ALGORITHMIC SPECIFICATIONS
OF ABSTRACT DATA TYPES

Jacques Loeckx

Fachbereich 10
Universität des Saarlandes
D-6600 Saarbrücken

Summary. A new method for the specification of abstract data types is presented. Being algorithmic it avoids several difficulties of the algebraic specification method.

1. Introduction

The algebraic specification of abstract data types as described in, for instance, [Zi 74, ADJ 78, GH 78 a, BDP 79] raises several problems of theoretical and practical nature. A first problem is the treatment of partial or ERROR-functions [Go 78, Gu 80, WPP 80]. Furthermore certain (partial computable) functions cannot be introduced [Mj 79, BM 80] and there are several problems attached to the enrichment of specifications [ADJ 78, EKP 78, Kl 80]. Next, the verification of an algebraic specification requires a proof of its consistency and sufficient-completeness [GH 78 a]. Finally, writing specifications for a given data type is not necessarily a trivial exercise - as is illustrated by the data type *Set-of-Integers* of [GH 78 a, Gu 80]; one of the axioms for the function *Delete* removing an element from a set is:

Delete(Insert(s, i), j) =
    __if__ $i = j$ __then__ Delete(s, j)
            __else__ Insert(Delete(s, j), i)   ;

the __then__-clause of this equation is Delete(s, j) rather than s because an element of the data type *Set-of-Integers* may contain duplicates; intuitively it is not directly clear why these duplicates may occur nor where they occur. Other examples of "difficult" specifications are in [Ka 79] and [Mo 80].

In the present paper a formal specification method for abstract data types is proposed which avoids these different problems. The basic idea consists in defining an abstract data type by a formal language - called *term language* - , an equivalence relation over the term language and some external functions taking arguments and/or values in the term language; the functions are defined constructively using λ-abstraction and minimal fixpoint abstraction. The carrier set of the data type defined by such a specification is, roughly speaking, the set of the equivalence classes induced by the equivalence relation; the operations on the data type are derived in the classical way from the external functions.

Section 2 is concerned with the definition of the term language. Section 3 introduces some notations. The definition of algorithmic specifications is in Section 4. The verification of specifications is treated in Section 5. Section 6 is devoted to comments including a comparison with related work.

2. Term languages

2.1 Definitions

A *basis* is a pair ($\underline{T}$, $\underline{F}$) where $\underline{T}$ is a set of *types*, such as *Integer* or *Stack*; and $\underline{F}$ a set of *constructors*, such as *plus* or *push*. With each $f \in \underline{F}$ are associated an integer $n \geq 0$, an element $(\tau_1, \tau_2, \ldots, \tau_n)$ of $\underline{T}^n$ and an element τ of $\underline{T}$; one writes:

$$f : \tau_1 \times \tau_2 \times \ldots \times \tau_n \rightarrow \tau$$

for instance:

push: *Stack* x *Integer* → *Stack*.

A basis ($\underline{T}$, $\underline{F}$) defines for each $\tau \in \underline{T}$ a *term language* $\underline{L}_\tau$ which is the smallest set defined by:

(i) if $f : \rightarrow \tau$ then $f \in \underline{L}_\tau$;

(ii) if $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, $n \geq 1$, and if $t_1 \in \underline{L}_{\tau_1}$, ..., $t_n \in \underline{L}_{\tau_n}$ then $f(t_1, \ldots, t_n) \in \underline{L}_\tau$.

The elements of $\underline{L}_\tau$ are called *terms (of type* τ*)*

Term languages bear similarities with the carrier set of the word algebra of [ADJ 78] and the tree language of [GHM 78 b]. As an essential difference constructors are syntactical entities used in the construction of words of a formal language and are not going to be interpreted as functions.

Henceforth only bases ($\underline{T}$, $\underline{F}$) will be considered where $\underline{T}$ contains (at least) the type *Boolean* and $\underline{F}$ the constructors

<u>true</u>: → *Boolean*

<u>false</u>: → *Boolean*

2.2 The syntactical functions

To each type τ (including τ = *Boolean*) are associated:

(i) a function which expresses the syntactical equality in $\underline{L}_\tau$:

$$\text{Equal.}\tau \; : \underline{L}_\tau^2 \rightarrow \underline{L}_{\text{Boolean}}$$

(ii) the function

$$\text{If-then-else}_\tau = \underline{L}_{Boolean} \times \underline{L}_\tau \times \underline{L}_\tau \to \underline{L}_\tau :$$

$$\text{If-then-else}_\tau\ (b, s, t) = \begin{cases} s & \text{if } b = \underline{true} \\ t & \text{if } b = \underline{false}. \end{cases}$$

To each constructor $f : \tau_1 \times \ldots \times \tau_n \to \tau$ are associated:

(i) a function which expresses that the leftmost constructor of a term is f:

$$\text{Is.f} : \underline{L}_\tau \to \underline{L}_{Boolean}$$

(ii) a function which extracts the $i^{\underline{th}}$ component, $1 \le i \le n$:

$$(\text{Arg}_i.f) : \{t \in \underline{L}_\tau \mid \text{Is.f}(t) = \underline{true}\} \to \underline{L}_{\tau_i} :$$

$$(\text{Arg}_i.f)(f(t_1,\ldots,t_n)) = t_i \quad ;$$

(iii) a function which constructs an element of $\underline{L}_\tau$:

$$(\text{Cons.f}) : \underline{L}_{\tau_1} \times \ldots \times \underline{L}_{\tau_n} \to \underline{L}_\tau \quad :$$

$$(\text{Cons.f})(t_1,\ldots,t_n) = f(t_1,\ldots,t_n).$$

When no ambiguity results we write If-then-else, t[i] and $f(t_1,\ldots,t_n)$ instead of If-then-else$_\tau$, $(\text{Arg}_i.f)(t)$ and $(\text{Cons.f})(t_1,\ldots,t_n)$ respectively.

Note the notational convention that constructors start with a lower-case, functions with an upper-case letter.

2.3 Structural induction

The principle of structural induction ([Bu 69, Au 79]) is applicable in a proof of a property of a term language. As an example, assume the constructors of type τ to be:

$$f_0 : \to \tau$$

$$f_1 : \tau \times \tau' \to \tau$$

with $\tau' \neq \tau$; for proving the property:

for all $t \in \underline{L}_\tau$: $q(t)$ holds it suffices to prove that:

(i) (*base step*) $q(f_0)$ holds

(ii) (*induction step*) for all $t \in \underline{L}_\tau$ and all $t' \in \underline{L}_{\tau'}$:

if $q(t)$ holds, then $q(f_1(t, t'))$ holds.

3. A formalism for computable functions

3.1 The formalism chosen

In order to provide a sound theoretical basis we decided to use (pure) LCF [Mi 72]. In this strongly typed formalism a function is described as an LCF-term; an LCF-term is build up from constants, variables and functions by composition, λ-abstraction and minimal fixpoint abstraction. Minimal fixpoint abstraction is expressed with the help of the operator α: if t is an LCF-term and M a function variable, [αM.t] denotes the minimal fixpoint of [λM.t].

A typical definition in this formalism is

Factorial = [αM.[λn ∈ Integer.
    if Is.zero(n) then suc(zero)
        else Mul(n, M(Pred(n)))]]

For more detailed descriptions of LCF and its foundations the reader is referred to [Mi 72, Mi 73, St 77].

3.2 The domains

The interpretation of the LCF-formalism requires the domains to be complete partial orders and the basic functions to be continuous [Mi 73]. The domains and functions introduced in Section 2 have therefore to be extended.

To this end we add to each term language $\underline{L}_\tau$ the elements ω_τ and Ω_τ called the *bottom* (or: *undefined*) element and the *top* (or: *error*) *ele-*

ment of type $\tau^{(*)}$; thus we define

$$\underline{\tau} = \underline{L}_\tau \cup \{\omega_\tau, \Omega_\tau\}$$

the *extended term language of type* τ. This language $\underline{\tau}$ together with the relation $\sqsubseteq_\tau$ defined by

$$(t_1 \sqsubseteq_\tau t_2) \underset{\text{def}}{\Longleftrightarrow} (t_1 = \omega_\tau) \text{ or } (t_1 = t_2) \text{ or } (t_2 = \Omega_\tau)$$

constitutes a complete partial order, as illustrated by Figure 1.

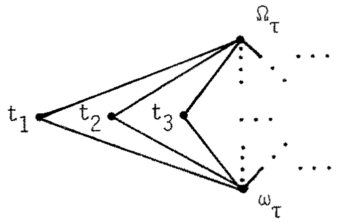

FIGURE 1: The cpo $(\underline{\tau}, \sqsubseteq_\tau)$ with $\underline{L}_\tau = \{t_1, t_2, t_3, \ldots\}$

The index τ of ω_τ and Ω_τ is omitted whenever no ambiguity arises.

The extension of an arbitrary syntactical function f different from If-then-else is the doubly strict function f_e defined by

$$f_e(t_1,\ldots,t_n) = \begin{cases} t_j & \text{if for some } j,\ 1 \le j \le n\text{: } (t_j = \omega \text{ or } t_j = \Omega) \text{ and} \\ & \quad (\text{none of the } t_{j+1},\ldots,t_n \text{ is } \omega \text{ or } \Omega)\ (**) \\ f(t_1,\ldots,t_n) & \text{otherwise} \quad (\text{cf. [St 77], p. 178}) \end{cases}$$

(*) Strictly speaking it is sufficient to introduce ω_τ. We moreover introduce Ω_τ for being able to distinguish between a result ω_τ corresponding to (possibly undecidable) non-termination and a result Ω_τ corresponding to a (decidable) "meaningless" computation.

(**) Intuitively this condition corresponds to computing the arguments from right to left.

The extension of the If-then-else$_\tau$ function is defined by:

$$\text{If-then-else}_{\tau,e}\,(b,\ s,\ t) = \begin{cases} b & \text{if } b = \omega_\tau \text{ or } b = \Omega_\tau \\ \text{If-then-else}_\tau\,(b,\ s,\ t) & \text{otherwise.} \end{cases}$$

This indices e and τ,e of the extended functions are omitted whenever no ambiguity results.

4. The algorithmic specification method

4.1 The specification of a data type

An *algorithmic specification* of the data type τ consists of

(i) a list of the constructors of type τ;

(ii) the definition of a subset $\underline{L}_\tau^0$ of the term language $\underline{L}_\tau$ by means of a doubly strict function Is.τ, called *acceptor function*:

$$\kappa \in \underline{L}_\tau^0 \underset{\text{Def}}{\Longleftrightarrow} \text{Is.}\tau(\kappa) = \underline{\text{true}};$$

(iii) the definition of (a doubly strict extension over $\underline{\tau}^2$ of) an equivalence relation over $\underline{L}_\tau$, noted Eq.τ; (*)

(iv) the definition of some functions; these functions, together with the syntactical function If-then-else$_{\tau,e}$ and the equivalence relation Eq.τ constitute the *external functions*, viz. the functions which are at the disposal of the user of the data type;

(v) the definition of some functions called *auxiliary* or *hidden*.

Note that none of the external or auxiliary functions has to be doubly strict.

The constants and functions which may occur in the right-hand sides of the function definitions in (ii) to (v) are:

(a) the elements of $\underline{L}_\tau \cup \{\Omega\}$; these elements constitute constants; (**)

(b) the (extended) syntactical functions of type τ;

(*) Do not confound Eq.τ with the syntactical function Equal.τ.

(**) ω is the "result" of a non-terminating computation and is therefore not representable as a constant.

(c) the <u>external</u> functions of other data types;

(d) the external and auxiliary functions of type τ, provided there exists no. sequence of external and/or auxiliary functions

$$F_1, F_2, \ldots, F_n \quad , \quad n \geq 2$$

where F_{i+1} occurs in the (right-hand side of the) definition of F_i, $1 \leq i \leq n - 1$, and $F_n = F_1$.

```
(i)   Constructors
           emptyset : → Set
           insert: Set x Integer → Set
(ii)  Acceptor function
           Is.Set = [αM.[λs ∈ Set. if Is.emptyset(s)
                          then true
                          else if Memberof(s[1], s[2])
                               then false
                               else M(s[1]) ]]
(iii) Equivalence relation
           Eq.Set = [λs1, s2 ∈ Set.
                          if Subset(s1, s2)
                          then Subset(s2, s1) else false  ]
(iv)  External functions
           Emptyset = emptyset
           Insert = [λs ∈ Set, i ∈ Integer.
                          if Memberof(s, i) then s else insert(s, i)]
           Delete = [αM.[λs ∈ Set, i ∈ Integer.
                          if Is.emptyset(s)
                          then emptyset
                          else if Eq.Integer(s[2], i)
                               then s[1]
                               else insert(M(s[1], i), s[2]) ]]
           Memberof = [αM.[λs ∈ Set, i ∈ Integer.
                          if Is.emptyset (s)
                          then false
                          else if Eq.Integer(s[2], i)
                               then true
                               else M(s[1], i) ]]
           Subset = [αM.[λs1, s2 ∈ Set.
                          if Is.emptyset(s1)
                          then true
                          else if Memberof(s2, s1[2])
                               then M(s1[1], s2)
                               else false  ]]
```

FIGURE 2: A specification of the data type *Set*; the data type *Integer* is assumed to have been specified. Note that Is.Set avoids the occurrence of duplicates in the term language and that Eq.Set identifies sets which differ only by the order of occurrence of their elements.

```
(i)    Constructors
                emptystackl: → Stackl
                pushl: Stackl x Integer → Stackl
(ii)   Acceptor function
                Is.Stackl = [λs ∈ Stackl. if Depth(s) ≤ 10
                                                 then true else false]
(iii)  Equivalence relation
                Eq.Stackl = Equal. Stackl
(iv)   External functions
                Emptystackl = emptystackl
                Pushl = [λs ∈ Stackl,  i ∈ Integer.
                      if Depth(s) < 10 then pushl(s, i) else Ω]
                Popl = [λs ∈ Stackl.
                      if Is.pushl(s) then s[1] else Ω]
                Topl = [λs ∈ Stackl.
                      if Is.pushl(s) then s[2] else Ω]
                Isnewl = [λs ∈ Stackl.
                      if Is.pushl(s) then false else true ]
(v)    Auxiliary function
                Depth = [αM.[λs ∈ Stackl.
                      if Is.pushl(s) then M(s[1]) + 1 else 0 ]]
```

FIGURE 3: A specification of the data type *Stackl* (*); the data type *Integer* (with the functions "≤", "<", etc.) is assumed to have been specified. The stack can not contain more than ten integers.

(*) In Bavarian "Stackl" means "little stack"

Examples of specifications are in Figure 2 and 3. More elaborate examples - including the traversable stack of [Mj 79] and the Turing machine may be found in [Lo 81].

For the data type *Set* of Figure 2 one has for instance

Delete(insert(insert(emptyset, zero), suc(zero)), zero)
= insert(Delete(insert(emptyset, zero), zero), suc(zero))
because of the fixpoint property
= insert(emptyset, suc(zero))

For the data type *Stackl* of Figure 3 one has for instance

Pushl(Popl(Emptystackl), Zero)
= Pushl(Popl(emptystackl), Zero)
= Pushl (Ω, Zero)
= <u>if</u> Depth (Ω) < 10 <u>then</u> ...
= <u>if</u> (<u>if</u> Is.pushl (Ω) <u>then</u> ...) <10 <u>then</u> ...
= <u>if</u> Ω < 10 <u>then</u> ...
because Is.pushl is doubly strict
= Ω if we assume that in the specification of *Integer* "<" has been defined as a doubly strict function.

4.2 Two more definitions

The specification of a data type τ is said to *depend* on a data type τ', $\tau' \neq \tau$, if it makes use of an external function of τ'.

A (finite) set of specifications is called ***hierarchical*** if it is possible to order its elements

$$S_1, S_2, S_3, \ldots, S_n$$

such that

- the specification S_i depends only on types specified by $S_1, S_2, \ldots, S_{i-1}$, for all i, $1 \leq i \leq n$;
- the specification S_1 is a specification of *Boolean*.

4.3 The algebra defined by a hierarchical set of specifications

The data types defined by a hierarchical set of specifications constitute a heterogeneous algebra. This algebra is defined by its carrier sets C_τ (viz. one carrier set $\underline{C}_\tau$ for each type τ) and its operations F_{op} (viz. one operation F_{op} for each external function F).

For each $t \in \underline{L}_\tau^o$ let [t] denote the equivalence class of t induced by Eq.τ on the set $\underline{L}_\tau^o$. Then the carrier set for type τ is

$$C_\tau = \{[t] \mid t \in \underline{L}_\tau^o\} \cup \{UNDEFINED_\tau, ERROR_\tau\}$$

where $UNDEFINED_\tau$ and $ERROR_\tau$ are two new elements.

Let φ be the function

$$\varphi : \bigcup_\tau (\underline{L}_\tau^o \cup \{\omega_\tau, \Omega_\tau\}) \rightarrow \bigcup_\tau \underline{C}_\tau :$$

$$\varphi(t) = \begin{cases} [t] \text{ if } t \in \underline{L}_\tau^o \text{ for some } \tau \\ UNDEFINED_\tau \text{ if } t = \omega_\tau \text{ for some } \tau \\ ERROR_\tau \text{ if } t = \Omega_\tau \text{ for some } \tau \end{cases}$$

To each external function

$$F : \underline{\tau}_1 \times \ldots \times \underline{\tau}_n \rightarrow \underline{\tau} \quad , n \geq o \quad ,$$

corresponds an operation

$$F_{op} : \underline{C}_{\tau_1} \times \ldots \times \underline{C}_{\tau_n} \rightarrow \underline{C}_\tau :$$

$$F_{op}(\varphi(t_1),\ldots, \varphi(t_n)) = \varphi(F(t_1,\ldots,t_n))$$

Note that for the definition of F_{op} to be consistent the function F must satisfy certain conditions; roughly speaking, F has to preserve the predicates Is.τ and Eq.τ; the study of these conditions is the subject of Section 5.

Note that the operation $(Eq.\tau)_{op}$ is the equality in the carrier set $\underline{C}_\tau$ or, more precisely, is an extension of the equality in $\underline{C}_\tau - \{ERROR_\tau, UNDEFINED_\tau\}$ (*).

(*) The difference between $(Eq.\tau)_{op}$ and the equality "=" in $\underline{C}_\tau$ may be illustrated as follows : $ERROR_\tau = ERROR_\tau$ but, if Eq.τ is doubly strict, $(Eq.\tau)_{op}(ERROR_\tau, ERROR_\tau) = ERROR_\tau$

Note that φ may be viewed as an epimorphism from the (heterogeneous) algebra constituted by the (extended) term languages and the external functions into the algebra defined by the specifications.

4.4 A few "practical" comments

The purpose of the acceptor function Is.τ is to eliminate some elements of the term language from consideration. For instance, in the data type *Set* of Figure 2 the attention is restricted to terms without duplicates.

The introduction of the equivalence relation Eq.τ allows one to "identify" terms which are syntactically different. In the data type *Set* of Figure 2 terms are defined to be equivalent if they are syntactically equal or if they differ only by the order of occurrence of their elements.

In general there exist several possible algorithmic specifications for a given data type which are more or less "natural". These specifications differ by the choice for the constructors and the acceptor function; for instance, replacing the acceptor function in Figure 2 by

$$\text{Is.Set} = [\lambda s \in \underline{\text{Set}}.\ \underline{\text{true}}]$$

(and modifying the definition of Eq. Set and of the external functions accordingly) leads to a specification with duplicates defining the same data type *Set*.

It is important to distinguish between the equivalence relation Eq.τ, the equality relation $(\text{Eq}.\tau)_{op}$ in the carrier set $\underline{C}_\tau$, the equality relation Equal.τ in the term language $\underline{L}_\tau$ and the relation "=" used in the definitions of the functions in the specification; "=" expresses the equality of (possibly 0-ary) functions having arguments and values in the extended term languages. (*)

(*) More precisely, "x = y" stand for "x $\sqsubseteq_\tau$ y" and "y $\sqsubseteq_\tau$ x"; while Equal.τ is doubly strict, "=" is not even monotone [Mi 72].

4.5 Proofs

For proving a property of a data type it is, roughly speaking, sufficient to prove the corresponding property of the term language; in the latter poof one may use structural induction - as indicated in Section 2.3. As a precise description and formal justification of this proof methodology is beyond the scope of this paper, we illustrate it by an example.

For proving the property of the data type *Set* of Figure 2:

for all $s \in \underline{C}_{Set}$, $i \in \underline{C}_{Integer}$:

if s and i have defined, non-error values

then $Memberof_{op}$ $(Delete_{op}(s, i), i) = \underline{false}_{op}$

one proves:

for all $s \in \underline{L}_{Set}$, $i \in \underline{L}_{Integer}$

if Is.Set(s) = Is.Integer(i) = true

then Memberof (Delete (s, i), i) = false

Structural induction on s leads to:

- (base step) s = emptyset:

  Memberof (Delete(emptyset, i), i)

  = Memberof(emptyset, i) = false

- (induction step) s = insert (s', j):

  1rst case: Eq. Integer (i, j) = true

  Memberof (Delete(insert(s', j), i), i)

  = Memberof (s', i)

  = Memberof (s', j)

  because Eq.Integer (i, j) = true (see also Section 5)

  = false

  as may be deduced from Is.Set(insert(s', j), i) = true

  2nd case: Eq.Integer (i, j) = false

  Memberof (Delete(insert(s', j), i), i)

  = Memberof (Delete(s', i), i)

  = false

  by induction hypothesis

5. The verification of a specification

5.1 Introductory remark

The verification of a specification consists in verifying the consistency of the definitions of Section 4.3.

The verification of a specification does not include a proof of the syntactical correctness which should, among other things, make sure that the right-hand sides of the function definitions are correctly typed LCF-terms. It is also different from a (semantical) "correctness proof" checking that the data type defined corresponds to the "intended" one - whatever this means.

5.2 The properties to be verified

The definitions of Section 4.3 are consistent provided the following three conditions hold:

(i) Eq.τ is an equivalence relation, i. e. Eq.τ is a total, reflexive, symmetric and transitive relation; this condition has to be verified because the definition of Eq.τ merely guarantees that Eq.τ is a (possibly partial) function with values of type *Boolean*;

(ii) each of the external functions preserves the equivalence relation, i. e. equivalent arguments lead to equivalent values;

(iii) each external function preserves the property Is.τ, i. e. the function value satisfies Is.τ if the arguments do.

More formally the *verification conditions* of a specification of the data type τ are:

(i) if $Is.\tau(t) = Is.\tau(t_1) = Is.\tau(t_2) = Is.\tau(t_3) = \underline{true}$
then:

(a) either $Eq.\tau(t_1, t_2) = \underline{true}$ or $Eq.\tau(t_1, t_2) = \underline{false}$

(b) $Eq.\tau(t, t) = \underline{true}$

(c) $Eq.\tau(t_1, t_2) = Eq.\tau(t_2, t_1)$

(d) if $Eq.\tau(t_1, t_2) = Eq.\tau(t_2, t_3) = \underline{true}$
then $Eq.\tau(t_1, t_3) = \underline{true}$

(ii) for each external function, say

$$F : \underline{\tau}_1 \times \ldots \times \underline{\tau}_n \to \underline{\tau} \quad , \quad n \geq 0$$

one has

for all terms t_i and t_i' of type τ_i, $1 \le i \le n$:

if $Is.\tau_i(t_i) = \underline{true}$ or $t_i = \omega$ or $t_i = \Omega$ for all i, $1 \le i \le n$,

and if $Is.\tau_i(t_i') = \underline{true}$ or $t_i' = \omega$ or $t_i' = \Omega$ for all i, $1 \le i \le n$,

and if $Eq.\tau_i(t_i, t_i') = \underline{true}$ or $t_i = t_i' = \omega$ or $t_i = t_i' = \Omega$ for all i, $1 \le i \le n$,

then $Eq.\tau(F(t_1, \ldots, t_n), F(t_1', \ldots, t_n')) = \underline{true}$

or $F(t_1, \ldots, t_n) = F(t_1', \ldots, t_n') = \omega$

or $F(t_1, \ldots, t_n) = F(t_1', \ldots, t_n') = \Omega$

(iii) for each external function, say

$$F : \underline{\tau}_1 \times \ldots \times \underline{\tau}_n \to \underline{\tau}, \quad n \ge 0$$

one has:

for all terms t_i of type τ_i, $1 \le i \le n$:

if $Is.\tau_i(t_i) = \underline{true}$ or $t_i = \omega$ or $t_i = \Omega$ for all i, $1 \le i \le n$

then $Is.\tau(F(t_1, \ldots, t_n)) = \underline{true}$

or $F(t_1, \ldots, t_n) = \omega$

or $F(t_1, \ldots, t_n) = \Omega$

These verification conditions are very similar to those of [GHM 78 b].

5.3 A worked-out example

The verification of the specification of the data type *Set* of Figure 2 has been performed mechanically with the AFFIRM-System [Mu 80, Th 79]; the proofs may be found in [Lo 80 a].

6. Concluding remarks

Algorithmic specifications have been shown to provide an elegant way for handling partial and ERROR-functions; they allow to define any data types with recursively enumerable carrier sets and partial computable functions; they do not require proofs of consistency and sufficient-completeness; finally, enrichment, i. e. the addition of external functions raises no problems. In [Lo 80 c] it is shown that the specification method leads to a simple definition of the implementation of abstract data types.

Similar constructive approaches are in [Ca 80, Kl 80]. As a main difference these authors do not use a formal language such as the term language introduced here. [Kl 80] moreover only considers primitive recursive functions.

In simple cases it is easy to tranform algorithmic specifications into algebraic ones. The main idea consists in deriving from the function definitions relations between their values; the values have to be chosen such that the syntactical functions - except If-then-else - are eliminated. For more details the reader is referred to [Lo 80 a, Lo 80 b].

Proofs of properties of data types may be performed mechanically either by first transforming the specifications into algebraic ones and by then using a system such as AFFIRM [Mu 80, Th 79, Lo 80 a], or directly by using a system based on the LCF-calculus [Mi 72, GMW 79].

The specification of parameterized data types and of data types with non-deterministic operations has not yet been examined.

Acknowledgment

The author is indebted to one of the referees for his detailed and helpful comments.

References

[Au 79] R. Aubin, "Mechanizing structural induction, part 1", Theoretical Computer Science 9, 3, pp. 329 - 345 (1979)

[BDP 79] M. Broy, W. Dosch, H. Partsch, P. Pepper, M. Wirsing, "Existential quantifiers in abstract data types", Proc. ICALP, Lect. Notes in Comp. Sc. 71, Springer-Verlag, pp. 71 - 87 (1979)

[BM 80] M. Broy, M. Wirsing, "Partial recursive functions and abstract data types", EATCS Bull. n^{o} 11, pp. 34 - 41 (June 1980)

[Bu 69] R. M. Burstall, "Proving properties of programs by structural induction", Comp. J. 12, pp. 41 - 48 (1969)

[Ca 80] R. Cartwright, "A constructive alternative to axiomatic data type definitions", Internal Report TR 80 - 427, Cornell University, June 1980

[EKP 78] H. Ehrig, H. J. Kreowski, P. Padawitz, "Stepwise specification and implementation of abstract data types", Proc. ICALP, Lect. Notes in Comp. Sc. 62, pp. 203 - 226, Springer-Verlag, 1978

[ADJ 78] J. A. Goguen, J. W. Thatcher, E. G. Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types", in "Current Trends in Programming Methodology IV" (R. Yeh, ed.), pp. 80 - 149, Prentice-Hall, 1978

[Go 78] J. A. Goguen, "Abstract errors for abstract data types" in E. J. Neuhold (ed.), "Formal description of programming concepts", North-Holland, 1978

[GMW 79] M. Gordon, R. Milner, C. Wadsworth, "Edinburgh LCF", Lecture Notes in Computer Science 78, Springer-Verlag, 1979

[GH 78 a] J. V. Guttag, J. J. Horning, "The algebraic specifications of abstract data types", Acta Informatica 10, 1, pp. 27 - 52 (1978)

[GHM 78 b] J. V. Guttag, E. Horowitz, D. R. Musser, "Abstract data types and software validation", Comm. ACM 21, 12, pp. 1048 - 1064

[Gu 80] J. V. Guttag, "Notes on type abstraction", IEEE Trans. on Softw. Eng. SE-6, 1, pp. 13 - 23 (1980)

[Ka 79] D. Kapur, "Specifications of Majster's traversable stack and Veloso's traversable stack", SIGPLAN Notices 14, 5, pp. 46 - 53 (1979)

[Kl 80] H. Klaeren, "A simple class of algorithmic specifications for abstract software modules", Proc. ICALP, Lect. Notes in Comp. Sc. 88, pp. 362 - 374, Springer-Verlag, 1980

[Lo 80 a] J. Loeckx, "Proving properties of algorithmic specifications of abstract data types in AFFIRM", AFFIRM-Memo-29-JL, USC-ISI, Marina del Rey, 1980

[Lo 80 b] J. Loeckx, "Algorithmic specifications of abstract data types", Internal Report A 80/12, Fachbereich 10, Universität des Saarlandes, Saarbrücken, 1980

[Lo 80 c] J. Loeckx, "Implementations of abstract data types and their correctness proofs", Internal Report A 80/13, Fachbereich 10, Universität des Saarlandes, Saarbrücken, 1980

[Lo 81] J. Loeckx, "Algorithmic specifications of some non-trivial data types", Internal Report, Fachbereich 10, Universität des Saarlandes, to appear

[Mj 79] M. E. Majster, "Data types, abstract data types and their specification problem", Theor. Comp. Sc. 8, 1, pp. 89 - 127 (1979)

[Mi 72] R. Milner, "Implementation and application of Scott's logic for computable functions", Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices 7, 1, pp. 1 - 6 (1972)

[Mi 73] R. Milner, "Models of LCF", Stanford University Memo AIM-186, January 1973

[Mo 80] A. Moitra, "A note on algebraic specifications of binary trees", SIGPLAN Notices 15, 6, pp. 64 - 67 (1980)

[Mu 80] D. R. Musser, "Abstract data type specification in the AFFIRM System", IEEE Trans. on Softw. Eng. SE-6, 1, pp. 24 - 32 (1980)

[St 77] J. E. Stoy, "Denotational semantics: The Scott-Strachey approach to programming language theory", MIT-Press, Cambridge (Mass.) 1977

[Th 79] D. H. Thompson (Ed.), "AFFIRM Reference Manual", Internal Report, USC-ISI, Marina del Rey, 1979

[WPP 80] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy, "On hierarchies of abstract data types", Internal Report TUM-I 8007, Technische Universität München, Mai 1980

[Zi 74] S. N. Zilles, "Algebraic specifications of data types", Comp. Struct. Group Memo 119, Lab. for Comp. Sc., MIT, 1974