function:

$RP(create,newq) \leftarrow$
$RP(ins(c_1,j),addq(x,i))$
$\leftarrow RIGHT(ins(c_2,i),ins(c_1,j),RP(c_2,x))$

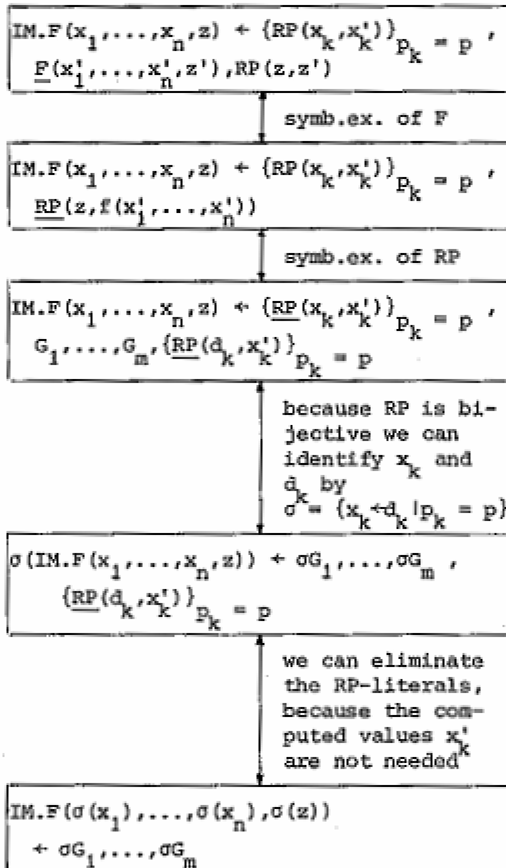
Starting from the abstract characterization of inductive implementations we are ready to divide the process of implementation into three phases:

1) implementation of the constructor functions of p
2) implementation of the other operations of p
3) optimizations and syntheses.

After stating the general theorems we shall demonstrate the phases by the *queue*-*imqueue* example.

Theorem
Let p be a type suited for an inductive implementation with constructor functions A and F, let M'_q be a partial specification for q relative to p and let RP: q→p be an inductive representation function.
Then clauses for IM.A and IM.F which define IM.A and IM.F as a correct implementation of A and F can be derived.

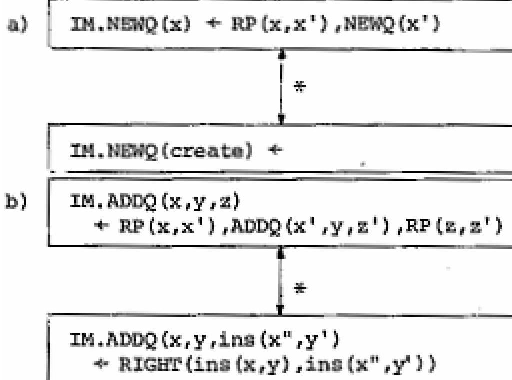
Proof. To get a clause for IM.F we carry out the following transformation steps:



Note that the first clause represents the condition (ii), which a correct implementation IM.F of F has to satisfy, and IM.F is characterized as a function. The transformation preserves the semantics of IM.F.

Example

In our current example we have



In the next phase the remaining functions are implemented automatically. First, it is necessary to eliminate the occurrences of p-terms in the clauses defining these operations.

Definition

Let p be a type with term constructors a,f.

A clause C is called p-free, if no term constructors a,f occur in C.

Lemma

Let p be a type suited for an inductive implementation with constructor functions A and F. Then the clauses for the operations P of p, $P \neq A$, $P \neq F$, can be formulated in a p-free way which does not change the semantics of P.

Sketch of proof

Replace each occurrence of a or f in a clause C by a new variable z and add a call of A or F to the body of C,

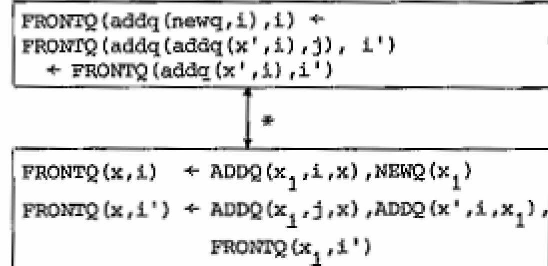
e.g. replace $B \leftarrow B_1,\ldots,B_{i-1}, P(f(x_1,\ldots,x_n)),B_{i+1},\ldots,B_m$

by $B \leftarrow B_1,\ldots,B_{i-1}, P(z),F(x_1,\ldots,x_n,z), B_{i+1},\ldots,B_m$

In order to remove composed terms we must iterate the above step.

Example

Consider the FRONTQ-clauses in our example



We now present the "substitution rule" which states the following fact:

A correct implementation IM.P for P can be derived by replacing calls of constructor functions A and F by calls of IM.A and IM.F.

Theorem "Substitution rule"

Let p,q,RP be as above and let IM.A, IM.F be the implementations of A and F generated by the transformations of the first phase. Let P denote a function of p different from A and F which is defined by p-free clauses.

Then a correct implementation IM.P of P is given by the following clauses:

if $B_o \leftarrow B_1,\ldots,B_m$ is a clause for P,

then choose $B'_o \leftarrow B'_1,\ldots,B'_m$ as a clause for IM.P, where

$$B_i' = \begin{cases} IM.A(z) & , B_i = A(z) \\ IM.F(x_1,\ldots,x_n,z) & , B_i = F(x_1,\ldots,x_n,z) \\ IM.P'(y_1,\ldots,y_r) & , B_i = P'(y_1,\ldots,y_r), \\ & \text{P' is a function of p} \\ B_i & , B_i \text{ calls a function from another type} \end{cases}$$

We omit the proof which relies on the bijection of RP.

Example

Applying the substitution rule to the *queue*-free FRONTQ-clauses we get

```
IM.FRONTQ(x,i)
  ← IM.ADDQ(x1,i,x),IM.NEWQ(x1)
IM.FRONTQ(x,i')
  ← IM.ADDQ(x1,j,x),IM.ADDQ(x',i,x1),
    IM.FRONTQ(x1,i')
```

The clauses for IM.DELETEQ and IM.APPENDQ can be derived similarly.

As the result of the first and second phase we have generated automatically correct characterizations of the implementing operations. The derived functions act upon the q-terms, all calls of RP have been removed.

But we are not content with these implementations simulating only the behaviour of the p-functions on the q-level ("modulo RP").

We want to transform the clauses for the operations IM.P to make optimal use of the operations of types q' the type q depends on. For instance, in our current example we want to perform the following transformation:

```
IM.FRONTQ(x,i) ← IM.ADDQ(x1,i,x),
                 IM.NEWQ(x1)
IM.FRONTQ(x,i')← IM.ADDQ(x1,j,x),
                 IM.ADDQ(z,i,x1),
                 IM.FRONTQ(x1,i')
          |
          | *
          |
IM.FRONTQ(x,y) ← VALUE(x,y)
```

The above-mentioned transformation cannot be carried out automatically, since it is not known in advance which transformation rules have to be applied. Therefore, the third phase can be considered as the "creative" phase of the process of an inductive implementation, in which the user has to choose appropriate rules.

Our investigations have shown that a characteristic collection of transformation rules can be found for this phase. The rules can be classified as

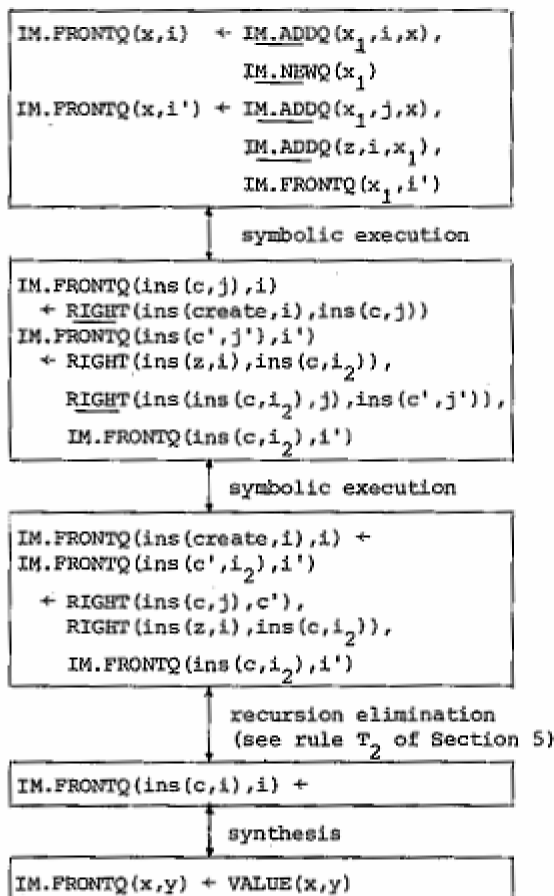
- optimization rules
- synthesis rules.

The rule of symbolic execution is a typical optimization rule. Calls which will take place at run time are executed in advance. Also a rule which eliminates "useless" calls in a procedure body can be regarded as an optimization rule. A useless call computes a value which does not contribute to the semantics. Finally, rules for eliminating recursion are highly desirable.

We do not explicitly present synthesis rules here but we can characterize them as follows: the clauses which we derive by applying optimization rules often have similar syntacical structure like clauses already available. Then it is easy to express the implementing functions in terms of these functions (see the next example).

Example

We demonstrate the third phase of an inductive implementation by manipulating the IM.FRONTQ-clauses:



With similar transformations it is possible to derive the clauses

IM.DELETEQ(x,y) ← DELETE(x,y)
IM.APPENDQ(x,y,z) ← JOIN(y,x,z)

So we have found the implementation of type *queue* in terms of type *clist* as in Section 4 by program transformation. By construction this implementation is correct.

It is worth mentioning that we have examined other examples which fit into the framework of an inductive implementation:

- implementation of a stack in terms of lists
- implementation of a symboltable in terms of a stack of vectors (Guttag et al. 1978b))
- implementation of a stack of booleans in terms of natural numbers
- implementation of a labelled tree in terms of binary trees (Burstall and Darlington 1977).

REFERENCES

Bauer, F.L. and Wössner, H. Algorithmische Sprache und Programmentwicklung. Springer-Verlag, Berlin, 1981.

Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs. J. ACM 24,1, pp. 44-67, 1977.

Clark, K.L. Predicate Logic as a Computational Formalism. Res. Report, Imperial College, London, 1979.

Goguen, J.A., Thatcher, J.W. Wagner, E.G. and Wright, J.B. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In: Current Trends in Programming Methodology 4, (R.T. Yeh, Ed.), Prentice Hall, pp. 80-149, 1978.

Guttag, J.V., Horowitz, E. and Musser, D.R. The Design of Data Type Specifications. In: Current Trends in Programming Methodology 4, (R.T. Yeh, Ed.), Prentice Hall, pp. 60-79, 1978a).

Guttag, J.V., Horowitz, E. and Musser, D.R. Abstract Data Types and Software Validation. Comm. of the ACM 21,12, pp. 1048-1064, 1978b).

Hansson, A. and Tärnlund, S. Program Transformation by a function that maps simple lists onto d-lists. Logic Programming Workshop 1980, pp. 225-229, 1980.

Heck, N. Abstrakte Datentypen mit automatischen Implementierungen. Ph.D. Thesis, University of Kaiserslautern, 1984.

Kowalski, R. Predicate Logic as Programming Language. Proc. IFIP 74, North Holland, Amsterdam, pp. 569-574, 1974.

Loeckx, J. Algorithmic specifications of abstract data types. Proc. ICALP 81, LNCS 115, pp. 129-147, 1981.

Mishra, P. Towards a Theory of Types in PROLOG. IEEE Symposium on Logic Programming, pp. 289-298, 1984.