JOURNAL OF COMPUTER AND SYSTEM SCIENCES 23, 223–253 (1981)

Complexity of Algebraic Implementations for Abstract Data Types

HARTMUT EHRIG AND BERND MAHR

TU Berlin, Fachbereich Informatik, Berlin 10, West Germany

Received September 5, 1980; revised February 18, 1981

A notion of complexity for algebraic implementations of abstract data types is introduced and studied. The main results concern the expressive power of algebraic specifications and implementations as well as upper and lower bounds on the complexity of implementations in terms of time on Turing Machines.

1. INTRODUCTION

In the last 5 years algebraic specification techniques for abstract data types have been studied in various papers from the theoretical and the application points of view. Probably the first precise mathematical approach was given by the ADJ-group in [1]. A first attempt towards an algebraic specification language, named CLEAR, was given by Burstall and Goguen in [5]. Up to now much less attention was paid to the problem of implementation of abstract data types although Guttag had already given an algebraic version of an implementation of symbol tables by stacks in [16]. For first approaches to an algebraic implementation concept we refer to ADJ [1] Goguen and Nourani [15], Ehrich [7, 8], Wand [24] and Lehmann and Smyth [20]. In contrast to most of these papers in our papers [9, 12] a formal concept is proposed which clearly distinguishes between a syntactical and a semantical level and gives correctness conditions for both levels. The full concept is thoroughly discussed in [10]. In this paper, which extends part of our STOC-paper [13], complexity of algebraic implementations for abstract data types is introduced and studied. This seems to be new within the algebraic theory of abstract data types; some work, however, has been published on compatibility of algebraic specifications and particular complexity results concerning algebras and their algebraic specifications (see, for example, [6, 17, 19, 21]. Our motivation for introducing a concept of complexity of implementations on the level of algebraic specifications was manyfold.

One reason was the need for a measure for implementations which enables one to compare implementations with respect to their efficiency. Such a comparison should be possible in both respects, relative and absolute. In other words we want to give a meaning to "implementation one is more efficient than implementation two" and "implementation x is most efficient."

0022-0000/81/050223-31$02.00/0

To consider the implementation concept as a computational model was another reason for looking at the complexity of implementations. It was the operational aspect of implementations which came to light a bit more when we were searching for an appropriate notion of complexity. Compatibility of implementations with Turing Machines is a result in this respect.

Finally from the complexity theoretic point of view it was desirable to have a precise notion of complexity for abstract data types, which provides a frame for complexity analysis. It should not be as far from a natural way of programming as time of Turing Machines (yet compatible with its) but in contrast to steps of ALGOL-programs it should be highly sensitive, flexible and mathematically well-defined. Even though we do not discuss in this paper how the concept relates to these imaginations, the reader may look at the definitions and results with this motivation in mind.

In Section 2 we briefly review the basic notions of algebraic specifications and implementations. Unfortunately we cannot present the implementation concept in more detail here since it would go beyond the scope of this paper. The reader is therefore referred to a full presentation in [10].

Section 3 contains the definitions of "cost" for terms and "complexity" for operations both with respect to a given algebraic implementation. Our complexity corresponds to worst case complexity which is representation independent. An implementation of sets by strings using hash tables, is analyzed as an example.

Section 4 is a preparation for Section 5 and of interest in its own right. It contains the algebraic specification of time bounded Turing Machines and the functions which they compute. The main result of this section is that total recursive functions $f: I^* \to \bar{O}^*$ have algebraic specifications.

In Section 5 we give a formal definition of the "function implementation problem" which is, roughly speaking, to find a specification and an implementation for a function $f: I^* \to \bar{O}^*$. It is shown that the function implementation problem is solvable for total recursive functions and any upper deterministic time bound for f with respect to an "algebraic time bounded" Turing Machine is also an upper bound for a solution of the function implementation problem for f.

Finally Section 6 shows that any function f which has a solvable function implementation problem, is total recursive. Furthermore a lower bound result is given which states that the logarithm of any lower bound on the nondeterministic time for f is also a lower bound on the complexity of the function implementation problem for f.

The results in this paper, however, are intended to be only a first step towards a unified theory of relative complexity for abstract data types. Much more remains to be done for such a theory. Just for our results, the more or less rough estimations for upper and lower bounds need considerable improvement. In the long run we hope that algebraic implementations will turn out to be a new computational model which is both general and flexible.

2. Algebraic Specifications and Implementations

The foundations for a strict mathematical theory of algebraic specifications were given by the ADJ-group in [1], while first approaches on how to use algebraic specifications for the design of software systems were given already by Zilles [25] and Guttag [16]. The main idea of the ADJ-approach is to give a syntactic description of an abstract data type using algebraic specifications. The semantics of the specification is given by the corresponding quotient term algebra (or any isomorphic algebra) which is the initial algebra in the category of all algebras satisfying the given specification. This is the reason for referring to the ADJ-approach as the "initial algebra approach," while the approach of some other authors, initiated by Wand [24], is called the "final algebra approach." We will follow the ADJ-approach as given in [1] and continued in [11]. In this section we give a short review of the concepts. An *(algebraic) specification* $\mathrm{SPEC} = \langle S, \Sigma, E \rangle$ consists of a set S of *sorts*, a family of sets $\Sigma = (\Sigma_{w,s})_{w \in S^*, s \in S}$ of *operation symbols*, and a family of sets $E = (E_s)_{s \in S}$ of *equations*. The pair (S, Σ) will also be called a *signature*. Operation symbols $\sigma \in \Sigma_{w,s}$, are written $\sigma: w \to s$ with domain $w = s1 \cdots sn$ $(si \in S,\ i = 1,..., n)$ and range $s \in S$. In the special case $w = \lambda$ (empty word) σ is called *0-ary* or *constant*. We will usually refer to operation symbols σ as *operations*, for simplicity. Terms built up from the operations of a specification are defined in the usual way and form an algebra $T_\Sigma = (T_{\Sigma,s})_{s \in S}$, called *term algebra*. Given a family of sets of variables $X = (X_s)_{s \in S}$, then the terms with variables of sort $s \in S$ form a set $T_\Sigma(X)_s$. An *equation* $e = (L, R) \in E_s$ of sort s is a pair of terms L, R with variables from $T_\Sigma(X)_s$. More intuitively equations are written as $L = R$.

The semantics of a specification $\mathrm{SPEC} = \langle S, \Sigma, E \rangle$ is given by a (many-sorted) Σ-algebra with data domains and operations corresponding to S and Σ, and which satisfies the equations E. More precisely the *semantics* of SPEC is the initial SPEC-algebra T_{SPEC} which is uniquely determined up to isomorphism and hence is representation invariant as required for abstract data types. A canonical construction for T_{SPEC} is the *quotient term algebra*. All algebras isomorphic to T_{SPEC} for some specification SPEC will be considered as *abstract data types*. For a more detailed motivation of abstract data types see [1]. The quotient term algebra is constructed by factorization of T_Σ by the congruence relation $\equiv_E$ generated by the equations E in SPEC. In fact the quotient term algebra is initial, i.e., for all other SPEC-algebras A there is exactly one Σ-homomorphism $f: T_{\mathrm{SPEC}} \to A$.

In order to discuss correctness of specifications we need a precise notion of semantics. In general, a specification SPEC is correct with respect to a SPEC-algebra A if A is isomorphic to the quotient term algebra T_{SPEC}, or equivalently A is an initial SPEC-algebra. But we want to allow "hidden" functions in the specifications which are not part of the signature of the algebra A and therefore have a little more subtle definition of correctness:

A *specification* SPEC′ is *correct* with respect to a SPEC-algebra A if $\mathrm{SPEC} = \langle S, \Sigma, E \rangle$ is a subspecification of $\mathrm{SPEC}' = \langle S', \Sigma', E' \rangle$, i.e., all sorts, operations and equations of SPEC are also in SPEC′, and the restriction of $T_{\text{SPEC}'}$ to the signature of SPEC is isomorphic to A.

This more generalized notion of correctness is introduced in [11] where also the following concepts are discussed in more detail.

An essential tool in the writing and use of specifications is the concept of combination:

Using $+$ for (componentwise) disjoint union, we say that $\text{SPEC}\,0 = \text{SPEC} + \langle S0, \Sigma 0, E0 \rangle$ is a *combination* if $\text{SPEC} = \langle S, \Sigma, E \rangle$ and $\text{SPEC}\,0 = \langle S + S0, \Sigma + \Sigma 0, E + E0 \rangle$ are specifications. Note, $\langle S0, \Sigma 0, E0 \rangle$ is not assumed to be a specification itself because it is allowed that equations in $E0$ involve operations in Σ and operations in $\Sigma 0$ involve sorts in S. A combination is called an *extension* if the restriction of $T_{\text{SPEC}\,0}$ to the signature of SPEC, written $(T_{\text{SPEC}\,0})_{\text{SPEC}\,1}$, is isomorphic to T_{SPEC}. This means that the combination protects the semantics of SPEC. An extension SPEC 0 of SPEC is called an *enrichment* if it satisfies $S0 = \varnothing$.

As already introduced we write the restriction of a SPEC′-algebra A to the signature of a subspecification $\text{SPEC} = \langle S, \Sigma, E \rangle$ of SPEC′ by $(A)_{\text{SPEC}}$ and also use $(A)_S$ or $(A)_\Sigma$ to denote the restriction just to sorts in S or operations in Σ, respectively.

In order to give the rather informal notion of "implementation" of abstract data types a precise mathematical meaning, concepts of algebraic implementations were introduced by several authors ([1, 7, 8, 15, 18, 24] and others). All these concepts describe how an abstract data type A is "simulated" by an abstract data type B. In the concept proposed in [12], this is done in clear distinction on a syntactical level and a semantical level, and correctness conditions are formulated which guarantee validity of a number of predefined conceptual requirements. We will follow this concept here and give a concise introduction. For extensive motivation of the concept the reader is referred to [10, 12]. The example below might help to show some of the primary intention and the application in mind when giving a concept of implementation of abstract data types on the level of algebraic specifications. Assume we have algebraic specifications SPEC 0 and SPEC 1 for abstract data types $A0$ and $A1$, respectively. The we describe on the "syntactical level," in terms of sorts, operations and equations, how data and operations of $A0$, represented by the specification SPEC0, are "simulated" by data and operations of $A1$, represented by the specification SPEC1. This description is formally denoted by

$$\text{SPEC1} \xRightarrow{\text{IMPL}} \text{SPEC0}.$$

On the "semantical level" the meaning of the syntactical description is given in terms of a functor SEM_{IMPL} which assigns to each SPEC1-algebra a SPEC0-algebra. Finally correctness conditions are given which assure that all data of $A0$, the implemented data type, can be represented by synthesized data of $A1$, the implementing data type, and all operations on data in $A0$ can be simulated by operations on corresponding synthesized data of $A1$.

This rough description of the implementation concept is made precise in the following definition. Throughout this paper we make the following

2.1. General Assumption. We assume we have the following algebraic specifications:

$$\begin{aligned} \text{SPEC} &= \langle S, \Sigma, E\rangle, \\ \text{SPEC0} &= \text{SPEC} + \langle S0, \Sigma 0, E0\rangle, \\ \text{SPEC1} &= \text{SPEC} + \langle S1, \Sigma 1, E1\rangle, \end{aligned}$$

where SPEC0 and SPEC1 are both extensions of SPEC (see above). SPEC is called the *common parameter part* of SPEC0 and SPEC1 (an actual parameter in the sense of [2, 3]).

2.2. Definition (Syntax, Semantics and Correctness of Implementations).

1. Given specifications SPEC0 and SPEC1 as in 2.1. An *algebraic implementation* of SPEC0 by SPEC1, formally denoted by IMPL: SPEC1 $\Rightarrow$ SPEC0, is a triple

$$\text{IMPL} = (\Sigma\text{SORT, EOP, HID})$$

consisting of

ΣSORT	a set of operations (intended to implement the sorts of SPEC0)
EOP	a set of equations (intended to implement the operations of SPEC0)
HID	a triple $\langle S\text{HID}, \Sigma\text{HID}, E\text{HID}\rangle$ of sorts, operations and equations, called *hidden part* (intended to support the implementation).

2. IMPL is called *syntactically correct* if the following *syntactical correctness conditions* are satisfied:

(1) $\text{SORTIMPL} = \text{SPEC1} + \langle S0 + S\text{HID}, \Sigma\text{SORT}, \varnothing\rangle$ is a combination,

(2) $\text{OPIMPL} = (\text{SORTIMPL} + \langle\varnothing, \Sigma\text{HID}, E\text{HID}\rangle) + \langle\varnothing, \Sigma 0, E\text{OP}\rangle$ is a combination,

(3) for all $\sigma \in \Sigma\text{SORT}$ the range of σ belongs to $S0 + \text{SHID}$.

3. The semantics of IMPL is a functorial construction, denoted by SEM_{IMPL}, and is given the following three construction steps:

$$T_{\text{SPEC1}} \xmapsto{\text{SYNTHESIS}} T_{\text{OPIMPL}} \xmapsto{\text{RESTRICTION}} \text{REP}_{\text{IMPL}} \xmapsto{\text{IDENTIFICATION}} S_{\text{IMPL}}.$$

SEM_{IMPL} assigns to the SPEC1-algebra T_{SPEC1} the SPEC0-algebra S_{IMPL}, called the *semantical algebra of* IMPL, while SYNTHESIS assigns to T_{SPEC1}, the semantics of SPEC1, the algebra T_{OPIMPL}, the semantics of OPIMPL.

RESTRICTION assigns to T_{OPIMPL} the algebra $\mathrm{REP}_{\mathrm{IMPL}}$ which is the restriction of T_{OPIMPL}, generated by $\Sigma + \Sigma 0$, i.e.,

$$\mathrm{REP}_{\mathrm{IMPL}} = \mathrm{eval}(T_{\Sigma+\Sigma 0}),$$

where eval is the unique term evaluation homomorphism

$$\mathrm{eval}\colon T_{\Sigma+\Sigma 0} \to (T_{\mathrm{OPIMPL}})_{\Sigma+\Sigma 0}.$$

IDENTIFICATION assigns to the $(\Sigma + \Sigma 0)$-algebra $\mathrm{REP}_{\mathrm{IMPL}}$ the SPEC-algebra S_{IMPL} which is the quotient of $\mathrm{REP}_{\mathrm{IMPL}}$ with respect to the congruence relation $\equiv_{E0}$, generated by $E0$

$$S_{\mathrm{IMPL}} = \mathrm{REP}_{\mathrm{IMPL}}/_{\equiv_{E0}}.$$

4. IMPL is called *semantically correct* if the following *semantical correctness conditions* are satisfied:

(1) for all terms $t \in T_{\Sigma+\Sigma 0}$ there is $t' \in T_{\Sigma+\Sigma 1+\Sigma\mathrm{SORT}}$ such that

$$t \equiv_{E(\mathrm{OPIMPL})} t' \qquad \text{(OP-}\textit{completeness}\text{)},$$

(2) S_{IMPL} is isomorphic to T_{SPEC0} (RI-*correctness*).

If not explicitly stated otherwise we will use the term "*implementation*" to denote implementations which are both syntactically and semantically correct.

The syntactical corectness conditions tell how sorts, operations and equations must look in order to constitute a syntactically correct implementation. In this case, the semantics of an implementation is well-defined.

The first semantical, correctness condition assures that all representations of SPEC0-data, i.e., terms $(\Sigma + \Sigma 0)$-operations, can be transformed into representations of SORTIMPL-data, i.e., terms of $(\Sigma + \Sigma 1 + \Sigma\mathrm{SORT})$-operations, by using the equations of OPIMPL. The second condition, called RI-correctness, assures that the implementation describes a construction, the functor $\mathrm{SEM}_{\mathrm{IMPL}}$, which starting from T_{SPEC1} ends up with T_{SPEC0}. This semantical construction is given in three steps: Beginning with T_{SPEC1} which is the semantics of SPEC1, new data and operations of signature $(S0, \Sigma 0)$ are synthesized by ΣSORT, EOP and the hidden part HID, respectively, from those of T_{SPEC1}. This leads to an extended data type T_{OPIMPL} which is the semantics of the combination OPIMPL. In the RESTRICTION step all sorts and operations, not belonging to $S0$ and $\Sigma 0$ are forgotten in T_{OPIMPL} leading to $(T_{\mathrm{OPIMPL}})_{\Sigma+\Sigma 0}$ and in $\mathrm{REP}_{\mathrm{IMPL}}$ only those data are considered which are representable by $(\Sigma + \Sigma 0)$-terms. The data of $\mathrm{REP}_{\mathrm{IMPL}}$ now are intended to represent those of T_{SPEC0}. Since it should not be allowed that different SPEC0-data are represented by the same datum in $\mathrm{REP}_{\mathrm{IMPL}}$, the RI-correctness demands that after the final IDENTIFICATION step the semantical algebra S_{IMPL} has to be isomorphic to T_{SPEC0}.

Before we give an example, we state the following facts concerning implementations, which we will use subsequently:

2.3. Facts. 1. Syntactical correctness of implementations IMPL: SPEC1 $\Rightarrow$ SPEC0 implies *type protection*, i.e.,

$$(T_{\mathrm{SORTIMPL}})_{\mathrm{SPEC1}} \cong T_{\mathrm{SPEC1}}.$$

2. A $(\Sigma + \Sigma')$-term t' of a sort in $(S + S')$ is called *Σ-normal*, if all subterms t of t' of sorts in S are already Σ-terms.

OP-completeness and (1) imply that each $(\Sigma + \Sigma0)$-term can be reduced via the equations of OPIMPL to a $(\Sigma + \Sigma1)$-normal term in SORTIMPL.

3. RI-correctness is equivalent to the existence of a surjective $(\Sigma + \Sigma0)$-homomorphism rep, called *representation homomorphism*, such that

$$\begin{tikzcd}
\mathrm{REP}_{\mathrm{IMPL}} \arrow[rr, "\mathrm{rep}"] & = & T_{\mathrm{SPEC0}} \\
 & T_{\Sigma+\Sigma0} \arrow[ul, "e"] \arrow[ur, "\mathrm{nat}"'] &
\end{tikzcd}$$

commutes, where e is the restriction of eval in 2.2 and nat the natural homomorphism from the term to the quotient term algebra.

4. Each enrichment SPEC 0 of SPEC, i.e., SPEC0 = SPEC + $\langle \varnothing, \Sigma0, E0 \rangle$ such that $(T_{\mathrm{SPEC0}})_{\mathrm{SPEC}} \cong T_{\mathrm{SPEC}}$, induces an implementation IMPL: SPEC $\Rightarrow$ SPEC 0, called *enrichment implementation*, by IMPL = $(\varnothing, E0, \langle \varnothing, \varnothing, \varnothing \rangle)$.

For a proof of these fact the reader is referred to [14].

2.4. Example. We implement sets of natural numbers by strings of natural numbers and assume we have the following specifications, given only by their signature for simplicity:

nat1 consists of the sorts *nat* and *bool* and operations *T*, *F*, AND, 0, SUCC and EQ, where *T* and *F* name the truth values "true" and "false," AND is the boolean "and," 0 names zero in the natural numbers, SUCC names the successor function and EQ denotes the binary equality predicate.

set(nat1) = nat1 +
sorts: *set*,
opns: CREATE: $\rightarrow$ *set*
  INSERT: *nat set* $\rightarrow$ *set*
  DELETE: *nat set* $\rightarrow$ *set*
  MEMBER: *nat set* $\rightarrow$ *bool*
  EMPTY: *set* $\rightarrow$ *bool*
  IF-THEN-ELSE: *bool set set* $\rightarrow$ *set*
eqns: see [13].

CREATE denotes the empty set, INSERT and DELETE name insertion and deletion of elements in sets, MEMBER names a test for existence of a given element, EMPTY denotes a test for emptyness and IF-THEN-ELSE is a "hidden" operation which denotes the distinction of cases.

string(nat1) = **nat1** +
sorts: *string*
opns: $\lambda: \to string$
ADD: *string nat* → *string*.

This specification requires no equations, thus its semantics is the term algebra. λ denotes the empty string and ADD the concatenation of a string with an element. We implement **set(nat1)** by **string(nat1)** and give an implementation

IMPL: **string(nat1)** ⇒ **set(nat1)**

which implicitly uses hash tables with m rows to represent sets.

Explicitly we have IMPL = (ΣSORT, EOP, ⟨SHID, ΣHID, EHID⟩) with

ΣSORT:
TUP: $string^m \to set$
ELEM(i): $\to nat_m$ $(i = 1,..., m)$

SHID (hidden sorts):
nat_m

ΣHID (hidden operations):
ENTRY: nat_m *set* → *string*
CHANGE: nat_m*set string* → *set*
HASH: $nat \to nat_m$
IF-THEN-ELSE-STR: *bool string string* → *string*
ADJOIN: *string nat* → *string*
REMOVE: *string nat* → *string*
SEARCH: *string nat* → *bool*
EMPTYSTR: *string* → *bool*

EHID (hidden equations):
ENTRY (ELEM(i), TUP($s1$,..., sm)) = si $(i = 1,..., m)$
CHANGE(ELEM(i), TUP($s1$,..., sm), s) =
TUP($s1$,..., $s(i-1)$, s, $s(i+1)$,..., sm) $(i = 1,..., m)$
HASH(SUCC^i(0)) = ELEM($i+1$) $(i = 0,..., m-1)$
HASH(SUCC^m(n)) = HASH(n)
ADJOIN(s, n) = IF SEARCH (s, n) THEN s ELSE ADD (s, n) STR
REMOVE(λ, n) = λ
REMOVE(ADD($s, n1$), $n2$) = IF EQ($n1, n2$) THEN s
ELSE ADD (REMOVE ($s, n2$), $n1$) STR
SEARCH(λ, n) = FALSE
SEARCH(ADD($s, n1$), $n2$) = IF EQ($n1, n2$) THEN TRUE ELSE
SEARCH($s, n2$)

EMPTYSTR(λ) = TRUE
EMPTYSTR(ADD(s, n)) = FALSE
IF TRUE THEN $s1$ ELSE $s2$ STR = $s1$
IF FALSE THEN $s1$ ELSE $s2$ STR = $s2$

*E*OP:

CREATE = TUP($\lambda,\dots,\lambda$)
INSERT(n, s) = CHANGE(HASH(n), s, ADJOIN(ENTRY (HASH(n), s), n))
DELETE(n, s) = CHANGE(HASH(n), s, REMOVE(ENTRY (HASH(n), s), n))
MEMBER(n, s) = SEARCH(ENTRY(HASH(n), s), n)
EMPTY(s) = EMPTYSTR(ENTRY(ELEM(1), s)) AND ... EMPTYSTR(ENTRY(ELEM(m), s))

We represent sets as m-tuples of strings (hash tables) which are expressed by the ΣSORT-operation TUP. nat_m is a hidden sort and represents the keys for the m-strings of the hash table. ELEM(i) for $i = 1,\dots, m$ generates the keys. The intention of the hidden operations is the following:

ENTRY results the ith string of the hash table.
CHANGE replaces the ith string of the table by another string.
HASH computes the key of an element.
ADJOIN adjoins an element to a string if it is not already in the string.
REMOVE removes an element from the string.
SEARCH tests existence of an element in a string.
EMPTYSTR tests emptyness of a string.

This intention is formalized in the hidden equations. The equations *E*OP finally express how the effect of the Σ0-operations is simulated by the implementation. The specification SORTIMPL, which is implicitly given in the implementation, here has the following form:

SORTIMPL = **string(nat1)** +
sorts: nat_m, *set*
opns: TUP: $string^m \to set$
ELEM(i): $\to nat_m$ $(i = 1,\dots, m)$.

OP-completeness, the first semantical correctness condition, now states that each term of ($\Sigma + \Sigma$0)-operations is reducible to a term of ($\Sigma + \Sigma 1 + \Sigma$SORT)-operations, using all equations of OPIMPL. We may thus reduce the term (using digits for *nat*-terms)

INSERT(3, DELETE (4, INSERT(3, INSERT(5, CREATE)))) which represents the set $\{3, 5\}$, to the term

$$\mathrm{TUP}(\mathrm{ADD}(\lambda, 3), \lambda, \mathrm{ADD}(\lambda, 5)),$$

where $m = 3$ is assumed and thus 5 has key remainder $(5/3) + 1 = 3$ and 3 has key 1. The algebra $\mathrm{REP}_{\mathrm{IMPL}}$, which is constructed in the semantics of the implementation, now can be imagined as an algebra whose elements of sort *set* are hash tables which represent sets and whose operations are those of SPEC0, but now working on hash tables instead of sets.

Finally since hash tables differ, if the order of the entries is permuted, the IDENTIFICATION-step is not trivial and we have different hash tables which represent the same set.

3. Complexity of Algebraic Implementations

Algebraic implementations as introduced in the last section form a very general and flexible model of computation. The subsequent sections may support this judgement. From this point of view it is natural to introduce notions of cost and complexity along with an implementation mainly to serve the purpose of analyzing and comparing implementations with respect to their efficiency. Even though a notion of complexity is rather arbitrary, it should meet a number of requirements. Namely, we want the notion to reflect our intuitive ideas about what is making an implementation complex or expensive. On the other hand this notion should be defined purely on syntax and semantics of implementations and should not use implicitly the time of Turing Machines or any other complexity measure. But it should be compatible with such measures and should allow us to talk about complexity of implementation problems, i.e., there should be lower bounds for the complexity of implementations. We will be concerned with these requirements in this and the subsequent sections.

Before we define functions $\mathrm{cost}_{\mathrm{IMPL}}$ and $\mathrm{comp}_{\mathrm{IMPL}}$ with respect to a given implementation, we have to expose more of the operational aspect of implementations, which lies in the use of equations as reduction rules for terms. In this respect it is helpful to look at terms with operation symbols as nodes.

Given a specification $\mathrm{SPEC} = \langle S, \Sigma, E \rangle$ and its term algebra T_Σ. As in the previous section by $\equiv_E$ we denote the congruence generated by E. Now given an equation (L, R) of E and a term $t \in T_\Sigma$ such that there is a substitution $h: X \to T_\Sigma$ for the variables in L and R which results in a pair of terms (without variables) (L_h, R_h) such that L_h is a subterm of t, then we can replace L_h in t by R_h and obtain a term t'. We say that t' is obtained by *application of* (L, R) *to* t and call this replacement a *direct reduction step from* t *to* t'. By $\mathrm{Dired}_{\mathrm{SPEC}}$ we denote *the set of direct reduction steps* which are obtained by application of the equations in E to the terms in T_Σ.

A *reduction from* t *to* t' *of length* r is a sequence of direct reduction steps producing a sequence of terms

$$t = t_1, t_2, \ldots, t_r = t'$$

such that for $i = 2, \ldots, r$ t_i is obtained from t_{i-1} by application of some equation in E or E^{-1} (E^{-1} denotes the inverse of the set of ordered pairs E). By Red_{SPEC} we denote the *set of reductions* and define for a reduction w its *first term* First $(w) = t$ and its *last term* $\text{Last}(w) = t'$ if w is a reduction from t to t'.

If $\text{SPEC} = \langle S, \Sigma, E \rangle$ and T_Σ its term algebra, then from the definition of $\equiv_E$ we obtain

FACT. $t \equiv_E t'$ if and only if there is a reduction $w \in \text{Red}_{\text{SPEC}}$ with $\text{First}(w) = t$ and $\text{Last}(w) = t'$.

3.1. DEFINITION (Computation). Given an implementation IMPL: SPEC1 $\Rightarrow$ SPEC0. Then a reduction $w \in \text{Red}_{\text{OPIMPL}}$ is called a *computation of* IMPL iff $\text{First}(w)$ is in $T_{\Sigma+\Sigma 0}$ and $\text{Last}(w)$ is in $T_{\Sigma+\Sigma 1+\Sigma\text{SORT}}$ and is $(\Sigma + \Sigma 1)$-normal.

The *set of computations of* IMPL *with first term* $t \in T_{\Sigma+\Sigma 0}$ is denoted by $\text{COM}_{\text{IMPL}}(t)$ and $\text{COM}_{\text{IMPL}} := \bigcup_{t \in T_{\Sigma+\Sigma 0}} \text{COM}_{\text{IMPL}}(t)$.

Note, from OP-completeness (first semantical correctness condition) and fact 2.3(2) it follows that for all $t \in T_{\Sigma+\Sigma 0}$ $\text{COM}_{\text{IMPL}}(t)$ is nonempty.

The length of computations is a good measure to tell about the complexity of an implementation. We therefore use it to define costs of terms:

3.2. DEFINITION (Cost Function). Given an implementation IMPL: SPEC1 $\Rightarrow$ SPEC0 and let length: $\text{COM}_{\text{IMPL}} \to \mathbb{N}_0$ assign to each computation its length. Then

$$\text{cost}_{\text{IMPL}}: T_{\Sigma+\Sigma 0} \to \mathbb{N}_0$$

is defined by

$$\text{cost}_{\text{IMPL}}(t) := \min\{\text{length}(w)/w \in \text{COM}_{\text{IMPL}}(t)\}$$

and called the *cost of* IMPL.

We use the cost function of an implementation to define complexity of $\Sigma 0$-operations with respect to a given implementation IMPL: SPEC1 $\Rightarrow$ SPEC0. The definition makes use of the diagram in fact 2.3(3) which shows the relation between data of T_{SPEC0} and their representation in REP_{IMPL}. The definition is meaningful only if IMPL is RI-correct.

3.3. DEFINITION (Complexity of Operations). Given an implementation IMPL: SPEC1 $\Rightarrow$ SPEC0 and let $\sigma \in \Sigma 0$, $\sigma: s1, \ldots, sn \to s$, denote the operation

$$\sigma_T: (T_{\text{SPEC0}})_{s1} \times \cdots \times (T_{\text{SPEC0}})_{sn} \to (T_{\text{SPEC0}})_s;$$

then the *complexity of σ with respect to* IMPL is given by the partial function

$$\text{comp}_{\text{IMPL}}(\sigma): (T_{\text{SPEC0}})_{s1} \times \cdots \times (T_{\text{SPEC0}})_{sn} \to \mathbb{N}_0$$

defined by

$$\mathrm{comp}_{\mathrm{IMPL}}(\sigma)(x_1,\dots,x_n) := \max\{f(r_1,\dots,r_n)/r_i \in (\mathrm{REP}_{\mathrm{IMPL}})_{si}, \mathrm{rep}(r_i)=x_i\}$$

with f defined by

$$f(r_1,\dots,r_n) := \min\{\mathrm{cost}_{\mathrm{IMPL}}(\sigma(t_1,\dots,t_n))/t_i \in (T_{\Sigma+\Sigma 0})_{si}, l(r_i)=r_i\},$$

where rep: $\mathrm{REP}_{\mathrm{IMPL}} \to T_{\mathrm{SPEC0}}$ and e: $T_{\Sigma+\Sigma 0} \to \mathrm{REP}_{\mathrm{IMPL}}$ are given in fact 2.3(2) and satisfy nat $=$ rep $\circ$ e.

3.4. Example. We discuss Definitions 3.2 and 3.3 in light of Example 2.4: Given a **set(nat1)**-term t of sort *set* and reduce it to a SORTIMPL terms t' which is normal with respect to the **string(nat1)**-operations. t' thus has TUP as its leading operation. Such a reduction is a computation in the sense of 3.1. In general, we can always find computations which are linear bounded in length by the number of operation symbols of their first term. We therefore conclude that $\mathrm{cost}_{\mathrm{IMPL}}$ is linear in the number of operation symbols. $\mathrm{comp}_{\mathrm{IMPL}}$(INSERT), the complexity of INSERT with respect to IMPL in 2.4, is estimated as follows: Let n and s denote arguments for INSERT, then $\mathrm{cost}_{\mathrm{IMPL}}(\mathrm{INSERT}(n, s))$ is linear in the number of operation symbols, as argued before. To obtain a value for f, the auxiliary function in the definition of $\mathrm{comp}_{\mathrm{IMPL}}$, we have to search for the "best" term representations of the $\mathrm{REP}_{\mathrm{IMPL}}$-data. Since data of sort *nat* are uniquely represented by terms involving the operations 0 and SUCC, it remains to consider the representations of $\mathrm{REP}_{\mathrm{IMPL}}$-data of sort *set*. Such representations are terms involving the operations CREATE, INSERT, DELETE or IF-THEN-ELSE. One can see that terms which include DELETE or IF-THEN-ELSE cannot be best. On the other hand if terms consisting of CREATE and INSERT have the same element to be inserted more than once, they are "redundant" and can be replaced by an "irredundant" term which represents the same $\mathrm{REP}_{\mathrm{IMPL}}$-datum optimally with respect to $\mathrm{cost}_{\mathrm{IMPL}}$.

The number of operation symbols for such "irredundant" terms now depends on the number of elements, which are inserted (in the set) and their value. We thus obtain that f is linear bounded in the sum of values of elements, which constitute the set.

In fact, this estimation is very rough and does not take into consideration the advantages of the hash table. To estimate $\mathrm{comp}_{\mathrm{IMPL}}$, we have to consider the different representations of SPEC0-data by $\mathrm{REP}_{\mathrm{IMPL}}$-data. Here $\mathrm{REP}_{\mathrm{IMPL}}$-data stand for tables with no multiple entries but can differ with respect to the order in which entries are inserted. So the following $\mathrm{REP}_{\mathrm{IMPL}}$-data d_1, d_2 differ, even though they represent the same set $\{3, 6\}$. Let $m = 3$, then

$$\begin{aligned} d_1 &= \mathrm{INSERT}(\mathrm{SUCC}^3(0), \mathrm{INSERT}(\mathrm{SUCC}^6(0), \mathrm{CREATE})) \\ &= \mathrm{TUP}(\mathrm{ADD}(\mathrm{ADD}(\lambda, \mathrm{SUCC}^6(0)), \mathrm{SUCC}^3(0)), \lambda, \lambda), \end{aligned}$$

$$\begin{aligned} d_2 &= \mathrm{INSERT}(\mathrm{SUCC}^6(0), \mathrm{INSERT}(\mathrm{SUCC}^3(0), \mathrm{CREATE})) \\ &= \mathrm{TUP}(\mathrm{ADD}(\mathrm{ADD}(\lambda, \mathrm{SUCC}^3(0)), \mathrm{SUCC}^6(0)), \lambda, \lambda). \end{aligned}$$

We have $\mathrm{rep}(d_1) = \mathrm{rep}(d_2) = \{3, 6\}$.

The worst $\mathrm{REP}_{\mathrm{IMPL}}$-datum d, which represents the set $\mathrm{rep}(d) = s$, is the one, where the element n, to be inserted, is rightmost in the string in the case where it was already inserted before. One can show that $\mathrm{comp}_{\mathrm{IMPL}}(\mathrm{INSERT})$ is defined for all arguments and is linear bounded by the sum of the values of the elements, which constitute the argument set.

Complexity analysis of hash tables is not just obvious, namely, if one is very accurate about it. This example might show that algebraic implementations and their complexity form a highly sensible frame for such analysis. By definition the complexity of operations is worst case complexity, which is expressed in the maximum, taken over all representations of SPEC0-data in $\mathrm{REP}_{\mathrm{IMPL}}$. The minimum in the definition of $\mathrm{comp}_{\mathrm{IMPL}}$ is meant to give independence of the "history" of data.

We have considered a very simple case of cost and complexity which does not reflect the specific nature of implementations of abstract data types: their relativity. Since an implementation implements an abstract data type $A0$ by an abstract data type $A1$, where $A1$ can have arbitrary complex operations, the notion of complexity of operations of the implementing data type $A1$ into consideration. On the other hand from a practical point of view it is desirable to have a more flexible notion which admits alternatives to the length of computations, or the choice of basic computation steps. In [14, 22] this is introduced and studied in an axiomatic framework. We are not going to discuss this extension of the concept here, but turns towards the following question:

How are algebraic implementations and their complexity related to Turing Machines and their time?

An answer to this question yields results about the expressive power of algebraic specifications (Section 4) and implementations (Section 5). It shows that Turing Machines can be simulated by algebraic implementations and that their complexity measures are compatible. Finally it shows that lower bounds exist for the complexity of implementations, which is one of the above requirements for complexity measures in general (see Corollary 6.4).

4. Specifications of Time Bounded Turing Machines and Total Recursive Functions

In order to relate Turing Machines and their time complexity to algebraic implementations and their complexity in Sections 5 and 6, we give algebraic specifications for (time bounded) Turing Machines and the functions they compute.

4.1. General Assumption. We assume we are given finite alphabets I and O, and a partial function $g: I^* \to O^*$ together with its totalization $f: I^* \to \bar{O}^*$, where I^* is the set of strings over I and $\bar{O}^* = O^* \cup \{\perp\}$, the set of strings over O with an element $\perp$ ("undefined") adjoined. By $\text{dom}(g) = \{w \in I^*/f(w) \in O^*\}$ we denote the domain of g.

We will show that a function $f: I^* \to \bar{O}^*$ has a correct algebraic specification (see Section 2 for "correct") if it is total recursive. The proof uses an algebraic specification of Turing Machines. Unfortunately we are not able to give specifications of arbitrary deterministic Turing Machines but have to assume that the machines have time bounds which have algebraic specifications. However, Theorem 4.3 below shows that this is not an essential restriction. We review an explicit definition of Turing Machines which will be used to show correctness of our specifications and implementations. For further details see [4] for example.

With a Turing Machine M computing a (partial) function $g: I^* \to O^*$ we associate a function

$$\text{time}_M: I^* \to \mathbb{N}_0$$

which assigns to $w \in \text{dom}(g)$ the number of steps which M performs to give $g(w)$. For technical reasons we assume

$$\text{time}_M(w) \geqslant \max(\text{length}(w), \text{length}(g(w))).$$

4.2. Definition (Algebraic Time Bound). A *Turing Machine M* is called *algebraic time bounded* if there is a total function Bound: $\mathbb{N}_0 \to \mathbb{N}_0$, called *algebraic time bound*, such that

1. there is a finite specification **time** which is an enrichment of **nat**, the specification of natural numbers, and is correct with respect to Bound;
2. for all $w \in \text{dom}(\text{time}_M)$

$$\text{Bound}(\text{length}(w)) \geqslant \text{time}_M(w).$$

Let g be a function and $f: I^* \to \bar{O}^*$ as given in 4.1 then f is called *algebraic time bounded* if there is an algebraic time bounded Turing Machine for g.

4.3. Theorem. *A function $f: I^* \to \bar{O}^*$ is algebraic time bounded if and only if it is total recursive.*

Proof. If f is algebraic time bounded then g is partial recursive with domain $\text{dom}(g) = (w \in I^*/f(w) \neq \perp\}$. Since there is an algebraic time bound, $\text{dom}(g)$ is decidable and thus f is total recursive. Decidability of $\text{dom}(g)$ is seen as follows: Let BOUND denote the operation which names the function Bound in the specification **time** (see 4.2). Then given a (nondeterministic) interpreter I (as in 6.2) such that I transforms each term BOUND(LENGTH(w)) to the term $\text{SUCC}^{\text{Bound}(\text{length}(w))}(0)$. Such an interpreter I exists by the enrichment property of **time**. Let $T(I)$ be a Turing

Machine which simulates I (reading terms as words), then we build a Turing Machine M which decides dom(f) as follows: First let $T(I)$ run with input term BOUND (LENGTH(W)). The term $\mathrm{SUCC}^{\mathrm{Bound(length}(w))}(0)$ results. Then let the machine TM which computes g run in parallel with a machine COUNT which reduces $\mathrm{SUCC}^i(0)$ to $\mathrm{SUCC}^{i-1}(0)$ with each step of TM. If $i=0$, then TM will not come to a halting state and thus $w \notin \mathrm{dom}(g)$. Otherwise TM will halt before $i=0$, leading to $w \in \mathrm{dom}(g)$.

Suppose f is total recursive. Thus it is computable by a Turing Machine TM with total $\mathrm{time}_{TM}: I^* \to \mathbb{N}_0$. Then also the function Bound: $\mathbb{N}_0 \to \mathbb{N}_0$, defined by

$$\mathrm{Bound}(n) = \max\{\mathrm{time}_{TM}(w)/\mathrm{length}(w) = n,\ w \in I^*\}$$

is total recursive. Bound together with the 0-ary operation 0 and the unary successor it constitutes a computable algebra in the sense of [6], where in the main theorem it is shown that exactly the computable algebras have finite algebraic enrichment specifications of **nat**. Thus f is algebraic time bounded. ∎

To specify Turing Machines means that we have to look at Turing Machines as a many sorted algebra and specify this algebra in terms of sorts, operations and equations:

4.4. Definition (Time Bounded Turing Machines). A deterministic on tape Turing Machine TM is given by a tuple $TM = (A, S, M, \mathrm{Time})$ consisting of the following items:

Time: $\mathbb{N}_0 \to \mathbb{N}_0$	a total function, called *time bound*;
$A = \{a_1, \ldots, a_n\}$	*tape alphabet*, containing a set I of *input symbols*, a set O of *output symbols* and distinctive elements b and $*$ called *blank* and *star*;
$S = \{s_1, \ldots, s_r\}$	*sets of states* (disjoint from A), containing distinctive elements B and H called *beginning* and *halting states*, respectively;

$$M: A \times S \times A \times A \to (S \times A \times A \times A \cup A \times S \times A \times A \cup A \times A \times S \times A)$$

a total function called *machine table.*

From these data we define the set of *configurations* C and *step function* step: $C \to C$ as follows:

$$C := A^*ASAAA^* \cup \{\text{time-limit}\};$$

for $n, v \in A^*$, $x, y, z \in A$ and $s \in S$ step is defined by

$$\mathrm{step}(nxsyzv) := u'abcdv' \qquad \text{if} \quad M(x, s, y, z) = (a, b, c, d)$$

with $u' = *$ if $u = \lambda$ and $u' := u$ otherwise and also $v' := *$ if $v = \lambda$ and $v' := v$ otherwise; for time-limit step is defined step(time-limit) := time-limit.

TM computes a total function $f: I^* \to \bar{O}^*$ which is defined as follows: Let $\text{Sit} := \mathbb{N}_0 \times C$ be called the set of *situations* and $\text{run}: \text{Sit} \to \text{Sit}$ be called the *run function* of *TM* be defined by

$$
\begin{aligned}
\text{run}(j, C) &:= (j, 0) && \text{if the state component of } C \text{ is } H \text{, the halting state}\\
&= (0, \text{time-limit}) && \text{if } j = 0\\
&= \text{run}(j-1, \text{step}(C)) && \text{otherwise.}
\end{aligned}
$$

The *result function* $f: I^* \to \bar{O}^*$ then is defined by

$$
\begin{aligned}
f(w) &:= v && \text{if run(time(length}(w)), *B\cancel{b}w*) = (j, *H\cancel{b}v*) \text{ for some } j \in \mathbb{N}_0\\
&= \perp && \text{otherwise.}
\end{aligned}
$$

4.5. Specification of Time Bounded Turing Machines. Given a time bounded Turing Machine as in 4.4.

1. **machine domains** =

sorts: *symb, states, strings, quadrup, config*

opns: $a :\to symb$ (for $a \in A$)
$s :\to states$ (for $s \in S$)
$\lambda\text{STR} :\to strings$
ADDL: *symb strings* $\to$ *strings*
ADDR: *strings symb* $\to$ *strings*
QUAD: *symb states symb symb* $\to$ *quadrup*
CONF: *strings quadrup strings* $\to$ *config*
TIMELIMIT $:\to$ *config*

eqns: (1) $\text{ADDR}(\lambda\text{STR}, a) = \text{ADDL}(a, \lambda\text{STR})$
(2) $\text{ADDR}(\text{ADDL}(a, S), b) = \text{ADDL}(a, \text{ADDR}(S, b))$

2. **machine = machine domains** +

opns: STEP: *config* $\to$ *config*

eqns: (3) STEP(TIMELIMIT) = TIMELIMIT
(4) $\text{STEP}(\text{CONF}(\lambda\text{STR}, \text{QUAD}(x, s, y, z), D))$
$= \text{CONF}(\lambda\text{STR}, \text{QUAD}(*, s', x', y'), \text{ADDL}(z', D))$
$\text{STEP}(\text{CONF}(\text{ADDR}(c, w), \text{QUAD}(x, s, y, z), D))$
$= \text{CONF}(c, \text{QUAD}(w, s', x', y'), \text{ADDL}(z', D))$
(for all (x, s, y, z) such that $M(x, s, y, z) = (s', x', y', z')$)
(5) $\text{STEP}(\text{CONF}(c, \text{QUAD}(x, s, y, z), D))$
$= \text{CONF}(c, \text{QUAD}(x', s', y', z'), D)$
(for all (x, y, z) such that $M(x, s, y, z) = (x', s', y', z')$)
(6) $\text{STEP}(\text{CONF}(c, \text{QUAD}(x, s, y, z), \lambda\text{STR}))$
$= \text{CONF}(\text{ADDR}(c, x'), \text{QUAD}(y', s', z', *), \lambda\text{STR})$
$\text{STEP}(\text{CONF}(c, \text{QUAD}(x, s, y, z), \text{ADDL}(w, D)))$
$= \text{CONF}(\text{ADDR}(c, x'), \text{QUAD}(y', s', z', w'), D)$
(for all (x, s, y, z) such that $M(x, s, y, z) = (x', y', s', z')$)

The machine table is not explicitly specified but the step function which is just an extension of the machine table to configurations is.

3. **function domains** =
   sorts: *insymb, inputs, outsymp, outputs*
   opns: $a{:}\rightarrow$ *insymb* (for $a \in I$)
   λIN: $\rightarrow$ *inputs*
   BUILDIN: *insymb inputs* $\rightarrow$ *inputs*
   $b{:}\rightarrow$ *outymb* (for $b \in 0$)
   UNDEFINED: $\rightarrow$ *outputs*
   λOUT: $\rightarrow$ *outputs*
   BUILDOUT: *outsymb outputs* $\rightarrow$ *outputs*
   eqns: BUILDOUT(b, UNDEFINED) = UNDEFINED (for $b \in 0$)

4. **system domains = nat + bool + function domains +**
   sorts: *sit*
   opns: SIT: *nat config* $\rightarrow$ *sit*
   (note, this is no specication in itself since it uses *config*)
   nat and **bool** are specifications for natural numbers and Boolean values.

5. **time = nat + bound** is assumed to be given such that **time** is enrichment of **nat** and **bound** contains an operation TIME which names the function TIME in the time bounded Turing Machine *TM*. **time** is assumed to be correct with respect to the time bound Time. **bound** is no specification in itself.

6. **turing system = machine + system domains + bound +**
   opns: FUNCTION: *inputs* $\rightarrow$ *outputs*
   RESULT: *sit* $\rightarrow$ *outputs*
   RUN: *sit* $\rightarrow$ *sit*
   START: *inputs* $\rightarrow$ *sit*
   STATE: *sit* $\rightarrow$ *states*
   EQS: *states states* $\rightarrow$ *states*
   IF-THEN-ELSE-SIT: *bool sit sit* $\rightarrow$ *sit*
   LENGTH: *inputs* $\rightarrow$ *nat*
   CODEIN: *inputs* $\rightarrow$ *strings*
   COMPOSE: *outputs outputs* $\rightarrow$ *outputs*
   CODEOUT: *strings* $\rightarrow$ *outputs*
   IF-THEN-ELSE-OUT: *bool outputs outputs* $\rightarrow$ *outputs*
   eqns: (7) FUNCTION(A) = RESULT(START(A))
   (8) RESULT(SIT(n, OVERFLOW)) = UNDEFINED
   RESULT(SIT(n, CONF(c, QUAD(x, s, y, z), D)))
   = IF EQS(STATE(SIT(n,CONF(c,QUAD(x,s,y,z),D))),H)
   THEN COMPOSE (CODEOUT(ADDR(c, x)),
   CODEOUT(ADDL(y, ADDL(z, D)))) ELSE
   RESULT(RUN(SIT(n, CONF(c, QUAD(x, s, y, z), D))))

(9) RUN(SIT(n, c)) = IF EQS(STATE(SIT(n, c)), H)
THEN SIT(n, c)
ELSE IF EQ(n, 0) THEN SN (0, OVERFLOW)
ELSE RUN(SIT(PRED(n), STEP(c)))

(10) START(λIN) = SIT(TIME(0), CONF(λSTR, QUAD($*, B, b, *$), λSTR))
START(BUILDIN(a, A)) = SIT(TIME(SUCC(LENGTH(A)))), CONF(λSTR, QUAD($*, B, b, a$), ADDR(CODEIN(A), $*$))
(for all $a \in I$)

(11) STATE(SIT(n, OVERFLOW)) = H (the halting state)
STATE(SIT(n, CONF(c, QUAD(x, s, y, z), D))) = s
(for all $s \in S$)

(12) EQS(s, t) = F (for all s, $t \in S$ with $s \neq t$)
EQS(s, s) = T (for all $s \in S$)

(13) IF T THEN S ELSE S' SIT = S
IF F THEN S ELSE S' SIT = S'

(14) LENGTH(λIN) = 0
LENGTH(BUILDIN(a, A)) = SUCC(LENGTH(A))
(for all $a \in I$)

(15) CODEIN(λIN) = λSTR
CODEIN(BUILDIN(a, A)) = ADDL($\bar{a}$, CODEIN(A))
(for all $a \in I$; $\bar{a}$ is the corresponding input element in A)

(16) COMPOSE(λOUT, A) = A
COMPOSE(UNDEFINED, A) = UNDEFINED
COMPOSE(BUILDOUT(a, A), B) = BUILDOUT(a, COMPOSE(A,

(17) CODEOUT(λSTR) = λOUT
CODEOUT(ADDL($\bar{a}, A$)) = CODEOUT(A) (for $\bar{a} \in A - 0$)
CODEOUT(ADDL($\bar{a}, A$)) = BUILDOUT(a, CODEOUT(A))
(for all $a \in I$; $\bar{a}$ is the corresponding input element in A)

(18) If T THEN A ELSE A' OUT = A
IF F THEN A ELSE A' OUT = A'

This specification follows the definition of f in 4.4. Here FUNCTION names f, RUN NAMES run and the other operation are auxiliary with the following intention:

RESULT	produces the output of the *TM* which is uniquely determined in each of its situations
START	produces from an input the starting situation of the *TM*
STATE	projects the state component of the situation
EQS	is equality predicate on the states
CODEIN	translates the input string into a string on the tape
CODEOUT	translates the string on the tape into an output string

4.6. Theorem (Correctness). *The specification* **turing system**, *given in* 4.5, *is a correct specification of time bounded Turing Machines and their result functions as well as their domains, as given in* 4.4. *Moreover* **turing system** *is an enrichment of the combination* **domains = machine domains + system domains.**

Proof. See Theorem 10.3 of [14]. ▮

4.7. Corollary (Algebraic Specification of Total Recursive Functions). *If $f: I^* \to \bar{O}^*$ is total recursive, then there is an algebraic specification for f.*

Proof. From Theorem 4.3 we conclude that f is algebraic time bounded, so that there is an algebraic time bounded Turing Machine for f. Using 4.6 we derive the existence of an algebraic specification. ▮

5. Solvability and Upper Bound on the Function Implementation Problem

In this section we give an upper bound on the complexity of algebraic implementations which implement algebraic time bounded functions. Since by the definition of correctness in Section 2 it is clear what is meant by an algebraic specification of a functions f (f is considered as a many sorted algebra with f as the only operation) it is not clear yet what it means to implement a function. The following definition makes it precise:

5.1. Definition (Function Implementation Problem). Let $f: I^* \to \bar{O}^*$ be a function, as in 4.1; then the *function implementation problem for f* is to find extensions SPEC1 = **functions domains** + $\langle S1, \Sigma 1, E1 \rangle$ and SPEC0 = **function domains** + $\langle S0, \Sigma 0, E0 \rangle$ of **function domains** and an implementation IMPL: SPEC1 $\Rightarrow$ SPEC0 such that

1. there is an operation F: *inputs* $\to$ *outputs* in $\Sigma 0$ (but not in $\Sigma 1$) such that

$$\begin{tikzcd}
I^* \arrow[r, "f"] & \bar{O}^* \\
(T_{SPEC0})_{\underline{inputs}} \arrow[u, "b_I"] \arrow[r, "F_T"'] \arrow[ur, phantom, "="] & (T_{SPEC0})_{\underline{outputs}} \arrow[u, "b_O"']
\end{tikzcd}$$

commutes with the semantics F_T of F and the canonical bijections b_I and b_O, existing from the correctness of **function domains** with respect to I^* and $\bar{O}^*$ (see 4.5(3) and 4.6).

2. All operations of SPEC0 with range *inputs* are already in **function domains.**

3. All operations of SORTIMPL with range *outputs* are already in **function domains.**

A *solution* of the function implementation problem is formally denoted by (IMPL, F).

While the first condition in 5.1 is to find a correct specification for f, the second and third conditions assure that the implementation is "honest," i.e., has computations which do not leave the work to the codings (see Lemma 6.1 below).

5.2. THEOREM (Solvability). *If $f: I^* \to \bar{O}^*$ is total recursive, then the function implementation problem for f has a solution.*

Proof. If f is total recursive, it is algebraic time bounded by Theorem 4.3 and thus has an algebraic specification by 4.7 of the kind given in 4.5. We use 2.3(4) and 4.6 and define

SPEC0 = **turing machine,**

SPEC = **domains,**

IMPL: SPEC ⇒ SPEC0 by $(\varnothing, E0, \langle \varnothing, \varnothing, \varnothing \rangle)$.

Then SPEC0 and SPEC are both extensions of **function domains** and SPEC0 has in FUNCTION the desired operation. Since SPEC0 has no operations other than λIN and BUILDIN with range *inputs*, also property (2) in 5.1 is satisfied. Since IMPL is an enrichment implementation, ΣSORT $= \varnothing$ and this we need for (3) that all operations in **domains** of range *outputs* are already in **function domains**, which is true.

Thus (IMPL, FUNCTION) is a solution of the function implementation problem for f. ∎

In the next section we will see that also the converse of 5.2 is true.

5.3. THEOREM (Upper Bound). *If $f: I^* \to \bar{O}^*$ is computable by a Turing Machine TM with algebraic time bound* Time, *then there is a solution* (IMPL, F) *of the function implementation problem for f such that*

$$\begin{aligned} \mathrm{comp}_{\mathrm{IMPL}}(F)(x) &\leqslant C \cdot (\mathrm{time}_{TM}(b_I(x)) + \mathrm{red}(x)) && \text{if } b_I(x) \in \mathrm{dom}(f) \\ &\leqslant C \cdot (\mathrm{Time}(\mathrm{length}(b_I(x)) + \mathrm{red}(x))) && \text{otherwise,} \end{aligned}$$

where C is some constant, red(x) *is the number of direct reduction steps to reduce the term* TIME$(\mathrm{SUCC}^{\mathrm{length}(b_I(x))}(0))$ *to the term* $\mathrm{SUCC}^{\mathrm{Time}(\mathrm{length}(b_I(x)))}(0)$, *and b_I is given in* 5.1.

Proof. We analyze the complexity of the enrichment for f, as given in the proof of 5.2, and thus go back to the specification of Turing Machines in 4.5.

Given for $t \in (T_{\Sigma 0})_{inputs}$ an arbitrary computation $c(t)$ of length $h(t)$ (i.e., number of direct reduction steps) such that $c(t)$ reduces the term FUNCTION(t) to some term $t' \in (T_{\Sigma + \Sigma 1 + \Sigma \mathrm{SORT}})_{outputs}$, then we have for some $a > 0$

$$\mathrm{cost}_{\mathrm{IMPL}}(\mathrm{FUNCTION})(x) \leqslant a \cdot h(t) \qquad \text{with} \quad [t] = x$$

and, since rep in 2.3(3) is identity in this case, also

$$\mathrm{comp}_{\mathrm{IMPL}}(\mathrm{FUNCTION})(x) \leqslant a \cdot h(t). \tag{$*$}$$

Thus it remains to determine form some computation $c(t)$ its length $h(t)$. This is done subsequently leading to

$$\begin{aligned} h(t) &\leqslant 9 \cdot \mathrm{time}_{TM}(b_I(x)) + \mathrm{red}(x) + 15 && \text{if } b_I(x) \in \mathrm{dom}(f) \\ &\leqslant 9 \cdot \mathrm{Time}(\mathrm{length}(b_I(x))) + \mathrm{red}(x) + 9 && \text{otherwise.} \end{aligned} \tag{$**$}$$

The estimation $(**)$ then together with $(*)$ proves the theorem for an appropriate constant C.

To derive $(**)$ we reduce the term FUNCTION(t) to some $(\Sigma + \Sigma 1 + \Sigma\mathrm{SORT})$-term and count the number of direct reduction steps. Three cases are distinguished:

First, $b_I(x) \in \mathrm{dom}(f)$ and $\mathrm{length}(b_I(x)) \geqslant 1$.

(1) FUNCTION(t).

Applying eqn (7) in 4.5

(2) RESULT(START(t)).

Applying eqn (10) using $t = \mathrm{BUILDIN}(a, Y)$

(3) RESULT(SIT(TIME(SUCC(LENGTH(A))), CONF(λSTR, QUAD($*, B, b, a$), ADDR(CODEIN(A), $*$)).

Using $A = a_1 \cdots a_r$ we obtain in length($b_I(x)$) steps of eqns (14), (15)

(4) RESULT(SIT(TIME(SUCC$^{\mathrm{length}(b_I(x))}$(0)), CONF($\lambda$STR, QUAD($*, B, b, a$), ADDR(ADDL($a_1$, ADDL(..., ADDL($a_r$, STR)...), $*$))).

Using eqns (1), (2) we obtain in length($b_I(x)$) steps

(5) RESULT(SIT(TIME(SUCC$^{\mathrm{length}(b_I(x))}$(0)), CONF)$\lambda$STR, QUAD($*, B, b, a$), ADDL($a_1$,..., ADDL($*$, λSTR))...).

In red(x) steps we obtain by assumption

(6) RESULT(SIT(SUCC$^{\mathrm{Time}(\mathrm{length}(b_I(x)))}$(0), CONF($\lambda$STR, QUAD($*, B, b, a$), ADDL($a_1$,..., ADDL($*$, λSTR))...).

Using eqns (8), (11), (13), (12) we obtain in four steps.

(7) RESULT(RUN(SIT(SUCC$^{\mathrm{TIME}(\mathrm{length}(b_I(x)))}$(0), CONF($\lambda$STR, QUAD($*, B, b, a$), ADDL($a_1$,..., ADDL($*$, λSTR))...).

Repeated application of eqns (9), (11), (13), (12) and eqns (4), (5), (6) results in $5 \cdot \mathrm{time}_{TM}(b_I(x))$ steps.

(8) RESULT(SIT(SUCC^m(0), CONF(λSTR, QUAD($*, H, \not b, b_1$), ADDL(b_2,..., ADDL($*$, λSTR))...),

where $m :=$ Time(length($b_I(x)$)) $-$ time$_{TM}$($b_I(x)$) and $f(b_I(x)) = b_1 \cdots b_r$ is assumed. Using eqns (8), (11) and (13) we obtain in three steps

(9) COMPOSE(CODEOUT(ADDR(λSTR, $*$)), CODEOUT(ADDL($\not b$, ADDL(b_1,..., ADDL($*$, λSTR))...).

Using eqns (1) and two times (17) we obtain in three steps

(10) COMPOSE(λOUT, CODEOUT(ADDL($\not b$,..., ADDL($*$, λSTR)...))).

Using eqn (16) and $(r + 3)$ times eqn (17) we obtain

(11) BUILDOUT(b_1, BUILDOUT(b_2,..., BUILDOUT(b_r, λOUT)...)).

We have now constructed a computation with first term given in (1) and lest term given in (11) which is of length

$$3 \cdot \text{length}(b_I(x)) + \text{red}(x) + 5 \cdot \text{time}_{TM}(b_I(x)) + \text{length}(f(b_I(x))) + 15$$

which is less than

$$9 \cdot \text{time}_{TM}(b_I(x)) + \text{red}(x) + 15$$

using the assumption time$_{TM}$($b_I(x)$) $\geqslant$ max(length($b_I(x)$), length($f(b_I(x))$)).

Second, $b_I(x) \notin \text{dom}(f)$ and length($b_I(x)$) $\geqslant 1$.

In this case we obtain terms (1) to (7) as before but after $5 \cdot$ Time (length($b_I(x)$)) steps we obtain the term

(12) RESULT(SIT(0, TIMELIMIT))

and in final step

(13) UNDEFINED.

The computation has length

$$3 \cdot \text{length}(b_I(x)) + \text{red}(x) + 5 \cdot \text{Time}(\text{length}(b_I(x))) + 3$$

which is less than

$$8 \cdot \text{Time}(\text{length}(b_I(x))) + \text{red}(x) + 3$$

using the assumption Time($b_I(x)$) $\geqslant$ length($b_I(x)$).

Third, length($b_I(x)$).

In this case we obtain in nine steps the term UNDEFINED.

These three cases show($**$) ∎

Remark. It would be desirable to have an upper bound which only depends on the Turing Machine *TM* and not on the algebraic time bound in addition. But such a bound is not yet known; it probably would imply that we can specify (or implement) a Turing Machine and the function which it computes without use of an algebraic time bound.

6. Lower Bound on the Function Implementation Problem

In this section we give a lower bound on the complexity of algebraic implementations which solve function implementation problems. The proof of this bound also shows that functions with a solvable function implementation problem are total recursive, which is the converse of Theorem 5.2.

We are dealing with nondeterministic Turing Machines and functions in this section and therefore introduce the following conventions:

A nondeterministic function $h: A \to B$ is a left total relation and gives rise to define a function $\bar{h}: A \to 2^B$ by $b \in \bar{h}(a)$ if and only if $(a, b) \in h$.

A nondeterministic Turing Machine is a Turing Machine where the machine table M is a nondeterministic function. We say that a nondeterministic Turing Machine NTM computes a nondeterministic function $h: A \to B$ iff for all inputs $a \in A$ NTM results in $b \in B$ with $b \in \bar{h}(a)$, if it stops, and has for each $b \in \bar{h}(a)$ at least one successful run.

Time complexity of nondeterministic Turing Machines NTM is a partial function $n\text{time}_{NTM}; A \to \mathbb{N}_0$ which assigns to each $a \in A$ the length of the shortest successful run of NTM.

6.1. Lemma (Realization). *Let $f: I^* \to \bar{O}^*$ be a function, as in* 4.1, *and* (IMPL, F) *a solution of the function implementation problem for f. Then there is a nondeterministic function*

$$\mathrm{REAL}_{\mathrm{IMPL}}(f): (T_{\Sigma+\Sigma\mathrm{O}})_{inputs} \to (T_{\Sigma+\Sigma 1+\Sigma\mathrm{SORT}})_{outputs}$$

called realization of f by IMPL, *such that the following holds*:

1. *For all $t \in (T_{\Sigma+\Sigma\mathrm{O}})_{inputs}$ and $t' \in (T_{\Sigma+\Sigma 1+\Sigma\mathrm{SORT}})_{outputs}$ we have $(t, t') \in \mathrm{REAL}_{\mathrm{IMPL}}(f)$ is and only if there is a computation $w \in \mathrm{COM}_{\mathrm{IMPL}}$ with first term $F(t)$ and last term t'.*

2. *There are surjective functions c_I and c_O, called* codings of I^* and $\bar{O}^*$, *such that the following diagram commutes*:

$$\begin{tikzcd}
(T_{\Sigma+\Sigma\mathrm{O}})_{\underline{inputs}} \arrow[rr, "\mathrm{REAL}_{\mathrm{IMPL}}(f)"] \arrow[d, "c_I"] \arrow[drr, phantom, "="] & & (T_{\Sigma+\Sigma 1+\Sigma\mathrm{SORT}})_{\underline{outputs}} \arrow[d, "c_O"'] \\
I^* \arrow[rr, "f"'] & & \bar{O}^*
\end{tikzcd}$$

3. *c_O is computable by a deterministic Turing Machine in polynomial time.*

4. *c_I is bijective and the inverse function $c_I^{-1} \cdot I^* \to (T_{\Sigma+\Sigma_0})_{inputs}$ is computable by a deterministic Turing Machine in polynomial time.*

Proof. (1) Since IMPL is assumed to be correct and thus OP-complete, $\mathrm{REAL}_{\mathrm{IMPL}}(f)$ becomes a nondeterministic function as desired if it is defined by 1.

(2) Define functions $d_I :=$ nat-in and $d_0 := g \circ h \circ$ nat-out, where

$$\text{nat-in}: (T_{\Sigma+\Sigma 0})_{inputs} \to (T_{\mathrm{SPEC0}})_{inputs},$$
$$\text{nat-out}: (T_{\Sigma+\Sigma 1+\Sigma\mathrm{SORT}})_{outputs} \to (T_{\mathrm{SORTIMPL}})_{outputs}$$

are the natural homomorphisms and

$$h: (T_{\mathrm{SORTIMPL}})_{outputs} \to (T_{\mathrm{SPEC}})_{outputs},$$
$$g: (T_{\mathrm{SPEC}})_{outputs} \to (T_{\mathrm{SPEC0}})_{outputs}$$

are the canonical isomorphisms where h exists by type protection (see 2.3(1)) and the general assumption 2.1 and g by the extension property of SPEC0 (see 2.1). Let b_I and b_O be given as in 5.1(1); then we define $c_I := b_I \circ d_I$ and $c_O := b_O \circ d_O$. Property (2) then follows from $b_O \circ F_T = f \circ b_I$ (see 5.1) and $d_0 \circ \mathrm{REAL}_{\mathrm{IMPL}}(f) = F_T \circ d_I$, where the latter is seen as follows:

Let $t \in (T_{\Sigma+\Sigma 0})_{inputs}$ and $t1 \in (T_{\Sigma+\Sigma 1+\Sigma\mathrm{SORT}})_{outputs}$ with $(t, t1) \in \mathrm{REAL}_{\mathrm{IMPL}}(f)$. Then from (1) and the fact before 3.1 it follows that $t1 \equiv_{\mathrm{OPIMPL}} F(t)$. With $d_O(t1) = [t2]$ we have $t1 \equiv_{\mathrm{SORTIMPL}} t2$ and hence $F(t) \equiv_{\mathrm{OPIMPL}} t2$. RI-correctness of IMPL then implies $F(T) \equiv_{\mathrm{SPEC0}} t2$, which means $[t2] = [F(t)] = F_T(d_I(t))$.

(3) Since by property 5.1(3) all terms in SORTIMPL of sort *outputs* are already **function domain**-term, c_O is obviously computable by a deterministic Turing Machine in polynomial time (reading terms as words).

(4) In fact since all terms in SPEC0 or sort *inputs* are already **function domain**-terms by 5.1(2) and the domain $(T_{\textbf{function domains}})_{inputs}$ is freely generated, nat-in in the proof of (2) above is bijective and thus c_I is a bijection which is nothing but a renaming and thus computable in polynomial time. ∎

The realization of a function f by an implementation IMPL can be looked at as the effect of a nondeterministic interpreter for an algebraic implementation and is therefore of interest in its own right. The following theorem which is also a preparation for our lower bound result, may thus be interpreted as a statement about a nondeterministic interpreter for implementations:

6.2. THEOREM (Nondeterministic Interpreter). *Let $f: I^* \to \bar{O}^*$ as in* 4.1 *and* (IMPL, F) *be a solution of the function implementation problem for f. Then there is a* *nondeterministic Turing Machine INT which computes the realisation of f by* IMPL *such that*

$$n\text{time}_{INT}(x) \leqslant d^{\text{cost}_{\text{IMPL}}(F(x))}$$

for $x \in (T_{\Sigma+\Sigma 0})_{inputs}$ *and some* $d > 1$.

Proof. The Turing Machine INT is combined from a number of smaller machines. Let $\text{EQ}_{\text{IMPL}} := \{(t, t') \mid (t = t') \in E(\text{OPIMPL}) \text{ or } (t' = t) \in E(\text{OPIMPL})\}$ define the set of "directed" OPIMPL-equations. Let there be for $e \in \text{EQ}_{\text{IMPL}}$ a function defined by

$$S(e)\colon T_{\Sigma(\text{OPIMPL})} \to 2^{T_{\Sigma(\text{OPIMPL})}}$$

with $S(e)(t) := \{t' \mid (t, e, t') \in \text{Dired}_{\text{OPIMPL}}\}$.

Then we denote by $M(e)$ *a nondeterministic Turing Machine* which realizes $S(e)$ nondeterministically as follows:

$M(e)$ takes all terms $t \in T_{\Sigma(\text{OPIMPL})}$, represented as an element of $\Sigma(\text{OPIMPL})^*$, as inputs and scans t in order to choose a left occurrence for e, if one exists, and then replaces the left occurrence by the right hand side of e resulting in some element of $S(e)(t)$. If no handle exists, it results in t.

It is well known that such a Turing Machine $M(e)$ can run in polynomial time, depending on the number of symbols of the input term, using traversing and replacement in trees.

Next we denote by $M(\text{test})$ *a* (*deterministic*) *Turing Machine* which works as follows:

$M(\text{test})$ takes all terms $t \in T_{\Sigma(\text{OPIMPL})}$ as inputs and tests if t is a $(\Sigma + \Sigma 1 + \Sigma\text{SORT})$-term in which case it results YES. It results NO otherwise.

Also $M(\text{test})$ can run in polynomial time, depending on the number of symbols of the input term.

INT then is combined from $M(e)$ for all $e \in \text{EQ}_{\text{IMPL}}$ and $M(\text{test})$ as follows:

INT takes all terms $t \in T_{\Sigma+\Sigma 0}$ as inputs and reduces t nondeterministically to a term $t' \in \text{REAL}_{\text{IMPL}}(f)(t)$. *INT* realizes the following procedure:

```
(1) TERM := "input term"
(2) if M(test) applied to TERM results YES
        then result TERM and stop!
        else do (3)
(3) choose e ∈ EQ_IMPL
    apply M(e) to TERM and obtain a result NEWTERM
    define TERM:= NEWTERM
    do (2).
```

INT now is a nondeterministic Turing Machine which computes the realization of f

by IMPL. So it is left to analyse *INT*. For this purpose we assign to each computation $w \in \mathrm{COM}_{\mathrm{IMPL}}$ a run of *INT* which is denoted by $R(w)$ and defined as follows:

By definition computation w is a sequence of direct reduction steps $(t_i, e_i, t_{i+1})_{i=1,\ldots,n}$, where $t_i \in T_{\Sigma(\mathrm{OPIMPL})}$, $e_i \in \mathrm{EQ}_{\mathrm{IMPL}}$ and $n = \mathrm{length}(w)$. By construction each direct reduction step uniquely (!) determines a sequence of steps of *INT* which corresponds to a run of M(test) with input t_i and a run of $M(e_i)$ with input t_i and output t_{i+1}. Let us call such a sequence of steps of *INT* a *superstep* for the moment; then we define $R(w)$ to be the run of *INT* which consists of the n supersteps defined by w.

Now, before we relate the length of w and of $R(w)$, respectively, we need to prove the following:

FACT. Let $w = (t_i, e_i, e_{t+1})_{i=1,\ldots,n}$; then there is $b \in \mathbb{N}$ such that for all $i = 1,\ldots, n$

$$\mathrm{size}(t_i) \leqslant b^n,$$

where size(t) denotes the number of symbols of t.

Proof. For $e \in \mathrm{EQ}_{\mathrm{IMPL}}$ we define $z(e) := zr(e)/zl(e)$, where $zl(e)$ denotes the number of (not necessarily distinct) variables of the left hand side of e and so $zr(e)$ analogously for the right hand side. Let z denote the maximum of these fractions, taken over all e_i apppearing in w. Then we have (using tree replacement)

$$\mathrm{size}(t_{i+1}) \leqslant z \cdot \mathrm{size}(t_i) + c,$$

where c is some constant depending on i. From this inequality we derive the existence of some $b \geqslant z$ such that

$$\mathrm{size}(t_i) \leqslant b^n$$

for all $i = 1,\ldots, n$. This proves the fact.

Now let p_e and p_{test} denote polynoms which bound the runnning time of $M(e)$ and M(test), respectively.

Then we derive for all computations w:

$$\begin{aligned}\mathrm{length}(R(w)) &\leqslant \sum_{i=1}^{n+1} p_{\mathrm{test}}(\mathrm{size}(t_i)) + \sum_{i=1}^{n} p_{e_i}(\mathrm{size}(t_i))\\ &\leqslant \sum_{i=1}^{n+1} p_{\mathrm{test}}(b^n) + \sum_{i=1}^{n} p_{e_i}(b^n)\\ &\leqslant c^n = c^{\mathrm{length}(w)} \qquad \text{for some constant } c > 1. \qquad (*)\end{aligned}$$

Since $n\text{time}_{INT}(x) \leqslant \text{length}(R(w))$ for all computations w, this must hold also for an optimal computation w_0, where $\text{length}(w_0) = \text{cost}_{\text{IMPL}}(F(x))$. Thus we obtain

$$n\text{time}_{\text{INT}}(x) \leqslant c^{\text{cost}_{\text{IMPL}}(F(x))}. \quad \blacksquare$$

The lower bound result now is as follows:

6.3. THEOREM (Lower Bound). *Let* $f\colon I^* \to \bar{O}^*$ *be as in* 4.1 *and* (IMPL, F) *a solution of the function implementation problem for f. Then*

1. *f is total recursive*;

2. *there is a nondeterministic Turing Machine NTM which computes the (deterministic) function f such that for all* $x \in I^*$

$$n\text{time}_{NTM}(x) \leqslant d^{\text{comp}_{\text{IMPL}}(F)(b_I^{-1}(x))}$$

for some constant $d > 1$ *and* b_I *given in* 5.1(1).

Proof. We construct a nondeterministic Turing Machine *NTM* as follows (see diagram in the realization Lemma 6.1(2)):

Let $M(I)$ denote a deterministic Turing Machine which computes c_I^{-1}. Such a machine exists by 6.1(4). Let $M(0)$ be a deterministic Turing Machine which computes c_O (see 6.1(3)). Furthermore let *INT* be the nondeterministic machine as constructed in the proof of 6.2 which computes $\text{REAL}_{\text{IMPL}}(f)$. Then the combination

$$NTM := M(0) \circ INT \circ M(I)$$

of these three machines computes f which follows from the commutativity of the diagram in 6.1(2). This proves (1) of 6.3. In order to prove the proposed estimation we construct a run r of *NTM* such that

$$\text{length}(r) \leqslant d^{\text{comp}_{\text{IMPL}}(F)(b_I^{-1}(x))}.$$

The run r consists of three components:

First: Let $r_{M(I)}$ be a run of $M(I)$ with input x resulting in a term $t \in (T_{\Sigma+\Sigma 0})_{inputs}$. Since by 5.1(2) $(T_{\Sigma+\Sigma 0})_{inputs} \cong (T_{\text{SPEC}0})_{inputs}$ we have

$$\text{comp}_{\text{impl}}(F)([t]) = \text{cost}_{\text{IMPL}}(F(t)) \tag{1}$$

which follows directly from the definition of $\text{comp}_{\text{IMPL}}$ in 3.3.

The run $r_{M(I)}$ is bounded in length by a polynom p_I (see 6.1(4)) so that we have

$$\text{length}(r_{M(I)}) \leqslant p_I(\text{length}(x)). \tag{2}$$

Second: Let r_{INT} be a run of *INT* with input $F(t)$ such that $r_{INT} = R(w)$, as given in the proof of 6.2, where w is an optimal computation of IMPL such that

$$\text{cost}_{\text{IMPL}}(F(t)) = \text{length}(w). \tag{3}$$

As shown in $(*)$ in the proof of 6.2, we have for some $b > 1$

$$\text{length}(r_{INT}) \leqslant b^{\text{length}(w)}. \tag{4}$$

Third: Let $r_{M(0)}$ be a run of $M(0)$ with last term last(w) as input. By 6.1(3) the run $r_{M(0)}$ is bounded in length by a monotone polynom p_O such that we have for $a > 0$

$$\begin{aligned}\text{length}(r_{M(0)}) &\leqslant p_O(\text{size}(\text{last}(w))) \\ &\leqslant p_O(a^{\text{length}(w)})\end{aligned} \tag{5}$$

using the fact in the proof of 6.2.

Now define r to be the composition od $r_{M(I)}$, r_{INT} and $r_{M(0)}$; then we have from (2), (4) and (5) for some $a > 1$

$$\begin{aligned}\text{length}(r) &\leqslant a \cdot (\text{length}(r_{M(I)}) + \text{length}(r_{INT}) + \text{length}(r_{M(0)})) \\ &\leqslant a \cdot (p_I(\text{length}(x)) + b^{\text{length}(w)} + p_O(c^{\text{length}(w)})),\end{aligned}$$

using (3) and (1) we obtain

$$\leqslant a(p_I(\text{length}(x)) + b^{\text{comp}_{\text{IMPL}}(F)([t])} + p_O(c^{\text{comp}_{\text{IMPL}}(F)([t])}))$$

and thus the existence of some constant $d > 1$ such that

$$\text{length}(r) \leqslant d^{\text{comp}_{\text{IMPL}}(F)(b_I^{-1}(x))}$$

assuming without loss of generality that $\text{length}(x) \leqslant b^{\text{comp}_{\text{IMPL}}(F)([t])}$. This proves (2) of 6.3. ∎

Immediately from 6.3 we derive a lower bound result for function implementation problems:

6.4. Corollary. *Let $f: I^* \to \bar{O}^*$ be a function as in 4.1 and $B: I^* \to \mathbb{N}_0$ be a lower bound on the nondeterministic time of f. Then for all solutions* (IMPL, F) *of the function implementation problem for f we have*

$$\text{comp}_{\text{IMPL}}(F)(x) \geqslant c \cdot \log(B(b_I(x)))$$

for all $x \in (T_{\text{SPEC0}})_{inputs}$ and some constant $c > 0$.

6.5. *Remark.* One would wish to have a tighter connection between nondeterministic time and algebraic complexity than the one given in 6.3. This in fact is to expected if the proof of 6.2 is refined at a certain point: The estimation $(*)$ where $\text{length}(R(w)) \leqslant c^{\text{length}(w)}$ is derived is exponential since the size of terms which are produced from the interpreter in intermediate steps can grow exponentially, as long as terms and term replacement are represented by trees and tree replacement, respectively. The refinement now would provide a representation by collapsed trees and (somewhat involved) subgraph replacement so that the representation of terms can grow at most polynomial, or even linear in suitable cases.

We expect that a precise analysis of such an improved interpreter yields a strong relationship between nondeterministic time and algebraic complexity of implementations. Furthermore this would be one of the main steps towards a hierarchy theorem for algebraic implementations and probably would also be of practical interest.

Another gap in our estimation for complexity of implementations is that the upper bound result concerns deterministic Turing Machines while the lower bound results concern nondeterministic machines. To fill this gap one would need the specification and implementation of nondeterministic Turing Machines.

7. Conclusion

This paper extends and further develops part of the material in our STOC-paper [13]. It is a condensed version of our technical report [14] (with the same title) where also more details concerning the concepts of algebraic specifications and implementations are given, as well as a correctness proof of our Turing Machine specification 4.5.

Besides the introduction of new concepts, the main results in this paper are sketched as follows:

Given a function $f: I^* \to \bar{O}^*$

1. If f is total recursive, then there is an algebraic specification of f (the converse is not true without some restrictions on the specification).

2. If f is computable by an algebraic time bounded Turing Machine TM (equivalent to being total recursive), then there is an upper bound UB on the time of TM and the algebraic complexity of the time bound is also an upper bound on the algebraic complexity of f, i.e., there is a solution (IMPL, F) of the function implementation problem for f with

$$\text{comp}_{\text{IMPL}}(F) \leqslant UB.$$

3. If LB is a lower bound on the nondeterministic time for f, then $\log(LB)$ is a lower bound on the function implementation problem for f, i.e., for all solutions (IMPL, F) of the function implementation problem for f

$$\log(LB) \leqslant \text{comp}_{\text{IMPL}}(F).$$

4. f is total recursive if and only if the function implementation problem is solvable.

5. There are function implementation problems with nontrivial complexity (which may be derived from (3) above and existence of lower bounds on nondeterministic time of Turing Machines).

The proposed concepts and results on the complexity of algebraic implementations for abstract data types are a first approach to studying the questions on complexity of abstract data types on the level of algebraic specifications. We intended to treat these questions independent of classical complexity measures, like time of Turing Machines or random access machines, and therefore define and investigate cost and complexity solely in the given algebraic calculus. The results of Sections 4, 5 and 6 show compatibility of classical measurement with ours even though the relation is not satisfying in all its aspects:

Upper and lower bounds on the IMPL-complexity of total recursive functions differ exponentially, and in nature by the use of deterministic and nondeterministic time, respectively.

A tighter connection between nondeterministic time and algebraic complexity (IMPL-complexity) is worth considering and should be extended towards a hierarchy theorem for algebraic implementations.

ACKNOWLEDGMENTS

This paper is part of a project on algebraic specifications and implementations of abstract data types at *TU* Berlin. We are most grateful to the other members, H.-J. Kreowski and P. Padawitz, of this project, all members of the *ADJ*-group, K. Fleischmann, C. Lautemann, and D. Siefkes for several valuable discussions concerning algebraic implementations and complexity within the framework of this paper which was also partly motivated by [23]. Special thanks also to the referee for his very careful and detailed report which convinced us to rewrite the complete paper.

REFERENCES

1. J. A. GOGUEN, J. W. THATCHER, AND E. G. WAGNER, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," IBM Research Report RC-6487, 1976; and *in* "Current Trends in Programming Methodology." Vol. IV, "Data Structuring" (R. Yeh, Ed.), Prentice-Hall, Englewood Cliffs, N. J., 1978.
2. J. W. THATCHER, E. G. WAGNER, AND J. B. WRIGHT, Data type specification: Parameterization and the power of specification techniques, *in* "Proceedings, 10 SIGACT Symposium on Theory of Computing, San Diego, 1978," pp. 119–132.
3. H. EHRIG, H.-J. KREOWSKI, J. W. THATCHER, E. G. WAGNER, AND J. B. WRIGHT, Parameterized specifications in algebraic specification languages, *in* "Proceedings, ICALP'80, Noordwijkerhout," pp. 157–168, Springer Lecture Notes in Computer Science, Vol. 85, Springer-Verlag, New York/Berlin, 1980.
4. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass., 1974.
5. R. M. BURSTALL AND J. A. GOGUEN, Putting theories together to make specifications, *in* "Proceedings International Conference on Artificial Intelligence, Boston, 1977."
6. J. A. BERGSTRA AND J. V. TUCKER, A characterization of computable data types by means of a finite, equational specification method, *in* "Proceedings, Conf. ICALP'80, Noordwijkerhout," pp. 76–90, Springer Lecture Notes in Computer Science, Vol. 85, Springer-Verlag, New York/Berlin, 1980.

7. H. D. EHRICH, Extensions and implementations of abstract data type specifications, *in* "Proceedings Conf. MFCS'78, Zakopane," pp. 155–163, Springer Lecture Notes in Computer Science, Vol. 64, Springer-Verlag, New York/Berlin, 1978.
8. H. D. EHRICH, "On the Theory of Specification, Implementation and Parametrization of Abstract Data Types," Forschungsbericht Uni. Dortmund, 1978.
9. H. EHRIG, H.-J. KREOWSKI, B. MAHR, AND P. PADAWITZ, Compound algebraic implementations: An approach to stepwise refinement of software systems, *in* "Proceedings, Conf. MFCS'80, Rydzyna," pp. 231–245, Springer Lecture Notes in Computer Science, Vol. 88, Springer-Verlag, New York/Berlin, 1980.
10. H. EHRIG, H.-J. KREOWSKI, B. MAHR, AND P. PADAWITZ, Algebraic implementations for abstract data types, *Theoret. Comput. Sci.*, in press.
11. H. EHRIG, H.-J. KREOWSKI, AND P. PADAWITZ, Stepwise specification and implementation of abstract data types, *in* "Proceedings, Conf. ICALP'78, Udine," pp. 205–226, Springer Lecture Notes in Computer Science, Vol. 62, Springer-Verlag, New York/Berlin, 1978.
12. H. EHRIG, H.-J. KREOWSKI, AND P. PADAWITZ, Algebraic implementation og abstract data types: Concept, syntax, semantics and correctness, *in* "Proceedings, Conf. ICALP'80, Noordwijkerhout," pp. 142–156, Springer Lecture Notes in Computer Science, Vol. 85, Springer-Verlag, New York|Berlin, 1980.
13. H. EHRIG AND B. MAHR, Complexity of implementations on the level of algebraic specifications, *in* "Proceedings, ACM Symposium on Theory of Computing, Los Angeles, 1980," pp. 281–293.
14. H. EHRIG AND B. MAHR, "Complexity of Algebraic Implementations for Abstract Data Types: Technical Report Version," Forschungsbericht der *TU* Berlin, FB 20, Nr. 80–33, 1980.
15. J. A. GOGUEN AND F. NOURANI, "Some Algebraic Techniques for Proving Correctness of Data Type Implementation," Extended Abstract, Computer Science Department, UCLA, Los Angeles, Calif. 1978.
16. J. V. GUTTAG, "Abstract Data Types and the Development of Data Structures," Supplement to "Proceedings, Conf. on Data Abstraction, Definition, and Structure, SIGPLAN Notices 8, March 1976.
17. M. A. HARRISON AND R. J. LIPTON, "Implementation of Abstract Data Types," Extended Abstract, Computer Science Division, University of California, Berkeley, 1979.
18. U. L. HUPBACH, Abstract implementation of abstract data types, *in* "Proceedings, Conf. MFCS'80, Rydzyna," pp. 291–304, Springer Lecture Notes in Computer Science, Vol. 88, Springer-Verlag, New York/Berlin, 1980.
19. D. KOZEN, Complexity of finite presented algebras, *in* "Proceedings, 9th ACM Symposium on Theory of Comp., 1977."
20. D. H. LEHMANN AND M. B. SMYTH, Data types, *in* "Proceedings, 18th IEEE Symposium on Found. of Computing, Providence, R.I., November 77," pp. 7–12.
21. N. LYNCH, Straight-line program length as a parameter for complexity measures, *in* "Proceeding, 10th ACM Symposium on Theory of Comp., 1978."
22. B. MAHR, "Measurable Implementations of Abstract Data Types," Forschungsbericht der TU Berlin, in press.
23. D. SIEFKES, Recursive equations as a programming system, *in* "Proceedings, Frege-Konf. May 1979, Jena (DDR)," in press.
24. M. WAND, "Final Algebra Semantics and Data Type Extensions," Indiana University, Computer Science Department, Technical Report No. 65, 1977.
25. S. N. ZILLES, "An Introduction to Data Algebras," working draft paper, IBM Research, San José, Spetember 1975.