Abstract algorithms

Claus Diem

September 17, 2014

Abstract

We give a framework to argue formally about algorithms with arbitrary data types. The framework is based on category theory, and types are based on collections of categories whose only morphisms are isomorphisms. The framework includes definitions for representations of data types and of specifications of commands in algorithms by other algorithms. The framework is well suited to establish the correctness of algorithms and to analyze their time complexity.

1 Introduction

The notion of *algorithm* is arguably the central notion of computer science and algorithmic mathematics. However, the word "algorithm" is used in a remarkably diverse manner. In some contexts, the word "algorithm" is a synonym for a formally defined object like a Turing machine or a random access machine or some other mathematical object related to computation. However, often the word "algorithm" is used in an informal way. Here, usually, using so-called pseudo-code some computation is outlined.

Of course, in order to prove mathematical statements, including such related to algorithms, one first needs mathematical objects one can argue about. For this, the gap between algorithms in the informal and the formal sense is often bridged by fixing representations of the mathematical objects involved in the description of the algorithm by "easier" objects. Most of the time, the representations are via bit-strings, but occasionally some other objects, like for example real numbers, are also considered.

An interesting aspect of the *informal use* of the notion of algorithm is however that often it does not seem to be necessary to fix such representations at all: One can just argue about the "algorithms" *as is* by assuming that variables can store the indicated objects and that via the commands the indicated "computations" are performed – no matter how complex these are. For example, one can often argue about the termination and the correctness of an "algorithm" which is given without any reference to a formal

model. Also, one can often argue about its complexity, provided one uses an appropriate complexity measure related to the operations present.

There are however obvious problems related to the approach just taken which are related to the lack of rigorous foundations of argumentations about what we just called "algorithms". This motivates to search for a *formal model* or a *formal framework* in which one can argue *rigorously* about algorithms operating with a very wide range of data types and using a very wide range of commands. Such a framework is presented in this work: the framework of *abstract algorithms*.

1.1 The framework of abstract algorithms

An *abstract algorithm* as defined in the following is a *mathematical object*, so one can argue about it in just the same way as one can argue about a Turing machine or a random access machine. On the other hand, the framework of abstract algorithms is so flexible that it can handle data of any "type". We describe now the main ideas which are behind the framework and which are realized by it.

The framework is based on ideas from category theory. The first idea which has been realized is:

- Every object considered is by definition an object from a particular category, and it is this category which is the *type* of the object.

Further ideas which have been realized in the model are:

- The application of a computation to isomorphic objects should lead to comparable results. However, morphisms different from isomorphisms should by themselves play no role in the definitions to be given. For these reasons, we consider in the following objects in categories in which all morphisms are isomorphisms. Let us recall that such categories are called large groupoids.

- In computations, often certain data are considered fixed. It should be possible to define a "computational problem", which is "relative" to the fixed data. Let us as an example consider the composition of elements in groups. Here the underlying group is a priori arbitrary but in the computation it is fixed. This leads to the idea that types should be given by labeled directed graphs, where the labeling is given by categories for the vertices and by functors between these categories for the edges.

- An object should be something elementary. This leads to the idea that types of objects should always have exactly one vertex with indegree 0.

- We wish to be able to represent the objects of some type by other objects. Definitions of "representation" and of "specification" / "implementation" should be part of the framework to be developed. Like this, from a very abstract description, by subsequent specifications, it should be conveniently possible to arrive at a *bit-oriented algorithm*.

- A quite technical but important aspect is that tuples of arbitrary length of objects of the same type are important. For example, vectors of elements of a fixed field can be represented by tuples of elements of the field. (Here a vector is a single object, and a tuple is a collection of objects.) And also tuples of vectors might be used to represent objects, for example matrices. It is therefore natural to consider tuples of tuples with an arbitrary iteration. We call the resulting objects *arrays*. An array can therefore have an arbitrary (finite) dimension.

- A final important point is complexity: If a computation is possible in a bit-oriented algorithm (which, as we recall, is a special case of an abstract algorithm and which can often be obtained by subsequent specifications) in a time of T, it should be possible to perform the computation in the addition RAM model with logarithmic time in a time of $\tilde{O}(T)$. At this point the reader might question how this is possible if, as just stated, the storage in the model proposed here can have an arbitrary dimension whereas in the RAM model, the storage is 1-dimensional. An answer is that the dimension is of minor importance in the RAM model. Very briefly, the reason for this is as follows: Via the Cantor pairing function ([Wik]), $\mathbb{N}_0^2$ is bijective to $\mathbb{N}_0$. This function is given by $(a, b) \mapsto \frac{1}{2}(a+b)(a+b+1)+b$. The computation can be performed in a time of $\tilde{O}(\log(a) + \log(b)))$.

1.2 Relationship with related works

The computer algebra system MAGMA. A major guideline for the framework of "abstract algorithms" was the computer algebra system MAGMA ([BCFS11]). The MAGMA language is designed in such a way that the code highlights the role of mathematical objects one wishes to perform computations with. An important feature of MAGMA is that the internal representation of objects is often hidden from the user of the language. Also, the MAGMA programs are often easily humanly readable and might be used where one might otherwise use pseudo-code to describe algorithms (in the informal sense).

This gives rise to the idea that it should be possible to develop a framework with the following characteristics: It should be possible to argue algorithms which are similar to the ones in MAGMA. Moreover, this should

be possible without considering representations of the objects involved by bit-strings.

Abstract state machines. The task to develop a formal framework in which one can conveniently and at the same time rigorously describe "abstract" algorithms was already taken up by Yuri Gurevich about 30 years ago. The framework was first called *evolving algebras* and then *abstract state machines (ASM)* by him and other researchers (see for example [Gur99], [Gur00], [BS03]). There is now a community of researchers working on this model, and the model is indeed used for the formal analysis of algorithms. For example, there is a book in which the concept is used to analyze the programming language Java and the the Java Virtual Machine ([SSB01]). So the question arises in which aspects the framework of abstract state machines and the framework developed here agree and differ. The following comments might give an indication of the similarities and differences. The reader should however be aware that the author is not an expert on abstract state machines.

Upon comparing the two approaches one realizes that even though the general purpose to develop the ASM framework and this framework seem to be similar, they do differ on a fundamental level and not only in some aspects. For this reason it does not seem to be reasonable to make a table-like comparison between the two approaches. So we contrast them here in their totality.

We first again highlight some of the most important aspect of the framework of abstract algorithms developed here: The starting point is what we call complex types; these are based on graphs of categories of large groupoids. A computational problem is a transformation between types. The commands of abstract algorithms are themselves based on appropriate computational problems. We stress that the commands are not interpreted by computational problems but that the computational problems are a defining part of the commands. This can be captured by saying that there is no difference between the syntax and the semantics of the programming language. The operation of an abstract algorithm is similar to that of a random access machine. Furthermore, it includes a framework for representations of types and for specification or implementation. Care has been taken that one can naturally obtain bit-oriented algorithms with a complexity with is similar to the one one can obtain by a direct use of the usual addition RAM model.

In the ASM framework, there is a clear distinction between syntax and semantics: The starting point for algorithms in the sense of abstract state machines is the vocabulary which is a collection of so-called *function names*. Each function name has an arity, and the function names of arity 0 can be

regarded as variables. A *state* of the algorithm is then given by an algebra in the sense of general algebra associated to the system of function names. By definition, a state can always be defined in terms of first-order logic. An algorithm consists of a finite system of commands which are expressions in the vocabulary and **if ... then ... else**-commands; **goto** or loop commands are not present. These commands are executed simultaneously. An algorithm therefore operates on the set of states, associating to each state a new state. By subsequent operations of an algorithm, one obtains a sequence of states. This sequence of states, that is, evolution of algebras, describes the operation of the algorithm.

Generally speaking, one can say that the framework of abstract state machines is more related to logic and symbolic methods whereas our approach emphasizes more the underlying mathematical ideas of computations. One might also say that the framework of abstract state machines gives a logician's view on algorithms whereas the framework developed here is inspired by the categorial point of view on mathematics. It comes with this that the theory of abstract state machines is more suited to analyze formal questions in the science of computing, for example questions related to programming languages or the behavior of programs. In contrast, the framework of abstract algorithms should be more suited to highlight the mathematical ideas underlying algorithms and to obtain complexity theoretic results.

Possible relationships with other works. The author is not aware of any substantial relationships with other works or approaches. We mention that just by the fact that the approach is based on category theory and also because of the terminology, a reader might expect a relationship with type theory. We see however no such relationship which is worth discussing.

1.3 What follows

Overview. The next section has two parts: In the first part we introduce what we call *elementary* and *complex types*, and in the second part we introduce *computational problems*.

The third section is on the definition and operation of abstract algorithms. After some initial comments on random access machines, we present some ideas on the storage of objects and complexes of objects as defined in the first section. Then in the third subsection of the section, we give the definition of *abstract algorithms*. From a logical point of view, this definition is independent on the storage of objects; one just needs to know what the storage is. We then define the operation of an abstract algorithm on states. Finally, we briefly discuss complexity theoretic issues.

In the first two subsections of the forth section, first the definition of

representations is given and then the relationship between computational problems and representations is discussed. From a logical point of view, these two sections depend solely on the second section. Then a definition for *specifications* or – what is by definition the same – *implementations* is given. Finally specifications to bit-oriented computations are discussed, and the tight complexity theoretic relationship to computations in addition RAMs presented.

In Section 5 we give perspectives for expansion of the framework. Finally, in Section 6 we discuss more in depth what we consider to be relevant aspects of the dichotomy between the formal and the informal use of the word "algorithm".

Throughout the presentation, for simplicity we speak simply of algorithms instead of abstract algorithms (cf. Terminology 59). Thus the word "algorithm" has a specific technical meaning which we are going to define. We remark that we do so for simplicity in the context of the presentation. When referring to the definitions and concepts presented here, the phrase "abstract algorithms" should be used.

Basic terminology. We use the following terminology.

- The integer 0 is not a natural number. We denote the natural numbers by $\mathbb{N}$ and the non-negative integers by $\mathbb{N}_0$.
- Tuples might be empty. We denote the empty tuple by $()$.

Linguistic guidelines for the definitions. Purely on a linguistic level, we tried to find names for the concepts which fit well together. In particular, a guiding principle is to avoid what Daniel Bernstein calls "non-restrictive" and "content-free" adjectives ([DJB]). This means that when we qualify a noun referring to a mathematical object with an adjective, the reader can be assured that the scope is more restricted than that given by the noun alone. There is only the following classical exception to this rule: The word "partial" is used in a generalizing way, analogously to the notion of a "partial function" being more general than the notion of a "function". In addition, we introduce "elementary types" and "complex types" but we do not introduce "types" themselves. So, "elementary" and "complex" at not attributes to "type".

The document itself. It is surely difficult to keep track of the great amount of definitions. In order to facilitate the reading a bit, we inserted hyperlinks into the document. These hyperlinks can be followed with a usual reader for PDF-documents. With such a reader one can also view a table of contents.

2 Computational problems and abstract algorithms

We start with a proposal for a definition of (abstract) computational problems, based on category theory.

2.1 Types

Recall that a category in which all morphisms are isomorphisms is called a *large groupoid*. Such categories are fundamental for the following. We denote them by bold letters, i.e. $\boldsymbol{A}, \boldsymbol{B}, \ldots$.

Definition 1 For a large groupoid $\boldsymbol{A}$ we define $\mathrm{array}^1(\boldsymbol{A})$ as the large groupoid given as follows: The objects are tuples of arbitrary positive length of objects of $\boldsymbol{C}$. In order that there can be a morphism between two tuples these must be of the same length. A morphism from $(a_1, \ldots, a_n)$ to $(b_1, \ldots, b_n)$ is then a tuple of (iso-)morphisms from a_i to b_i.

Now, for $d \in \mathbb{N}$, we denote the category obtained by d-fold iteration of this construction by $\mathrm{array}^d(\boldsymbol{A})$. Moreover, we set $\mathrm{array}^0(\boldsymbol{A}) := \boldsymbol{A}$. We call the objects of $\mathrm{array}^d(\boldsymbol{A})$ *d-dimensional arrays* of objects of $\boldsymbol{C}$ and consequently $\mathrm{array}^d(\boldsymbol{A})$ the *category of d-dimensional arrays over $\boldsymbol{A}$*.

Definition 2 Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be large groupoids, and let f be a functor from $\boldsymbol{A}$ to $\boldsymbol{B}$. We define a functor from $\mathrm{array}^1(\boldsymbol{A})$ to $\mathrm{array}^1(\boldsymbol{B})$ as follows: A tuple $(a_1, \ldots, a_n)$ is mapped to $(f(a_1), \ldots, f(a_n))$. A morphism from $(a_1, \ldots, a_n)$ to $(b_1, \ldots, b_n)$ in $\boldsymbol{A}$, which is by definition given by a tuple of morphisms $(\alpha_1, \ldots, \alpha_n)$, is mapped to $(f(\alpha_1), \ldots, f(\alpha_n))$, which is a morphism from $(f(a_1), \ldots, f(a_n))$ to $(f(b_1), \ldots, f(b_n))$. We say that the functor just defined is given by *componentwise application* of f, and we denote it again by f. We iterate this definition to arrays of higher dimension. In particular, given a functor from $\boldsymbol{A}$ to $\mathrm{array}^d(\boldsymbol{B})$ for some $d \in \mathbb{N}_0$, we obtain induced functors from $\mathrm{array}^i(\boldsymbol{A})$ to $\mathrm{array}^{d+i}(\boldsymbol{B})$ for any $i \in \mathbb{N}_0$.

In the following, we consider labeled directed graphs. We fix the following terminology:

Terminology 3 By a *path* in a directed graph we mean a directed path.

We now describe the labeled directed graphs we consider:

A graph as shall be considered has a vertex set V, where an edge between two vertices v and w is given by the tuple (v, w) (denoted by vw). The labeling is given as follows: The vertices are labeled by tuples $(\boldsymbol{A}, d)$, where $\boldsymbol{A}$ is a large groupoid and $d \in \mathbb{N}_0$; the interpretation is that such a tuple defines the category of d-dimensional arrays over $\boldsymbol{A}$. If a vertex is labeled by $(\boldsymbol{A}, d)$, we call $\boldsymbol{A}$ the *category* of the vertex and d the *dimension* of the array

defined at the vertex. (Note that we do not call $\mathrm{array}^d(\boldsymbol{C})$ the category of the vertex.)

In order that there can be an edge from a vertex labeled by $(\boldsymbol{A}, d)$ to one labeled by $(\boldsymbol{A}', d')$ it is necessary that $d \leq d'$. Such an edge is then labeled by a functor f from $\boldsymbol{A}$ to $\mathrm{array}^{d'-d}(\boldsymbol{A}')$. Here we call the integer $d' - d$ the *target dimension* of the edge. We extend the functor f to a functor from $\mathrm{array}^d(\boldsymbol{A})$ to $\mathrm{array}^{d'}(\boldsymbol{A}')$ by applying the functor f componentwise. With this extension we can speak of *commutativity* of such a graph – which is defined in the obvious way.

A problem we encounter is that graphs as described with different vertex sets are by definition different. This is however quite unnatural as can be seen by considering graphs with just one vertex: Let us fix some category $\boldsymbol{A}$, some $n \in \mathbb{N}_0$ and two different elements $*$ and $\circ$. Then the labeled graph on the vertex set $\{*\}$ with labeling $(\boldsymbol{A}, d)$ and the labeled graph on the vertex set $\{\circ\}$ with the same labeling are different. On the other hand, of course it is natural to identify them.

In general, it is natural to identify objects in a category if there is a certain unique distinguished isomorphism between them. However, the graphs we consider do have non-trivial automorphisms. It is therefore reasonable to consider a more rigid definition. For this reason, we proceed as follows:

We consider graphs as described above. On such a graph we consider now the following additional structure: First, we fix a total ordering on the set of vertices with outdegree 0. Second, for each vertex v we fix a total ordering on the set of edges ending at v – equivalently an ordering at the set of vertices from which there is an edge which ends at v. Note that a total ordering on the set of vertices induces an ordering as described.

Notation 4 We denote such structures also by bold letters, for example by $\boldsymbol{A}, \boldsymbol{B}, \ldots$. The vertex set of $\boldsymbol{A}$ is then denoted by $V_{\boldsymbol{A}}$. If v is a vertex of $\boldsymbol{A}$, we denote the corresponding category by $\boldsymbol{A}_v$ and the corresponding dimension by $d_{\boldsymbol{A},v}$. For an edge vw, we denote the corresponding target dimension by $d_{\boldsymbol{A},vw}$.

For any two vertices v, w of $\boldsymbol{A}$ such that there is a path from v to w, we denote the composition of the functors for the corresponding edges by $f^{\boldsymbol{A}}_{vw}$. We define the *target dimension* of the path as the sum of the target dimensions of the edges in the path, which is equal to the difference between the end and the starting point of the path. We denote it again by $d_{\boldsymbol{A},vw}$.

Remark and Definition 5 Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be two such structures. By a *strong isomorphism* between $\boldsymbol{A}$ and $\boldsymbol{B}$ we mean a bijection from $V_{\boldsymbol{A}}$ to $V_{\boldsymbol{B}}$ which induces an automorphism of directed graphs respecting the labels, the

ordering on the vertices with outdegree 0 and for each edge the ordering on the ending edges.

The crucial feature is now that there is at most one strong isomorphism from $\boldsymbol{A}$ to $\boldsymbol{B}$. If there is one, we say that $\boldsymbol{A}$ and $\boldsymbol{B}$ are *essentially equal*, and we write $\boldsymbol{A} \cong \boldsymbol{B}$.

Definition 6 An *elementary type* is a commuting graph with the "additional structure" described above with exactly one vertex with indegree 0 (that is, there are no functors in the graph terminating at the category) for which moreover the dimension of the array is equal to 0 (that is, the category of arrays is equal to the underlying category). This unique vertex is called the *top vertex* of the elementary type and the corresponding category the *top category*. The top category of an elementary type $\boldsymbol{A}$ is denoted by $\mathrm{Top}(\boldsymbol{A})$.

Remark 7 The dimensions of the vertices of an elementary type are uniquely determined by the target dimensions of the edges.

Definition 8 A *typed object* is a tuple consisting of an elementary type and an object in the top category of the type. The elementary type is then called the *type* of the typed object.

Terminology 9 Instead of typed objects of elementary type $\boldsymbol{A}$ we speak of *objects of type* $\boldsymbol{A}$.

Moreover, instead of saying, for example, that a is an object of the elementary type of commutative algebras, we say that a is an object of type commutative algebra.

Remark 10 Let an object of type $\boldsymbol{A}$ be given. Then for every vertex v of $\boldsymbol{A}$, via the application of the corresponding functor for the edge from the top edge to v we obtain an an object of the category $\boldsymbol{A}_v$.

We fix the following notation.

Notation 11 Typed objects and the underlying "plain" objects are usually denoted in the same way. However, sometimes, it is natural to consider an object of one type also as an object of another type. In this case, we use different notations for the different typed objects.

Terminology 12 If $\boldsymbol{A}$ is an elementary type with longest path length $d-1$, we say that $\boldsymbol{A}$ is a d^{th} *order elementary type*. For any natural number i, the i^{th} *layer* of $\boldsymbol{A}$ is the set of all vertices v of $\boldsymbol{A}$ for which the longest path starting at v has length i.

By definition, a *first order elementary type* is the same as a large groupoid. We give some examples.

Example 13 Any set A gives a type by turning A into a category whose only morphisms are the identities. This category is then denoted by $\boldsymbol{A}$ (and similarly by the bold letters corresponding to the notation of the set).

Examples 14 From any category one can obtain a first order elementary type by deleting the morphisms which are not isomorphisms. We obtain in this way the elementary types of sets, groups, rings, fields, etc.

Let now $\boldsymbol{A}_1$ be one of the four mentioned categories. We can then consider "pointed objects"; these are objects of the form (a, P), where $a \in \boldsymbol{A}_1$ and $P \in a$. These pointed objects form a class, and if we define the morphisms from (a, P) to (a', P') as the isomorphisms from a to a' which map P to P', we obtain another large groupoid which is a category over $\boldsymbol{A}_1$. Therefore, we have defined a second order elementary type.

A further example of a second order elementary type is given by commutative algebras. Pointed commutative algebras then define a third order elementary type.

The inputs to the computational problems will not only consist of one typed object but of several typed objects. These typed objects might be intertwined, for example, one might consider two pointed groups, where the groups are the same, or one might consider two commutative algebras over the same ring. This leads to the next definitions.

Definition 15 An arbitrary structure as described above in Notation 4 is called a *complex type*.

Definition 16 Let $\boldsymbol{C}$ be a complex type. Furthermore, let for every vertex v of $\boldsymbol{C}$ a $d_{v,\boldsymbol{C}}$-dimensional array over $\boldsymbol{C}_v$, that is, an object of $\mathrm{array}^{d_{\boldsymbol{C},v}}$, be given such that for every edge vw of $\boldsymbol{C}$, the selected object in $\mathrm{array}^{d_{\boldsymbol{C},v}}(\boldsymbol{C}_v)$ is mapped to the one in $\mathrm{array}^{d_{\boldsymbol{C},w}}(\boldsymbol{C}_w)$ via the functor $f^{\boldsymbol{C}}_{vw}$. Then we call this family of objects an *object complex*. If we attach to each object in the complex the respective category, we obtain a *typed object complex*.

Terminology 17 As for objects of an elementary type, instead of typed object complexes of complex type $\boldsymbol{C}$ we speak of *object complexes of type* $\boldsymbol{C}$.

Remark 18 Every elementary type is a complex type. Moreover, by Remark 10 every object of some type defines in a unique way an object complex of the same type.

Remark 19 The notions of an elementary type of d^{th} *order* generalizes immediately to complex types. So does the notion of the i^{th} *layer*.

Remark 20 In the definition of $\text{array}^1(\boldsymbol{C})$ we consider tuples (arrays) of arbitrary positive length. We do not define here arrays of a fixed length because this is not really necessary: The definition of *complex type* has enough flexibility to include the idea of tuples of a fixed length. Indeed, this would be the case even if the labels of the edges would be plain categories and not categories of arrays.

Remark and Definition 21 Let $\boldsymbol{C}$ be a complex type. There is an obvious definition of morphisms between object complexes of type $\boldsymbol{C}$. Again every morphism is an isomorphism. Thus, the object complexes of a given type form a category. We call this category the *category of object complexes of type* $\boldsymbol{C}$ and denote it by $\text{Cat}(\boldsymbol{C})$.

Remark 22

a) For an elementary type $\boldsymbol{A}$ we have $\text{Top}(\boldsymbol{A}) = \text{Cat}(\boldsymbol{A})$.

b) Any strong isomorphism between complex types $\boldsymbol{C}$ and $\boldsymbol{D}$ induces an isomorphism between $\text{Cat}(\boldsymbol{C})$ and $\text{Cat}(\boldsymbol{D})$. This implies that if $\boldsymbol{C}$ and $\boldsymbol{D}$ are essentially equal, then $\text{Cat}(\boldsymbol{C})$ and $\text{Cat}(\boldsymbol{D})$ are canonically isomorphic.

c) We do not rule out that the graphs defining a complex type is empty. We then have the *trivial complex type* which we denote by $\emptyset$. There is exactly one element of this type, the *trivial complex*.

Definition 23 Let $\boldsymbol{C}$ be a complex type. We say that a complex type $\boldsymbol{D}$ is *over* $\boldsymbol{C}$ or is *derived* from $\boldsymbol{C}$ if the following holds:

- The graph underlying $\boldsymbol{C}$ is a subgraph of the graph underlying $\boldsymbol{D}$, and the orderings are compatible.
- For each vertex v of $\boldsymbol{C}$, we have $\boldsymbol{C}_v = \boldsymbol{D}_v$ and $d_{\boldsymbol{C},v} \leq d_{\boldsymbol{D},v}$.
- For each vertex v of $\boldsymbol{C}$ and each edge $e = vw$ of $\boldsymbol{D}$, the vertex w appears in $\boldsymbol{C}$. The target dimensions of the functors in $\boldsymbol{C}$ and in $\boldsymbol{D}$ as well as the functors themselves are identical. (The condition on the dimension means that $d_{\boldsymbol{D},w} - d_{\boldsymbol{D},v} = d_{\boldsymbol{C},w} - d_{\boldsymbol{C},v}$.)

If we always have an equality $d_{\boldsymbol{C},v} = d_{\boldsymbol{D},v}$, we say that the type $\boldsymbol{D}$ *restricts to* the type $\boldsymbol{C}$. Under this condition every object complex c of $\boldsymbol{D}$ defines in an obvious way an object complex of $\boldsymbol{C}$; we call this the *restriction* of c to $\boldsymbol{C}$ and denote it by $\text{res}^{\boldsymbol{D}}_{\boldsymbol{C}}(c)$.

Remark and Definition 24 Let $\boldsymbol{C}$ be a complex type.

a) Let V be a subset of $V_{\boldsymbol{C}}$. We then have the closure of V under "going from the beginning of a vertex to its end". This set is denoted by $\downarrow V$. Note that the set of edges can be seen as defining a relation on the set of vertices and it is the transitive closure of this set we are talking about. We call $\downarrow V$ the *closure of V under going down*. If V is equal to its transitive closure, we call V *closed under going down*.

b) Let now W be a subgraph of $V_{\boldsymbol{C}}$ which is closed under going down. We have the *full-dimensional type* defined by W, which is defined in the obvious way. We denote it by $\boldsymbol{C}^{\text{full}}_{|W}$. If W is the closure of some set V under under going down we denote it by $\boldsymbol{C}^{\text{full}}_{\downarrow W}$. If moreover $V = \{v\}$ for some vertex v, we denote it by $\boldsymbol{C}^{\text{full}}_{\downarrow v}$.

c) If now a is an object of $\boldsymbol{C}$, we have the restriction to $\boldsymbol{C}^{\text{full}}_{|W}$ which we denote by $\text{res}^{\boldsymbol{C}}_{W}(b)$. If W is the closure under going down of V, we denote it by $\text{res}^{\boldsymbol{C}}_{\downarrow V}(b)$, and if $V = \{v\}$ we denote it by $\text{res}^{\boldsymbol{C}}_{\downarrow v}(b)$.

Definition 25 Let now $\boldsymbol{B}$ and $\boldsymbol{C}$ be two complex types which restrict to the complex type $\boldsymbol{A}$. Then we define a new type, $\boldsymbol{B} \times_{\boldsymbol{A}} \boldsymbol{C}$, as follows:

The set of vertices is given by the disjoint union of the sets of vertices of $\boldsymbol{B}$ and $\boldsymbol{C}$ modulo the identification with respect to the set of vertices given by $\boldsymbol{A}$. The graph is labeled by categories and functors in the obvious way as induced by $\boldsymbol{B}$ and $\boldsymbol{C}$. The orderings are defined as follows: On the set of vertices with outdegree 0, first there are all vertices in $V_{\boldsymbol{B}} \backslash V_{\boldsymbol{A}}$, then there are all vertices in $V_{\boldsymbol{A}}$ and finally there are the vertices in $V_{\boldsymbol{C}} \backslash V_{\boldsymbol{A}}$. Inside these sets, the orderings are the obvious ones. For each vertex v, the ordering on the set of edges ending in v is also as just described.

Definition 26 Let now $\boldsymbol{C}$ be a complex type and let v be a vertex of $\boldsymbol{C}$. Then there is a unique elementary type $\boldsymbol{A}$ such that

- $\boldsymbol{C}$ is derived from $\boldsymbol{A}$,
- the top-vertex of $\boldsymbol{A}$ is v and this vertex has dimension 0,
- the set of vertices of $\boldsymbol{A}$ is the closure of v under "going along the edges".

This type is called the *elementary type defined by v in $\boldsymbol{C}$*. We denote it by $\boldsymbol{C}_{\downarrow v}$.[1]

[1] Note that this need not be equal to $\boldsymbol{C}^{\text{full}}_{\downarrow v}$, which has been defined in Definition 24.

Remark 27 If the type $\boldsymbol{B}$ be derived from the type $\boldsymbol{A}$, then for each vertex v of $\boldsymbol{A}$, $\boldsymbol{A}_{\downarrow v} = \boldsymbol{B}_{\downarrow v}$.

We now illustrate with examples an important principle.

Examples 28 Let us consider the second order type of pointed groups. Then for each group G, which is an object with a first order type, the objects of type pointed group over G form a first order type. These objects are of course exactly the group elements of G. We therefore obtain for every fixed group G the first order type of elements of G.

Let us consider the second order type of commutative algebras. Then for each commutative ring A, which is also an object with a first order type, the objects of type commutative algebra over A form a first order type. These objects are the A-algebras, and we obtain for every fixed commutative ring A the first order type of A-algebras.

Let now $\boldsymbol{A}$ be an elementary type.

Remark and Definition 29 By deleting the top vertex and the edges starting from it, we obtain a complex type which we call the *base* of $\boldsymbol{A}$, Base($\boldsymbol{A}$). Note that if $\boldsymbol{A}$ is a first order type then Base($\boldsymbol{A}$) is empty. If $\boldsymbol{A}$ is not a first order type then for an object of type $\boldsymbol{A}$, we define Base(a) as the object complex of type Base($\boldsymbol{A}$) defined by mapping a along the vertices starting at Top($\boldsymbol{A}$). Note here that as stated in Remark 22 c), if Base($\boldsymbol{A}$) is empty, there still is an object complex of type Base($\boldsymbol{A}$), the trivial complex.

Now for every object complex b of type Base($\boldsymbol{A}$), we obtain the category of objects of type $\boldsymbol{A}$ which are mapped to b. This category might be called the *fiber* defined by b in $\boldsymbol{A}$. This is a first order type which we call the type *defined by* b *in* $\boldsymbol{A}$. Inspired by the usual notation for the fiber, we denote it by $\boldsymbol{A}_b$. We can thus say that we have a family $(\boldsymbol{A}_b)_b$ of first order elementary types indexed by the objects b of type Base($\boldsymbol{A}$). Based on these considerations, we introduce the following notation. For fixed b, every object c of type defined by b in $\boldsymbol{A}$ defines an object of type $\boldsymbol{A}$. We denote this object by

$$(b; c)$$

Clearly, every object of type $\boldsymbol{A}$ is of this form. For an object a of type $\boldsymbol{A}$ we have $a = (\text{Base}(a); c)$ for a unique object c of type $\boldsymbol{A}_{\text{Base}(a)}$.

Examples 30 Coming back to the examples above, we obtain this notation: A pointed group (G, P) is denoted by $(G; P)$. A commutative algebra $A \longrightarrow B$ is denoted by $(A; B)$.

If $\mathrm{Base}(A)$ is also a non-first order elementary type, we can iterate the above:

Notation 31 Let $a = (b; c)$ be an object of type $\boldsymbol{A}$, and let us set $d := \mathrm{Base}(b)$. We then have $b = (d; e)$ for a unique object e of type $\mathrm{Base}(A)_{\mathrm{Base}(b)}$. Therefore $a = ((d; e); c)$. We denote this also by $(d; e; c)$. We might iterative this now.

Example 32 We consider the third order elementary type of pointed commutative algebras. Following the introduced notation, the objects of this type are $(A; B; P)$, where A is a commutative ring, B a commutative A-algebra and $P \in B$.

It might be that all fibers are "identical". This can be captured as follows:

Let $\boldsymbol{C}$ be a complex type. Then for each vertex v of $\boldsymbol{C}$ we have a canonical functor $\mathrm{Cat}(\boldsymbol{C}) \longrightarrow \boldsymbol{C}_v$. In particular, for each elementary type $\boldsymbol{A}$ and each vertex v of $\boldsymbol{A}$ not equal to the top vertex we have a canonical functor $\mathrm{Base}(\boldsymbol{A}) \longrightarrow \boldsymbol{A}_v$, and the functor $\mathrm{Top}(\boldsymbol{A}) = \mathrm{Cat}(\boldsymbol{A}) \longrightarrow \boldsymbol{A}_v$ factors in the two functors $\mathrm{Top}(\boldsymbol{A}) \longrightarrow \mathrm{Base}(\boldsymbol{A})$ and $\mathrm{Base}(\boldsymbol{A}) \longrightarrow \boldsymbol{A}_v$.

Definition 33 An elementary type $\boldsymbol{A}$ is *trivially fibered* if $\mathrm{Top}(\boldsymbol{A})$ is of the form $\mathrm{Base}(\boldsymbol{A}) \times \boldsymbol{F}$ for some first-order type $\boldsymbol{F}$ and the projection $\mathrm{Top}(\boldsymbol{A}) \longrightarrow \mathrm{Base}(\boldsymbol{A})$ is given by projection to the first factor. The type $\boldsymbol{F}$ is then the fiber of every object complex of type $\mathrm{Base}(\boldsymbol{A})$. Correspondingly, it is called the *fiber* of the top-trivialization. The objects of $\boldsymbol{A}$ are therefore of the form $(b; a)$ for $a \in \boldsymbol{F}$.

Remark and Definition 34 If in the above definition there is exactly one object of type $\boldsymbol{F}$ then $\boldsymbol{A}$ is isomorphic to $\mathrm{Base}(\boldsymbol{A})$ via the canonical projection. We express this case by saying that the fiber is *trivial*.

One often encounters the situation that all vertices of a graph of a complex type have outdegree 0 or 1. In this case, one can generalize the notation just presented in an obvious way:

Under the given condition, if one inverts the edges, one obtains a forest (by which we mean a disjoint union of rooted trees). Now forests can be described by elements of a certain term algebra. We assume that this representation is quite well known, but we did not find it in the literature and therefore describe it in the following. As indicated, for an application to objects of complex types, one first has to invert the edges.

We consider labeled forests whose edges are contained in the set $\{1, \ldots, n\}$, where to each i a label a_i in a set A is attached. We now consider the free

monoid on the disjoint union of A with a set consisting of four elements. These for elements are denoted by the symbols ",", ";", "(", ")". As usual, we write the elements in the free monoid as "words" or "strings". We emphasize again that the symbols ",", ";", "(", ")" now represent elements of a set on which we consider words.

To a forest G as described we associate a word $s(G)$ in the following recursive way:

- To the empty forest we associate the empty word.
- To a tree with one edge i, we associate the word (a_i).
- Let a forest with trees $T_1, \ldots, T_k$ with roots $r_1, \ldots, r_k$ with $r_j < r_{j+1}$ for all j be given, and let $s(T_i) = (t_i)$. Then to the forest we associate the word $(t_1, \ldots, t_k)$.
- Let a tree T be given, and let a_i be its root. Upon removing the root, we obtain a forest of trees. Let t be the word associated to the forest (as defined in the previous item). Then to T we associate the word $(a_i; t)$.

Remark 35 It is most natural to apply the above assignment if the edges are enumerated according to a depth-first search. To demonstrate the result, let a forest with set of edges exactly $\{1, \ldots, n\}$ be given, where the enumeration is according to a breadth-first search. Moreover, let the edges be labeled by themselves. In this case the associated word has the following feature:

All numbers from 1 to n occur in it exactly once, and from left to right they occur in the "natural ordering" of $1, \ldots, n$.

Discussion

One might ask if an elementary object should not be denoted in the same way as its top object and also addressed in a way which reflects its top object.

We demonstrate this with the example of pointed groups. There are two possibilities here: The first possibility is as follows: Instead of a speaking of a pointed group $(G; P)$ (which might itself be denoted by G_* or so), one would speak of a "group element" which is denoted by P. (There are types in the MAGMA language which are based on this idea.) The second possibility is: If the pointed group is denoted by G_*, the notation $(G; P)$ should be changed to $(G; G_*)$.

Our answer is that the general idea presented indeed seems to be nice. There are however arguments to allow the flexibility to not always proceed like this.

On the first proposal: First, we want that types are based on categories. But what would for example the "category of group elements" be? In particular, what would be its isomorphisms? Second, typed objects should be inputs to algorithms. Thus an input to an algorithm could then be a "group element". But here one should remark that this is not how one usually talks about algorithms. If the group is variable (which it is here because "group elements" would just be pointed groups by definition) then one would not say that the input is "a group element" but a group and a group element in the given group. Or conversely, if one said that the input is a "group element", this could cause the impression that the group is fixed.

On the second proposal: It seems a bit unnecessary to have G essentially twice in the notation. Also, again thinking about inputs to algorithms and about storage of objects in algorithms: What has to be input and stored here is really (G, P) and not (G, G_*). So the notation $(G; P)$ better reflects what we imagine really that happens in an algorithm.

Having said this we remark however that sometimes, the second proposal is implicitly fulfilled by the usual notation in mathematics. As an example we consider the second order category of commutative algebras. According to the notation in Example 30, we denote a commutative algebra $A \longrightarrow B$ by $(A; B)$. According to the standard notation, it is also denoted simply by B.

2.2 A formalism for computational problems

We pursuit the idea that a computational problem should in a certain sense be modeled by a transformation between complex types.

Let now two complex types $\boldsymbol{I}$ and $\boldsymbol{T}$ be given. We want to interpret the objects of $\boldsymbol{I}$ as "input instances" and the objects of $\boldsymbol{T}$ as "target instances", and we want to discuss now what an appropriate definition of "computational problem" from $\boldsymbol{I}$ to $\boldsymbol{T}$ should be.

As pointed out at the end of the previous subsection, the object complexes of a given type form themselves categories. One might therefore think that one should define a computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$ as a functor from the category of object complexes of type $\boldsymbol{I}$, $\mathrm{Cat}(\boldsymbol{I})$, to the category $\mathrm{Cat}(\boldsymbol{T})$. However, often in the intuitive description of computational problems, there is a certain arbitrariness. Consider for this the following example.

Example 36 An intuitive description of the "computational problem" we want to define is: Given a univariate polynomial over a field, output whether it has a root over the field, and if this is the case, compute one of these.

We define the complex type $\boldsymbol{I}$ as follows: First, we consider the ele-

mentary type of fields. Over this category we consider the second order elementary type of tuples (k, f), where k is a field and $f \in k[T]$. According to our notation, we have $(k, f) = (k; f)$. A morphism from $(k; f)$ to $(k'; f')$ is an isomorphism from k to k' such that the induced morphism from $k[T]$ to $k'[T]$ maps f to f'.

Moreover, we define the complex type $\boldsymbol{T}$: This is again a second order elementary type over the elementary type of fields. The objects are: First, tuples $(k; a)$, where k is a field and $a \in k$ and, second, for each field k an object called $(k;$ "no"$)$. The morphisms from $(k; a)$ to $(k'; a')$ are isomorphisms from k to k' which map a to a'.

So, the two types are defined but up to now, no definition of the problem within the desired framework of "computation problems" has been given. We emphasize the important point that there is a certain degree of arbitrariness in the intuitively described problem: It does not matter which root is computed. This arbitrariness should be reflected by the definition of the "computational problem".

An obvious idea is to model this arbitrariness as follows: One associates to an instance $(k; f)$ to $(k;$ "no"$)$ if there is no root and otherwise to the set of tuples $(k; a)$ of $\boldsymbol{T}$, where a is a root of f. We stress here that such a set of tuples is not itself an object of $\boldsymbol{T}$; it is its elements which are objects of $\boldsymbol{T}$.

We want that a "computational problem" "applied" to isomorphic objects leads to isomorphic "results".

This in mind, we propose the following definition for computations with from one complex type to outputs of one other complex type.

Definition 37 A *computational problem* $\mathcal{P}$ from a complex type $\boldsymbol{I}$ to a complex type $\boldsymbol{T}$ is an assignment which assigns to every object complex of type $\boldsymbol{I}$ a non-empty class of object complexes of $\boldsymbol{T}$ such that the following holds:

For each two isomorphic object complexes a and a' of type $\boldsymbol{I}$ and each $b \in \mathcal{P}(a)$, there exists an object complex $b' \in \mathcal{P}(a')$ which is isomorphic to b.

We then call the objects in $\boldsymbol{I}$ the *input instances* and the objects in $\boldsymbol{T}$ the *target instances*. The category $\boldsymbol{I}$ itself is called the *input type* and the category $\boldsymbol{T}$ the *output type* of the problem. If b is an object in $\mathcal{P}(a)$ for some object a of $\boldsymbol{I}$, we say that b is a *possible result of* $\mathcal{P}$ *applied to* a.

Computational problems can in an obvious way be composed:

Definition 38 Let $\mathcal{P} : \boldsymbol{A} \longrightarrow \boldsymbol{B}$ and $\mathcal{Q} : \boldsymbol{B} \longrightarrow \boldsymbol{C}$ be two computational problems. Then the *composition* $\mathcal{Q} \circ \mathcal{P}$ of the two problems is the computational problem from $\boldsymbol{A}$ to $\boldsymbol{C}$ which assigns to any object a of $\boldsymbol{A}$ the class

consisting of the objects in $\mathcal{Q}(b)$ for b in $\mathcal{P}(a)$.

One sees easily that $\mathcal{Q} \circ \mathcal{P}$ is a computational problem from $\boldsymbol{A}$ to $\boldsymbol{B}$: Let a and a' be two isomorphic objects of $\boldsymbol{A}$, and let c be an object in $(\mathcal{Q} \circ \mathcal{P})(a)$, $(\mathcal{Q} \circ \mathcal{P})(c)$. Then c is in $\mathcal{Q}(b)$ for some object b in $\mathcal{P}(a)$. Now, b is isomorphic to an object b' in $\mathcal{P}(a')$, and therefore $c = \mathcal{Q}(b)$ is isomorphic to an object in $\mathcal{Q}(b')$.

We have the following four classes of examples:

Example 39 Let the first order type $\boldsymbol{T}$ be obtained from a set T by defining only the identities to be morphisms. A computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$ is now the same as an assignment from the objects of $\boldsymbol{I}$ to $\mathbb{P}(T)$ which maps isomorphic objects to the same sets. (Here for a set X, $\mathbb{P}(X)$ is the power set of X.) If in addition $\boldsymbol{I}$ is also a first order type which is obtained from a set I by defining only the identities to be morphisms, a computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$ is nothing but a map $I \longrightarrow \mathbb{P}(T)$. We call a computational problem defined by a map from the set of bit strings $\{0,1\}^*$ to $\mathbb{P}(\{0,1\}^*)$ a *classical computational problem.*

Example 40 Let $\boldsymbol{I}$ and $\boldsymbol{T}$ be complex types. Then every functor $\mathcal{P}$ from $\mathrm{Cat}(\boldsymbol{I})$ to $\mathrm{Cat}(\boldsymbol{T})$ induces in an obvious way a computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$. We call such a computational problem *functorial.*

Example 41 Let $\boldsymbol{I}$ be an d^{th} order elementary type with top vertex v, let $\boldsymbol{A}$ be a first order type, and let $p : \boldsymbol{A} \longrightarrow \boldsymbol{I}_v$ be a functor. Then we obtain an $(n+1)^{\text{th}}$ elementary type $\boldsymbol{T}$ with top category $\boldsymbol{A}$.

Suppose now that for every object b of $\boldsymbol{A}_v = \boldsymbol{I}_v$ there exists an object a of $\boldsymbol{A}$ with $p(a) = b$ and that for every (iso-)morphism $\beta : b \longrightarrow b'$ of $\mathrm{Cat}(\boldsymbol{I})$ and every a in the *fiber* over b, there exists an object a' in the fiber over b' and an (iso-)morphism $\alpha : a \longrightarrow a'$ with $\beta \circ p = p \circ \alpha$. (Note that $\boldsymbol{A}$ with p is a special case of a category fibered in groupoids.)

Then by assigning to every object of type $\boldsymbol{I}$ its fiber we obtain a computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$.

Example 42 Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be complex types. Then every essentially surjective functor from $\mathrm{Cat}(\boldsymbol{A})$ to $\mathrm{Cat}(\boldsymbol{B})$ defines a computational problem from $\boldsymbol{B}$ to $\boldsymbol{A}$: One assigns to every object complex b of type $\boldsymbol{B}$ the class all object complexes of type $\boldsymbol{A}$ whose image is isomorphic to b in $\mathrm{Cat}(\boldsymbol{B})$.

We can also formally complete Example 36 within the framework of computational problems:

Example 43 As already mentioned, an instance $(k; f)$ is mapped to $(k; "no")$ if f has no roots in k and otherwise to the set of $(k; a)$, where a is a root of f in k.

Remark 44 The above computational problem should be contrasted to a problem which can intuitively be described as follows: Given a univariate polynomial over a field, compute all its roots.

An appropriate target type can now be defined as follows: The objects are tuples $(k; S)$, where k is a field and S is a finite subset of k. The morphisms from $(k; S)$ to $(k'; S')$ are the isomorphisms from k to k'.

The computational problem is now given by the functor which assigns to $(k; f)$ the target instance $(k; \{a \in k \mid f(a) = 0\})$.

Example 45 Let $\boldsymbol{F}$ be any category of fields. We define the second order type $\boldsymbol{I}_0$ just as the type $\boldsymbol{I}$ in Example 36, except that instead of arbitrary fields we only allow finite fields. Now the computational problem defined in examples 36 and 43 restricts to a computational problem from $\boldsymbol{I}_0$ to $\boldsymbol{T}$.

Remark 46 One reason why we took as a first example for a computational problem the problem to determine if a univariate polynomial over a field has a root and if so to compute one is to emphasize that one does not need to formulate any "representation" of the objects concerned, for example via bit-strings. Of course, a representation of the objects up to isomorphism via bit-strings would not even be possible simply because there are uncountably many isomorphism classes of objects.

However, if one considers restricted problems, for example the restricted problem for polynomials over finite fields or over number fields, one might indeed fix a representation of the objects up to isomorphism by bit-strings. In subsection 4.5 we will discuss this aspect. We will also show how one can in this way obtain classical computational problems as defined in Example 39.

Often, it is important that a computational problem is formulated relative to some data given in the input and target types. This means that these data are not changed during the computation. This is for example the case in examples 36 / 43 and 44: Here the underlying field is fixed. This leads to the following definition.

Definition 47 Let $\boldsymbol{I}$ and $\boldsymbol{T}$ be two complex types which restrict to the same complex type $\boldsymbol{B}$, and let $\mathcal{P}$ be a computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$. If now for each object complex a of type $\boldsymbol{I}$ we have $\{\mathrm{res}^{\boldsymbol{I}}_{\boldsymbol{B}}(a)\} = \mathrm{res}^{\boldsymbol{T}}_{\boldsymbol{B}}(\mathcal{P}(a))$, we say that the problem $\mathcal{P}$ is *relative* to $\boldsymbol{B}$. (See Definition 23 for the definition of a *restriction* of an object.)

Example 48 Let $\boldsymbol{B}$ be the category whose objects are fields and whose morphisms are isomorphisms of fields. As already mentioned, in examples 36 / 43 and 44 computational problems with respect to $\boldsymbol{B}$ are given.

Let us consider another example to illustrate the importance of the notion of relativity of computational problems.

Example 49 An intuitive description of the problem is: Given two rational points on an elliptic curve over some field, compute their sum.

The definition of the types is: We first consider the first order type of fields. Over this we consider the type of elliptic curves over fields. The morphisms from E/k to E'/k' are: Tuples of an isomorphism of fields $k \longrightarrow k'$ and an isomorphism of abstract group schemes $E \longrightarrow E'$ such that the diagram

$$\begin{array}{ccc} E & \longrightarrow & E' \\ \downarrow & & \downarrow \\ \mathrm{Spec}(k) & \longrightarrow & \mathrm{Spec}(k') \end{array} ,$$

where the rows are given by the isomorphisms, commutes.

The type $\boldsymbol{T}$ is now the type of pointed elliptic curves. The objects are thus given by $(k; E; P)$ with $P \in E(k)$.

The type $\boldsymbol{I}$ is defined similarly. However, here the underlying graph has a Y-shape: there are two top vertices. The objects are double-pointed elliptic curves.

Note that according to our general notation, objects of type $\boldsymbol{T}$ are denoted by $(k; E; P)$ and objects of type $\boldsymbol{I}$ are denoted by $(k; E; (P_1, P_2))$. The computational problem is then given by $(k; E; (P_1, P_2)) \mapsto (k; E; P_1 + P_2)$.

3 Abstract algorithms

3.1 Classical RAM models

A classical computational model is the model of the random access machine (RAM). An intuitive description is: There are registers R_0,R_1,R_2,... which can store non-negative integers and there is a program Π.[2] Upon input of some non-negative integer the program modifies the contents of the registers. An important feature is hereby indirect addressing. We remark here that the "registers" are just introduced to fix the intuition. By definition a RAM just consists of a program. Whereas intuitively, the execution of one command

[2] In some RAM-models there is in addition an accumulator. Here we take models without accumulator as a starting point.

of a RAM is given by a modification of the contents of "registers", formally it is given by an operation on $\mathbb{N}_0^{\mathbb{N}_0}$.

There are different RAM models with different sets of commands. In fact, one might introduce any function from $\mathbb{N}_0^n$ to $\mathbb{N}_0$ as a command. Of course, the introduction of such a function can have dramatic consequences, and it might even be that in a particular such model functions are computable which are not computable in the usual sense. There is however no reason for not studying computations in dramatically different models.

Another aspect concerns complexity. A change of the complexity measure can have dramatic differences. A telling example is the following one, which was proven in [Sch79] (see also Theorems 20.12 and 20.35 in [WW86]): If one considers a RAM with commands for addition and multiplication and the so-called uniform complexity measure, one obtains a model for which the set of problems which can be decided in polynomial time is equal to the set of problems which can be decided in polynomial space on a Turing machine or on the same RAM machine with logarithmic complexity measure.

3.2 The storage of objects

Taking the RAM model as a starting point, our idea is to allow machines with registers in which objects of an arbitrary elementary type can be stored. More precisely, in each register one can either store a first order object or an object with the help of objects of lower order.

In this subsection we describe how objects are stored. In the subsequent subsections we then give the definition of *abstract algorithms* and define what states are and how an abstract algorithm operates on states.

We fix for the following a complex type $\boldsymbol{C}$ with a fixed vertex v_0 which is at the bottom of $V_{\boldsymbol{C}}$ such that the associated type is $\boldsymbol{N}_0$.

We fix a natural number d which we call the *dimension of the storage*. We require that d is at least 2 and that $d \geq d_{\boldsymbol{C},v}$ for all vertices v of $\boldsymbol{C}$. Furthermore, we fix a so-called *input type* $\boldsymbol{I}$, which is a complex type from which $\boldsymbol{C}$ is derived.

The intuitive idea is now that the machine is initialized with the following storage: There is a d-dimensional storage; the corresponding registers are denoted by $\mathrm{R}_{\underline{i}}$ with $\underline{i} \in \mathbb{N}_0^d$. In these registers, called *computation registers*, objects of an arbitrary type from which $\boldsymbol{C}$ is derived can be stored as follows: First, in each of the registers one can store an object of a first order type from which $\boldsymbol{C}$ is derived. Second, for each elementary type $\boldsymbol{A}$ of order at least two from which $\boldsymbol{C}$ is derived and for each b of type $\mathrm{Base}(\boldsymbol{A})$, one can store in each of the registers an object of type $\boldsymbol{A}_b$ – provided that b is also stored and there are appropriate pointers to b. This then also leads to the possibility to store objects of an arbitrary type from which $\boldsymbol{C}$ is derived.

Furthermore, for each vertex v of $V_{\boldsymbol{I}}$, there is a $d_{\boldsymbol{I},v}$-dimensional storage whose registers are denoted by $\mathrm{I}_{v,\underline{i}}$ with $\underline{i} \in \mathbb{N}_0^{d_{\boldsymbol{I},v}}$. These registers are called *input registers*. We note here that the storage $\mathrm{I}_{v,\bullet}$ might be 0-dimensional, that is, contain a single element. Following the notation of () for the empty tuple, we then denote the unique register by $\mathrm{I}_{v,()}$.

We now describe how one can store objects of any (elementary) type from which $\boldsymbol{C}$ is derived. For this, we proceed by induction on the order of the types. For the induction, we need to not only consider objects and elementary types but also object complexes and complex types.

In the following description, whenever we talk about *types*, we consider types up to strong isomorphism. This means that we identify essentially equal types.

First, in each of the registers $\mathrm{R}_{\underline{i}}$ one can store a non-negative integer. Moreover, in any register $\mathrm{R}_{\underline{i}}$ for which no entry of $\underline{i}$ is 0, one can store in $\mathrm{R}_{\underline{i}}$ any object of a first order elementary type from which $\boldsymbol{C}$ is derived.

This gives the following method to store arrays of such objects: Let $\boldsymbol{B}$ be a complex first order type from which $\boldsymbol{C}$ is derived with just one edge v and the corresponding label $(\boldsymbol{B}_v, d_{\boldsymbol{B},v})$, where $\boldsymbol{A} = \boldsymbol{B}_v$. We set $d_{\boldsymbol{B}} := d_{\boldsymbol{B},v}$. An object from $\boldsymbol{B}$ is thus a $d_{\boldsymbol{B}}$-dimensional array of objects of $\boldsymbol{A}$. Assume first that the dimension is 1, and let a 1-dimensional array a of length l of objects of type $\boldsymbol{A}$ be given. Then for any tuple $\underline{i}_0 \in \mathbb{N}_0^{d-1}$, one can store a in the registers $\mathrm{R}_{\underline{i}_0 1}, \ldots, \mathrm{R}_{\underline{i}_0 \ell}$ as follows: The length l is stored in the register $\mathrm{R}_{\underline{i}_0 0}$, and in the registers $\mathrm{R}_{\underline{i}_0 1}, \ldots, \mathrm{R}_{\underline{i}_0 \ell}$ the entries of the array a are stored. If now $d_{\boldsymbol{B}}$ is 2 and a tuple $\underline{i}_0 \in \mathbb{N}_0^{d-2}$ is given, we interpret the array as a 1-dimensional array of arrays and store it analogously. This means that first, in $\mathrm{R}_{\underline{i}_0 00}$ we store the length of the array of arrays. If this is ℓ then in the registers $\mathrm{R}_{\underline{i}_0 1\bullet}, \ldots, \mathrm{R}_{\underline{i}_0 \ell\bullet}$ we store the 1-dimensional arrays. We proceed like this inductively. We stress that the registers whose index contains a 0 are solely used for the purpose of storing lengths of arrays as described.

We denote registers used to store arrays as described by $\mathrm{R}_{\underline{i}_0\bullet}$.

The storage of objects in the input registers is more restrictive: We postulate that for each vertex v at the bottom of the graph $V_{\boldsymbol{I}}$, in each of the registers $\mathrm{I}_{v,\underline{i}}$ one can store an object of type $\boldsymbol{C}_{\downarrow v}$ or of type $\boldsymbol{N}_0$, depending on whether $\underline{i}$ does not or does contain 0, where the same rules as above apply.

The previous obviously also gives a way to store object complexes whose type is of first order from which $\boldsymbol{C}$ can be derived. After all, such a complex is nothing but a collection of objects whose type is of first order.

Now, let an integer $e \geq 2$ be given, and let first $\boldsymbol{A} = \boldsymbol{C}_{\downarrow v}$ be an e^{th} order elementary type defined by a vertex v of $\boldsymbol{C}$. Recall that every object a of type $\boldsymbol{A}$ is of the form $(b; c)$, where b is an object complex of type $\mathrm{Base}(\boldsymbol{A})$

and $c \in \boldsymbol{A}_b$. Let such an object be given. The object complex b can be stored in registers $\mathrm{R}_{\underline{i}}$ and $\mathrm{I}_{v,\underline{i}}$. Let us assume that the object complex is stored in such registers. Then a can be stored by additionally storing in any register $\mathrm{R}_{\underline{i}}$ or $\mathrm{I}_{v,\underline{i}}$ the object c of type $\boldsymbol{A}_b$ as well as a tuple of *pointers* – one pointer w for each edge vw starting at the top vertex v of $\boldsymbol{A}$ and heading to the top registers used to store b. Precisely, let an edge vw be given, and let us assume that the registers $\mathrm{R}_{\underline{i}_0\bullet}$ contain the data for the corresponding vertex w. Then the pointer for this edge is $\underline{i}_0$. Similarly, if the registers for w are $\mathrm{I}_{w,\underline{i}_0\bullet}$, the edge defining the pointer is $w\underline{i}_0$.

In an obvious way one can then also store object complexes whose type is of e^{th} order and such that $\boldsymbol{C}$ is derived from it. Let such an object complex a of type $\boldsymbol{C}$ be given. By deleting the vertices in the e^{th} layer, we obtain the full-dimensional type $\boldsymbol{B}$ of order $e-1$ (see Definition 24). The object complex a restricts to an object complex b of type $\boldsymbol{B}$. Let us assume that b has been stored as described. Then for each vertex in the e^{th} layer we proceed as described in the previous paragraph and in the above description for the storage of arrays.

As an example to illustrate the storage of elementary types of second order one can consider Example 28: Here one first has to store a group G. Given G, the fiber over G is canonically isomorphic to the set of elements of G. A whole object is then additionally stored by storing the group element and a reference to the place of storage of the corresponding group.

Note also that if an elementary type $\boldsymbol{A}$ is trivially fibered with fiber $\boldsymbol{F}$ then one can store an object a by first storing $\mathrm{Base}(a)$ and second an element from $\boldsymbol{F}$.

We now define what we mean by a possible *content* of the registers.

Definition 50 A *content* is a map c from the set of registers (of both kinds) (or more formally, of the union of the sets of the corresponding addresses) such that the following holds:

There are finite sets of registers $S_1, \ldots, S_k$ such that: First, outside the set $\bigcup_{i=1}^{k} S_i$ the map c has the constant value of 0 as an object of type $\boldsymbol{N}_0$. Second, the restriction of the map c to each of the sets S_i defines the storage of an object complex as described above. (This means in particular that any register in S_i whose index contains a 0 is used to describe the length of a tuple in an array – as described above.)

Notation 51 Such a content is denoted by $(I, R) = ((I_{v,\underline{i}})_{v,\underline{i}}, (R_{\underline{i}})_{\underline{i}})$.

We obtain in this way the set of *contents of the registers*.

Discussion

One might define registers of different (elementary) types. Objects should then only be stored in a register of an appropriate type. Here it would be reasonable to proceed as follows: Let $\boldsymbol{\mathcal{E}}$ be the set classes of elementary types of $\boldsymbol{C}$ up to strong isomorphism. Now for every class $[\boldsymbol{A}]$ in $\boldsymbol{\mathcal{E}}$ we introduce corresponding registers $\mathrm{R}_{\boldsymbol{A},\underline{i}}$. This means that for two elementary types $\boldsymbol{A}$ and $\boldsymbol{B}$ which are essentially equal, the registers $\mathrm{R}_{\boldsymbol{A},\underline{i}}$ and $\mathrm{R}_{\boldsymbol{B},\underline{i}}$ are by definition equal.

A particular role would then play the registers $\mathrm{R}_{\mathbf{N}_0,\underline{i}}$: These would be the only registers allowed to serve as address registers for indirect addressing (see below).

We have decided against this possibility for two reasons: First, it makes no essential difference because whenever an object is stored, the corresponding type is stored anyway because the type is part of the object. Second, the potential definition would cause some additional technical problems in the discussion on abstract algorithms and representations in subsection 4.3.

3.3 Algorithms

We continue with the above setting, that is, with a complex type $\boldsymbol{C}$, some *dimension* $d \geq 2$ and *input type* $\boldsymbol{I}$ – with the restrictions outlined in the beginning of the previous subsection. We now want to define what we mean by an *abstract algorithm for a machine of type* $\boldsymbol{C}$ *and of dimension* d. For simplicity, we are going to refer to an abstract algorithm as we are going to define it simply as an *algorithm*.

For this, we first fix a so-called *output type* $\boldsymbol{T}$ from which $\boldsymbol{C}$ is derived and a *base type* $\boldsymbol{B}$ to which both $\boldsymbol{I}$ and $\boldsymbol{T}$ restrict. As stated above, the machine then has *computation registers* $\mathrm{R}_{\underline{i}}$ for $i \in \mathbb{N}_0^d$ and *input registers* $\mathrm{I}_{v,\underline{i}}$ for $v \in V_{\boldsymbol{I}}$ and $\underline{i} \in \mathbb{N}_0^{d_{\boldsymbol{I},v}}$.

As the name suggests, an object complex $a = (a_v)_{v \in V_{\boldsymbol{I}}}$ of type $\boldsymbol{I}$ input to the algorithm is stored in the input registers: a_v is stored in registers $\mathrm{I}_{v,\bullet}$. The input registers $\mathrm{I}_{v,\bullet}$ with $v \in V_{\boldsymbol{B}}$ are read-only.

Concerning the other input registers, there is are two possibilities which are called **preserving** and **updating** which must be specified at the beginning of each algorithm. If the option **preserving** is set, the input registers are read-only registers. If the option **updating** is set, only the input registers for edges in $\boldsymbol{B}$ are read-only, the other input registers are then read-and-write registers.

A guiding idea is that just as there is no a priori restriction on types, there should also no a priori restriction on the commands which modify the registers. This leads to the radical proposal to allow *any relative computa-*

tional problem itself to serve as the basis of a command – provided that its input and output types fit to the given types $\boldsymbol{C}$ and $\boldsymbol{B}$ of the algorithm.

Here we employ analogous ideas to the ones described concerning the input-output structure of an algorithm: For each complex types $\boldsymbol{X}$ and $\boldsymbol{Y}$ from which $\boldsymbol{C}$ is derived and each complex type $\boldsymbol{Z}$ to which both $\boldsymbol{X}$ and $\boldsymbol{Y}$ restrict, we have two kinds of commands: *preserving* and *updating* commands: *Preserving* commands are based on computational problems from $\boldsymbol{X}$ to $\boldsymbol{Y}$ relative to $\boldsymbol{Z}$. *Updating* commands are based on computational problems from $\boldsymbol{X}$ to $\boldsymbol{X} \times_{\boldsymbol{Z}} \boldsymbol{Y}$ relative to $\boldsymbol{Z}$. (See Definition 25 for the definition of $\boldsymbol{X} \times_{\boldsymbol{Z}} \boldsymbol{Y}$.) A command is then based on such a computational problem and addresses for the *input type* $\boldsymbol{X}$ and the *output type* $\boldsymbol{Y}$.

We thereby allow indirect addressing, which means that we read the contents of registers, and if these contents are non-negative integer, we interpret these as addresses of registers whose content we then read.

Preserving commands are called like this because the input is not directly affected by the application of a command. (It might however be affected because it is overwritten due to the addresses used.) By contract, an application of an updating command might modify the input. One aspect is of importance here: The output always might override the input, also for preserving commands. For an updating command it might happen that the output is modified and also overwritten. We postulate here that the overriding by the output is dominant. This corresponds to the intuition that the overriding takes place internally in the command and finally the commands outputs some data which are then stored.

Furthermore, we have a branching command which relies on the test whether a particular register for non-negative integers contains 0, and we have a command for output.

We now describe the commands of an algorithm based on a machine of type $\boldsymbol{C}$ and dimension d with input type $\boldsymbol{I}$ and output type $\boldsymbol{T}$.

Definition 52 Let $\boldsymbol{A}$ be any complex type from which $\boldsymbol{C}$ is derived. As usual, let $V_{\boldsymbol{A}}$ be the set of vertices of $\boldsymbol{A}$. Furthermore, let {d,i} and {R,I} be sets with each two elements. Then a *register assignment* for $\boldsymbol{A}$ consists of, first of all $\boldsymbol{A}$ itself, second a subset V of $V_{\boldsymbol{A}}$ containing the top vertices of $\boldsymbol{A}$ such that each vertex of $\boldsymbol{A}$ not in V is the end of exactly one vertex and third an assignment from V which assigns to each vertex of V one of:

- a tuple $(\mathrm{d},\mathrm{R},\underline{i})$, where $\underline{i}$ is a possibly empty tuple of non-negative integers of length at most $d - d_{\boldsymbol{A},v}$,
- a tuple $(\mathrm{i},\mathrm{R},\underline{i})$, where $\underline{i}$ is a possibly empty tuple of non-negative integers of length $d - 1$,

- if v is a vertex of $\boldsymbol{I}$: a tuple (d,I,$\underline{i}$), where $\underline{i}$ is a possibly empty tuple of non-negative integers of length at most $d_{\boldsymbol{I},v} - d_{\boldsymbol{A},v}$,
- if v is a vertex of $\boldsymbol{I}$: a tuple (i,I,$\underline{i}$), where $\underline{i}$ is a possibly empty tuple of non-negative integers of length $d-1$.

and the following condition is satisfied:

- Whenever for some path vw the vertex v is assigned to a tuple of the form (d,I,$\underline{i}$) or (i,I,$\underline{i}$) then w is also assigned to such a tuple.
- If at a vertex v of V starts an edge which does not end in V and v is assigned to (d,R,$\underline{i}$) then the length of $\underline{i}$ is at most $d - d_{\boldsymbol{A},v} - 1$.

Definition 53 The *input assignment* is the register assignment given by mapping a vertex v of $\boldsymbol{I}$ to (d,I,()). Recall here that by () we denote the empty tuple.

For any vertex v of $V_{\boldsymbol{A}}$, the interpretation of a tuple the form (d,...) is that we give *directly* an array of the appropriate type and dimension, the interpretation of a tuple of the form (i,...) is that we do so *indirectly*. Let $\boldsymbol{A} := \boldsymbol{A}_v$. A tuple (d,R,$\underline{i}$) as described means that we point to the registers $\mathrm{R}_{1\cdots 1\underline{i}\bullet}$, where the number of $1'$s is appropriate. Likewise, a tuple (d,I,$\underline{i}$) means that we point to the registers $\mathrm{I}_{v,1\cdots 1\underline{i}\bullet}$. We then call such a register the *data register* (for the given assignment and v). A tuple of the form (i,R,$\underline{i}$) means that we address registers indirectly. Concretely, indirect addressing for such a tuple is as follows: Suppose that the register $R_{\underline{i}0}$ contains a non-negative integer (an object of type $\boldsymbol{N}_0$) l with $1 \leq l \leq d - d_{\boldsymbol{A},v}$ and that the registers $R_{\underline{i}1}, \ldots, R_{\underline{i}l}$ contain non-negative integers $r_1, \ldots, r_l \geq 1$. Now, these integers give the addresses of a $d_{\boldsymbol{A},v}$-dimensional array $R_{1\cdots 1r_1\cdots r_l\bullet}$ or $I_{1\cdots 1r_1\cdots r_l\bullet}$. (If for indirect addressing the given conditions on the content of the registers are not met, an algorithm using this indirect addressing will fail.) We call the registers $R_{\underline{i}\bullet}$ the *address registers* and the registers pointed to by the content of these registers again the *data registers*.

The reader might now ask her- or himself why we allow addresses which are "too small" then pad the address with $1'$s in front. There are two reason for this: The first reason is that we want to by able to derive addresses for vertices not in V in an automated way. We explain this in the following definitions. The second reason is as follows: We want to be able to interpret an algorithm on a machine of type $\boldsymbol{C}$ and for a larger dimension than the given one d. Both these reasons will become important when we consider representations of types and their applications to algorithms in subsection 4.3.

Notation 54 If a is a register assignment and v a vertex, we denote the corresponding tuple by a_v.

Definition 55 A *direct register assignment* is a register assignment in which only tuples of the form $(\mathrm{d}, \ldots)$ occur. If the set V is equal to $V_{\boldsymbol{A}}$ we speak of a *full direct register assignment.*

Definition 56 To any register assignment and content as described above, we try to associate the obvious corresponding direct register assignment by evaluating the addresses for indirect addressing – if possible. If this is not possible, we say that the register assignment *does not define a direct register assignment.*

Let now a *direct* register assignment be given. We extend the direct register assignment as follows to registers which are not in V, obtaining what we call the corresponding *full direct register assignment.*

Note first that by assumption at each edge w not in V there is a unique edge ending at w. Furthermore V contains the top vertices. It follows that for each such vertex w there is a (unique) smallest path starting in V and ending in w.

Let v be a vertex of V such that there is an edge vw with $w \notin V$.

Let us first suppose that to v we associate via a given register assignment and content the input register $\mathrm{I}_{v,\underline{i}}$ as data register. Let vw be an edge, where w is not in V. Then we associate to w the register $\mathrm{I}_{w,\underline{i}}$ as data register. We continue like this by induction.

Let us now suppose that to v we associate some computation register. Note that by definition of a register assignment, the address has the form $1 \cdots 1\underline{i}$ with a non-trivial number of $1'$s at the beginning. Now we have to define an ordering on the edges not in V "below" w. For this, we first order all vertices of $\boldsymbol{A}$ via the given orderings of the edges at the vertices and a breadth-first search. (The bottom vertices come first.) This ordering induces an ordering on the edges starting at some vertex, for example v. We then associate to the j^{th} edge at v the register $\mathrm{R}_{j1\cdots 1\underline{i}}$.

Definition 57 Let now $\boldsymbol{C}$ be a complex type which restricts to complex types $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ which in turn restrict to a complex type $\boldsymbol{B}$. Furthermore, let a_1, a_2 be register assignments for $\boldsymbol{A}_1, \boldsymbol{A}_2$ respectively. We now say that a_1 and a_2 are *consistent* with respect to $\boldsymbol{B}$ if the domains of both maps contain the top vertices of $\boldsymbol{B}$ and if on the set vertices of $\boldsymbol{B}$ on which both are defined they define the same map.

Now the *commands* are:

- Two commands called **preserving** and **updating**.
- For each two complex types $\boldsymbol{X}$ and $\boldsymbol{Y}$ from which $\boldsymbol{C}$ is derived and

each complex type $\boldsymbol{Z}$ to which both $\boldsymbol{X}$ and $\boldsymbol{Y}$ restrict,[3] for each computational problem $\mathcal{P}$ from $\boldsymbol{X}$ to $\boldsymbol{Y}$ relative to $\boldsymbol{Z}$, and for each register assignments a_0 of $\boldsymbol{X}$ and a_1 of $\boldsymbol{Y}$ which are consistent relative to $\boldsymbol{Z}$, a command

let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$.

(This command is called *preserving*.)

- For each two complex types $\boldsymbol{X}$ and $\boldsymbol{Y}$ from which $\boldsymbol{C}$ is derived and each complex type $\boldsymbol{Z}$ to which both $\boldsymbol{X}$ and $\boldsymbol{Y}$ restrict, for each computational problem $\mathcal{P}$ from $\boldsymbol{X}$ to $\boldsymbol{X} \times_{\boldsymbol{Z}} \boldsymbol{Y}$ relative to $\boldsymbol{Z}$, and for each register assignments a_0 of $\boldsymbol{X}$ and a_1 of $\boldsymbol{Y}$ which are consistent relative to $\boldsymbol{Z}$, a command

  let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$ **with update** .

  (This command is called *updating*.)

- For each elementary type $\boldsymbol{A}$ such that for all $a \in \mathrm{Base}(\boldsymbol{A})$ the fiber $\boldsymbol{A}_a$ is a set and each register assignments a_0 of $\mathrm{Base}(\boldsymbol{A})$ and a_1 of $\boldsymbol{A}$ such that a_1 restricts to a_0 (which is equivalent to the condition that a_0 and a_1 be consistent relative to $\mathrm{Base}(\boldsymbol{A})$), a command

  choose a_1 **relative to** a_0 .

- For each tuple of non-negative integers $\underline{i}$ of length at most d and each natural number c a command

  if $\mathrm{R}_{1\cdots 1\underline{i}} \neq 0$ **then goto** c .

- For each register assignment a of $\boldsymbol{T}$ which is consistent with the input assignment relative to $\boldsymbol{B}$ a command

  output a .

The meaning of the commands should be clear from their names. The reader might ask him- or herself what is the difference between the commands **let** and **choose**. A first answer is that a **choose**-command is nothing but a special case of a **let**-command. However, **choose**-commands are seen as a source of randomization. In contrast, **let**-commands are a priori non-deterministic.

[3] See Definition 23 for the definition of "derived" and "restricts to"; note that the second notion is stronger than the first one.

Definition 58 For a complex type $\boldsymbol{C}$, two types $\boldsymbol{I}$ and $\boldsymbol{T}$ from which $\boldsymbol{C}$ is derived, and a number d as above, an *abstract algorithm for a machine of type $\boldsymbol{C}$ and of dimension d with input type $\boldsymbol{I}$ and output type $\boldsymbol{T}$* is a finite sequence of the above commands with the following restrictions: (We say that the i^{th} command occurs in the i^{th} line.)

- The command in the first line is either **preserving** or **updating**, and these commands only occur in the first line.
- The command in the last line of the algorithm is of the form **if** ... or **output** ....
- Each integer c occurring in the **if** ... commands is at most the largest line number, that is, the length of the sequence.

Terminology 59 As stated already at the beginning of this subsection and as done throughout the subsection, we continue by referring to abstracts algorithms simply as *algorithms*.

The **choose**-commands should be seen as a source of randomization in the algorithm. We thus define:

Definition 60 An algorithm is *non-randomized* if does not contain a **choose**-command.

Furthermore, we define:

Definition 61 An algorithm is *functorial* if all **let**-commands commands are based on functorial computational problems. (Such a **let**-command is itself called *functorial*).

This definition is important for the following reason: If there is a **choose**-command we want to consider the computation from a probabilistic point of view. However, as already stated above the definition, a **let**-command is a priori a non-deterministic command. As it does not seem to be reasonable to have non-determinism and randomization in the same algorithm, if there is a **choose**-command it is reasonable to demand that the **let**-commands be functorial.

Discussion

We make some remarks here, discussing the intuition surrounding these definitions and potential alternatives.

- We would like to stress that the commands are really mathematical objects. We write **let** etc. merely to refer to these objects. An algorithm is then a tuple of such objects.

- Instead of fixing the types $\boldsymbol{C}, \boldsymbol{I}, \boldsymbol{T}$ and the dimension d in advance, we could have introduced commands for the initialization of the machine like

  type $\boldsymbol{C}$

  for any complex type $\boldsymbol{C}$ which is derived from the type $\mathbb{N}_0$,

  dimension $\boldsymbol{A}, d$

  for any elementary type $\boldsymbol{A}$ and any $d \in \mathbb{N}$,

  input $\boldsymbol{I}$

  for any type from which $\boldsymbol{C}$ is derived, and instead of the output commands **output** a, we could have introduced a command

  output $\boldsymbol{T}, a$

  for every type $\boldsymbol{T}$ from which $\boldsymbol{C}$ is derived and and object a of type $\boldsymbol{T}$. Every algorithm would then start with the first three commands, and instead of the output command there would now be commands **output** $\boldsymbol{T}, a$ for a single type $\boldsymbol{T}$. The other rules would be as stated above.

- As complex types can be empty, in particular the complex types $\boldsymbol{Z}$ in the **let ...** commands above can be empty. This can be interpreted as an "absolute" rather than a "relative" computation. One might ask why we have introduced the relative computations at all as maybe it seems unnecessary to specify that something is fixed. The answer is that this corresponds to the usual intuition of computation, is relevant for complexity theoretic considerations and is relevant when we introduce representations of types by other types.

- Alternatively to the definition of an algorithm via commands, we could also define an algorithm as a finite labeled directed graph with certain properties. Among these properties are: Every vertex has at most one ingoing edge and at most two outgoing edges. The input command then corresponds to vertex with just an outgoing edge, and the output commands correspond to vertices with just one ingoing edge.

- All **let**-commands (preserving or updating) can be seen as operating by call-by-reference. The essential difference is that with the preserving commands the input data are guaranteed to be fixed whereas with the updating commands it might be changed as required by the computational problem.

- Even though all objects have types, the registers (or variables) themselves are type-less. This means that objects of any type can be stored in any registers. This could be changed. We have refrained from giving types to registers in order to lower the complexity of the description.

3.4 States and operations on states

We define the set of *states* of an algorithm:

Definition 62 A *state* is a tuple $(l, (I, R), o)$ where

- l is a natural number which is at most equal to the length of the algorithm called the *current line number*,
- (I, R) is a content of the registers, defined in Definition 50,
- o is either an object from the output type $\boldsymbol{T}$ or one of two other objects called "running" and "failure". (In the first case, we say that the algorithm has *terminated* with output o.)

Now for each command we would like to describe how it operates on the set of states. Before we come to this, we have to address however two potential problems concerning register assignments used in the commands: an assignment to read data might be inconsistent with the content of the registers and an assignment to write data might not lead to what we called content; the result might not even be interpretable anymore. We first address these problems.

Definition 63 Let (I, R) a content of the registers (see Definition 50) be fixed.

- Let a be a register assignment (see Definition 52) of some type $\boldsymbol{X}$ from which $\boldsymbol{C}$ is derived. We say that a is *consistent with the content for reading* if: First, a defines a direct register assignment as indicated in Definition 56. Second, the full direct register assignment associated to a has this property: The addresses given point to arrays and the data in these arrays define an object of type $\boldsymbol{X}$. Third, for each edge vw of $\boldsymbol{X}$ and each register in which some object in the array for v is stored, the pointer for vertex w points to the appropriate subarray of the array given by w in the register assignment.
- Let $\boldsymbol{W}$ be a type from which $\boldsymbol{C}$ is derived. Let a be a register assignment for $\boldsymbol{W}$ and b an object of type $\boldsymbol{W}$. We say that a is *consistent with the content for writing* if the following holds: First, a defines a direct register assignment. Second, the full direct register assignment

associated to a has this property: If the algorithm is preserving (that is, the first command of the algorithm is **preserving**), there is no vertex $v \in V_{\boldsymbol{W}}$ for which the register given by the register assignment is $\mathrm{I}_{v,\underline{i}}$ for some $\underline{i}$. If the algorithm is updating, there is no vertex $v \in V_{\boldsymbol{W}} \cap V_{\boldsymbol{B}}$ with this property. Third, when we change the content by writing b as indicated by the register assignment, we again obtain a content.

Definition 64 Let again a content (I, R) be fixed.

- A command

  let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$

  is consistent with the content if a_0 is consistent with the content for reading and a_1 is consistent with the content for writing.

- A command

  let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$ **with update**

  is consistent with the content if a_0 is consistent with the content for reading and a_0 as well as a_1 are consistent with the content for writing.

- A command

  choose a_1 **relative to** a_0 .

  is consistent with the content if a_0 is consistent with the content for reading and a_0 is consistent with the content for writing.

- A command

  if $\mathrm{R}_{\underline{i}} \neq 0$ **then goto** c

  is consistent with the content if the register $\mathrm{R}_{\underline{i}}$ contains an object of type $\boldsymbol{N}_0$.

- A command

  output a

  is consistent with the content if a is consistent with the content for reading.

Remark and Definition 65 Now, given a state and a command as described, there are two possibilities:

Either the storage assignment is consistent with the state. In this case we obtain in an obvious way a class of states be applying the command to

the state. Note here that as stated above in terms of the operation defined here, a **choose**-command is a special case of a **let**-command.

Or the register assignment is inconsistent with the state. In this case we define that the result is the state "failure".

We can therefore apply an algorithm to an input instance. If at every **let**-command and every **choose**-command we choose one possibility from the class obtained by applying the computational problem to the corresponding contents, we obtain a *complete computation path* $(l_t, (I, R)_t, o_t)_{t\in\mathbb{N}_0}$. Here the index t is called the *uniform time* or simply *time* if no confusion with *time complexity*, which will be discussed in the next subsection, is possible. If we first fix a *time bound* t_0, we obtain a *computation path up to (uniform) time* t_0. We say that the computation *fails* for the given input instance if one complete computation path leads to "failure". Otherwise, for a preserving algorithm, we consider the class of all outputs for all terminated states, referred to by the class of *possible outputs*. For updating algorithms, for every terminated state we consider all objects of type $\boldsymbol{I} \times_{\boldsymbol{B}} \boldsymbol{T}$ given by the objects in the input registers and the outputs. Again we consider the class of these objects called the class of *possible outputs with input modification*.

We now want to define what it means that an algorithm computes a computational problem. For this, we consider only non-randomized and functorial algorithms.

Definition 66 Let a computational problem $\mathcal{P}$ from $\boldsymbol{I}$ to $\boldsymbol{T}$ over $\boldsymbol{B}$ and a non-randomized algorithm with input type $\boldsymbol{I}$, output type $\boldsymbol{T}$ and base type $\boldsymbol{B}$ be given. If now for every object a of type $\boldsymbol{I}$ for no computation path the algorithm fails and the class of possible outputs or the class of possible outputs with input modification (depending on the mode of the algorithm) is non-empty and contained in $\mathcal{P}(a)$ we say that the algorithm *computes* $\mathcal{P}$.

We remark here that above we have said that **let**-commands are potentially non-deterministic. However, in the definition we consider all computation paths. In fact, we are not interested in non-determinism as considered in complexity theory. Rather, we want to allow that not all choices for computation have to be fixed when an algorithm is written down. We refer here again to Example 36 for an illustration. In the same spirit, when we consider the time complexity of algorithms, we will take the maximum of the time (as defined there) over all computation paths.

For functorial algorithms, we can talk about randomized computation. For this, as usual, we make the choices in the **choose**-commands uniformly and independently of each other. We define:

Definition 67 Let a computational problem $\mathcal{P}$ from $\boldsymbol{I}$ to $\boldsymbol{T}$ over $\boldsymbol{B}$ and a functorial algorithm with input type $\boldsymbol{I}$, output type $\boldsymbol{T}$ and base type $\boldsymbol{B}$ be given. If now for every object a of type $\boldsymbol{I}$ for no computation path the algorithm fails, the algorithm terminates with a probability of 1 and the class of possible outputs or the class of possible outputs with input modification (depending on the mode of the algorithm) is contained in $\mathcal{P}(a)$ we say that the algorithm *computes* $\mathcal{P}$.

Discussion

It can happen that in a computation path at two times $t_1 < t_2$ the sates are isomorphic (or even equal) but the following states at the time $t_1 + 1$ and $t_2 + 2$ are not isomorphic. It arises the question if this is reasonable. Or should this be ruled out?

Our answer is that this should not be ruled out or at least not a priori. The reason for this is related to the concepts of representation and specification discussed in Section 4 below: A **let**- or a **choose**-command might be implemented by another algorithm which does not take objects of the given type as input but rather objects which are representatives of the given objects. Now it is possible that one object has different non-isomorphic representatives, and we do at this point not know how the algorithm implementing the command operates with these representatives.

3.5 Complexity

We now consider the idea of *time complexity* of a computation, where by *complexity*. At this point, it is reasonable to analyze a particular algorithm with complexity measures which reflect the abstract nature of the objects involved in the computations. So, for example, one might say that on a particular input an algorithm need so-and-so many group operations or field operations or operations in an elliptic curve.

It is however reasonable to distinguish between what might be called *addressing costs* and *computation costs*. The addressing costs are the costs intuitively follow from the fact that data has to be fetched at different parts of the storage before computations can be carried out. These costs have the unit of numbers,[4] in contrast to the computation costs which might have the unit of addition in a group or so.

In this general situation not too much can be reasonably said on how one should measure these costs. For example, one might overestimate the addressing costs by considering all addresses which are involved in the input data because not all these data might actually be used in the computation.

[4]We use the word *unit* here as in physics.

We now fix, until the end of the subsection, an algorithm $\boldsymbol{\mathcal{A}}$ for a machine of type $\boldsymbol{C}$.

So we just assume that to every state and every command in a given algorithm some *complexity measure* is associated. We then can define the *time complexity* of a particular computation path up to some uniform time t by adding the costs for the computations. We stress here that – as indicated above – for a particular algorithm it might be reasonable to consider different complexity measures with different units.

In the study of algorithms it is usual to denote functions on the input instances just as evaluations of the functions on the said instances. We follow this tradition:

Notation 68 We often denote functors from some first order elementary type to $\boldsymbol{R}_{\geq 0} \dot{\cup} \{\infty\}$ in the same way as evaluations of such functors. This applies in particular to the time complexity of input instances and related assignments.

Moreover, for two functors from some first order elementary type to $\boldsymbol{R}_{\geq 0} \dot{\cup} \{\infty\}$ by $t \leq T$ we mean that for all objects a of the large groupoid we have $t(a) \leq T(a)$.

Definition 69 We say that a non-randomized algorithm *terminates* with a time complexity of $T \in \boldsymbol{R}_{\geq 0} \dot{\cup} \{\infty\}$ on some input instance a if for every computation path for a the time complexity is at most T. Given a functor T from the category of the input type to $\boldsymbol{R}_{\geq 0} \dot{\cup} \{\infty\}$ we say that the algorithm *terminates* with a time complexity of T if for every instance a the algorithm terminates with a time complexity of $T(a)$.

We now discuss the notion of *expected time complexity* for functorial algorithms. By this we want to capture the intuition that there is an *internal randomization* in the algorithm. Randomizations among the input instances shall not be considered here.

Definition 70 Let a functorial algorithm and an input instance a to the algorithm be given. As above, at every **choose**-command we choose one of the finitely many possible objects uniformly independently at random. The time complexity of the algorithm applied to a is then a random variable. In particular, we can talk about the *expected time complexity* of an application of the algorithm to a. This can be finite as well as infinite.

4 Representations and computations with representing objects

4.1 Representations of types

Whenever one wants to describe some computation with mathematical objects which are not bit-strings or non-negative integers via a Turing machine or a RAM, one first has to describe how the objects shall be represented by bit-strings / non-negative integers.

Similarly, for some fixed field k, one might consider computations on register machines which can store elements of both non-negative integers and elements from k. One might then describe how one represents some mathematical objects with elements from k and non-negative integers.

To motivate the following definitions, we now consider some examples which we first describe intuitively and then within the already developed framework of types.

Example 71 Let us fix a (non-separably closed) field k and consider the problem to represent finite separable field extensions in terms of field elements in k and natural numbers. Every such extension is primitive, thus it is isomorphic to an extension of the form $(k[t]/(f(t)))|k$ with $f(t) \in k[t]$ separable and irreducible. Because of this, it is reasonable to say that "every finite separable extension of k can be represented by a polynomial in $k[t]$". In this statement, one might (but need not) add the information that the polynomials in question are separable and irreducible. This situation can be described from a functorial point of view as follows:

Let $\boldsymbol{A}$ be the first order elementary type of finite separable field extensions of k. This is the type of the objects which shall be represented. In addition, we have $\boldsymbol{k[t]}$ which is the type of representing objects (cf. Example 13).

Let S be the set of all separable irreducible polynomials in $k[t]$, and again let $\boldsymbol{S}$ be the corresponding type. Now the assignment $f(t) \mapsto k[t]/(f(t))$ is a functor from $\boldsymbol{S}$ to $\boldsymbol{A}$. The fact that every extension is isomorphic to an extension of the form $k[t]/(f(t))$ can be expressed by saying that this functor is *essentially surjective*.

Example 72 Continuing with the previous situation, one might say that one represents polynomials over k, that is, elements from $k[t]$, by arrays of elements from k. This statement seems to be harmless, but we have to be a bit careful in order to formulate it formally:

First, we have the set $k[t]$, and this set is the type of the objects to be represented. Now, we have $\mathrm{array}^1(k)$. Note that the elements in $\mathrm{array}^1(k)$ are non-trivial tuples of elements of k.

We now consider the subset S of $\mathrm{array}^1(k)$ consisting of the following elements: First, the tuple (0), and second, every tuple which does not contain a 0 at the end (that is, as last entry). We have now an obvious isomorphism from S to $k[t]$. Put differently, we have an isomorphism of categories from $\boldsymbol{S}$ to $\boldsymbol{k}[\boldsymbol{t}]$.

Inspired by these examples, we define:

Definition 73 Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be categories. Then a *partial functor* from $\boldsymbol{X}$ to $\boldsymbol{Y}$ is a functor $\mathcal{F}$ from a full subcategory $\boldsymbol{S}$ of $\boldsymbol{X}$ to $\boldsymbol{Y}$. The category $\boldsymbol{S}$ is then called the *domain* of $\mathcal{F}$ and denoted by $\mathrm{dom}(\mathcal{F})$.

Recall here that a *full subcategory* $\boldsymbol{S}$ of a category $\boldsymbol{X}$ is a subcategory of $\boldsymbol{X}$ such that for all $a, b \in \boldsymbol{S}$, $\mathrm{Mor}_{\boldsymbol{S}}(a, b) = \mathrm{Mor}_{\boldsymbol{X}}(a, b)$.

The idea of "representation" in the above examples is captured with the following definition.

Definition 74 Let $\boldsymbol{A}$ and $\boldsymbol{A'}$ be first order types (that is, large groupoids). Then a *partial representation* of $\boldsymbol{A}$ by $\boldsymbol{A'}$ is a partial functor from $\boldsymbol{A'}$ to $\boldsymbol{A}$. A *representation* is a partial representation which is essentially surjective.

Remark 75 A composition of partial representations is a partial representation and a composition of representations is a representation.

Remark 76 Note that we say, for example, that we represent finite separable k-algebras by polynomials over k, whereas formally, it is the type of finite separable k-algebras which is represented by the type of polynomials over k. Note in addition that laxly, one might also write in an informal description of an algorithm a phrase like this one: "Let a finite separable k-algebra A be given, represented by a polynomial over k." Note here that the algebra is of course still variable, and so is the polynomial. Again, formally, one represents one type by another type.

In order to avoid any confusion, in the definitions below, we will only use the formally correct wording that a type is represented by another type and not the informal wording that objects of a particular type are represented by objects of some other type.

In the above examples, the field k is fixed. If we let k vary, in an obvious way we obtain second order types. Intuitively, we obtain in an obvious way again representations. The corresponding definition is as follows.

Definition 77 Let $\boldsymbol{C}$ and $\boldsymbol{C'}$ be a complex types. A *partial representation* of $\boldsymbol{C}$ by $\boldsymbol{C'}$ consists of

- a subset V of $V_{\boldsymbol{C}'}$ containing the top vertices such that each vertex of $\boldsymbol{C}'$ which is not in V is the endpoint of a unique edge,
- a bijection φ from V to $V_{\boldsymbol{C}}$ which induces a map from the paths between the vertices of V to the paths of $\boldsymbol{C}$ such that for all $v \in V$, $d_{\boldsymbol{C}',v} \geq d_{\boldsymbol{C},\varphi(v)}$,
- a system $(\mathcal{F}_v)_{v\in V}$, where $\mathcal{F}_v$ is a partial functor from $\mathrm{array}^{d_{\boldsymbol{C}',v}-d_{\boldsymbol{C},\varphi(v)}}(\boldsymbol{C}'_v)$ to $\boldsymbol{C}_{\varphi(v)}$, such that for every two vertices v, w of V for which there is a directed path from v to w in $V_{\boldsymbol{C}'}$, $f^{\boldsymbol{C}'}_{vw}$ maps the domain of $\mathcal{F}_v$ to the domain of $\mathcal{F}_w$

such that for each edge ww, when we consider the induced functor $\mathcal{F}_v$ from $\mathrm{array}^{d_{\boldsymbol{C}',v}}(\boldsymbol{C}'_v)$ to $\mathrm{array}^{d_{\boldsymbol{C},\varphi(v)}}(\boldsymbol{C}_{\varphi(v)})$ (which is also denoted by $\mathcal{F}_v$) and the corresponding functor induced by $\mathcal{F}_w$, we have the compatibility relation

$$f^{\boldsymbol{C}}_{\varphi(v)\varphi(w)} \circ \mathcal{F}_v = \mathcal{F}_w \circ f^{\boldsymbol{C}'}_{vw}$$

on the domain of $\mathcal{F}_v$.

Remark 78 A partial representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$ induces a partial representation of $\mathrm{Cat}(\boldsymbol{C})$ by $\mathrm{Cat}(\boldsymbol{C}')$.

Remark and Definition 79 Let $\boldsymbol{C}$ be a complex type over a complex type $\boldsymbol{B}$ and let $\mathcal{F}$ be a partial representation of the complex type $\boldsymbol{C}$ by the complex type $\boldsymbol{C}'$. Then there is a unique type $\boldsymbol{B}'$ such that $\boldsymbol{C}'$ is over $\boldsymbol{B}'$ and the representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$ restricts to a representation of $\boldsymbol{B}$ by $\boldsymbol{B}'$. Here $\boldsymbol{B}'$ is characterized as follows: The defining set of vertices is $\downarrow V \cap V_{\boldsymbol{B}'}$, the map is the restriction of φ to the set $V \cap V_{\boldsymbol{B}'}$, for each vertex v of $V_{\boldsymbol{B}'}$ we have

$$d_{\boldsymbol{B}',v} = d_{\boldsymbol{B},\varphi(v)} + d_{\boldsymbol{C}',v} - d_{\boldsymbol{C},\varphi(v)} = d_{\boldsymbol{C}',v} - (d_{\boldsymbol{C},\varphi(v)} - d_{\boldsymbol{B},\varphi(v)}) \ ,$$

and for each edge vw in $\boldsymbol{B}'$ we have

$$d_{\boldsymbol{B}',w} - d_{\boldsymbol{B}',v} = d_{\boldsymbol{C}',w} - d_{\boldsymbol{C}',v} \ .$$

We say that the given partial representation *restricts* to a partial representation of $\boldsymbol{B}$ by $\boldsymbol{B}'$.

Note that the second set of conditions must be satisfied in order that $\boldsymbol{C}'$ can be over $\boldsymbol{B}'$ (see Definition 23). Here we also need the assumption that at every vertex of $\boldsymbol{C}'$ not in V ends exactly one edge. Otherwise there are easy examples for which is it not possible to satisfy the second set of conditions on the dimensions for all edges vw of $\boldsymbol{B}'$.

Example 80 We consider as $\boldsymbol{C}$ the type of pointed groups and as $\boldsymbol{C}'$ the type of smooth proper curves over fields together with an element in the degree 0 class group. This means that the objects of $\boldsymbol{C}$ are of the form $(G; P)$, where G is a group and $P \in G$, and the objects of $\boldsymbol{C}'$ are of the form $(k; \mathcal{C}; c)$, where k is a field, $\mathcal{C}$ a smooth proper curve over k and $c \in \mathrm{Cl}(\mathcal{C})^0$. We have an obvious partial representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$.

From an intuitive point of view, this example shows that a partial representation can have the role of what might be called "specialization of types".

Remark and Definition 81 Let the elementary type $\boldsymbol{A}$ be partially represented by the complex type $\boldsymbol{C}'$. Then $\boldsymbol{C}'$ is also an elementary type. As stated in Remark 78, the representation induces a partial functor from $\mathrm{Cat}(\boldsymbol{C}')$ to $\mathrm{Cat}(\boldsymbol{A})$. Here, this is equal to the partial functor from $\mathrm{Top}(\boldsymbol{C}')$ to $\mathrm{Top}(\boldsymbol{A})$ which is part of the data of the representation. We say that we have a *representation of the elementary type* $\boldsymbol{A}$ by $\boldsymbol{C}'$ if this partial functor is essentially surjective (that is, defines a representation).

We also consider relative versions of the previous definitions:

Definition 82 Let $\boldsymbol{C}$ and $\boldsymbol{C}'$ be complex types which restrict to complex types $\boldsymbol{B}$, $\boldsymbol{B}'$, respectively. Then a *partial representation of* $\boldsymbol{C}$ *over* $\boldsymbol{B}$ *by* $\boldsymbol{C}'$ *over* $\boldsymbol{B}'$ is a partial representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$ which restricts to a partial representation of $\boldsymbol{B}$ by $\boldsymbol{B}'$. If moreover $\boldsymbol{B}$ is equal to $\boldsymbol{B}'$ and the restricted representation is the identity, we speak of a *partial representation of* $\boldsymbol{C}$ *over* $\boldsymbol{B}$.

Definition 83 Let $f : \boldsymbol{A} \longrightarrow \boldsymbol{B}$ be a functor of large groupoids (defining a second order type), and let $f' : \boldsymbol{A}' \longrightarrow \boldsymbol{B}'$ be such a functor. Moreover, let a partial representation of $\boldsymbol{A} \longrightarrow \boldsymbol{B}$ by $\boldsymbol{A}' \longrightarrow \boldsymbol{B}'$ be given. If now for every object a of $\boldsymbol{A}$ and every object b' of $\boldsymbol{B}'$ mapping to an object isomorphic to $f(a)$ there is an object a' of $\boldsymbol{A}'$ in the domain of the partial representation which maps to an object isomorphic to a and satisfies $f(a') = b'$ we speak of a partial representation of $\boldsymbol{A}$ over $\boldsymbol{B}$ by $\boldsymbol{A}'$ over $\boldsymbol{B}'$.

Definition 84 Let $\boldsymbol{C}, \boldsymbol{C}', \boldsymbol{B}, \boldsymbol{B}'$ be as above, where $\boldsymbol{C}$ is an elementary type, and let a partial representation of $\boldsymbol{C}$ over $\boldsymbol{B}$ by $\boldsymbol{C}'$ over $\boldsymbol{B}'$ be given. Then this partial representation is a *relative representation of the elementary type* $\boldsymbol{C}$ *over* $\boldsymbol{B}$ *by* $\boldsymbol{C}'$ *over* $\boldsymbol{B}'$ if it is a representation of $\mathrm{Top}(\boldsymbol{C})$ over $\mathrm{Cat}(\boldsymbol{B})$ by $\mathrm{Cat}(\boldsymbol{C}')$ over $\mathrm{Cat}(\boldsymbol{B}')$.

The following definition is convenient for applications in algorithms.

Definition 85 Let $\mathcal{F}$ be a representation of the complex type $\boldsymbol{C}$ by the complex type $\boldsymbol{C}'$. If the following holds, we call $\mathcal{F}$ a *strong representation* of the complex type $\boldsymbol{C}$ by the complex type $\boldsymbol{C}'$.

Let $\boldsymbol{C}$ be derived from $\boldsymbol{T}$ and $\boldsymbol{C}'$ derived from $\boldsymbol{T}'$ such that $\mathcal{F}$ induces a partial representation of $\boldsymbol{T}$ by $\boldsymbol{T}'$. Furthermore, let complex types $\boldsymbol{B}, \boldsymbol{B}'$ be given to which $\boldsymbol{T}, \boldsymbol{T}'$ restrict. Then the induced partial representation of $\boldsymbol{T}$ by $\boldsymbol{T}'$ is a relative representation of $\boldsymbol{T}$ over $\boldsymbol{B}$ by $\boldsymbol{T}'$ over $\boldsymbol{B}'$.

Note here that we do not attach a meaning to the phrase "representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$" without the attribute "strong".

Remark 86 The use of the notation $\boldsymbol{T}, \boldsymbol{T}'$ is no coincidence and is related to an application of the definition to output types of algorithms in subsection 4.3.

4.2 Representations of computational problems

We now come to the application of representations to computational problems. Our idea is that given a computational problem and representations of the input and output types, we want to be able to formulate what it means to compute with the objects used for the representation. Concretely, say that we have a computational problem $\mathcal{P}$ from $\boldsymbol{I}$ to $\boldsymbol{T}$ and representations of $\boldsymbol{I}$ and $\boldsymbol{T}$ by $\boldsymbol{I}'$ and $\boldsymbol{T}'$, respectively. Then we want to formulate what it should mean to compute with the representing objects, that is, with the objects of $\boldsymbol{I}'$ and $\boldsymbol{T}'$, instead with the objects of $\boldsymbol{I}$ and $\boldsymbol{T}$. However, now not every object of $\boldsymbol{I}'$ need be involved in the computation because not every object of $\boldsymbol{I}'$ needs to lie in the domain of the representation. We therefore define:

Definition 87 Let $\boldsymbol{I}$ and $\boldsymbol{T}$ be categories. A *partial computational problem* from $\boldsymbol{I}$ to $\boldsymbol{T}$ is a computational problem $\mathcal{P}$ from a full subcategory $\boldsymbol{S}$ of $\mathrm{Cat}(\boldsymbol{I})$ which is closed under isomorphism to $\boldsymbol{T}$. Here, the subcategory $\boldsymbol{S}$ is called the *domain* of the problem and denoted by $\mathrm{dom}(\mathcal{P})$.

Definition 88 Let $\mathcal{P}$ be a partial computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$, and let partial representations $\mathcal{F}$ of $\boldsymbol{I}$ by $\boldsymbol{I}'$ and $\mathcal{G}$ of $\boldsymbol{T}$ by $\boldsymbol{T}'$ be fixed. Let furthermore a partial computational problem $\mathcal{P}'$ from $\boldsymbol{I}'$ to $\boldsymbol{T}'$ be given. We then say that $\mathcal{P}'$ *partially computes* $\mathcal{P}$ if the domain of $\mathcal{P}'$ contains all elements from $\mathrm{dom}(\mathcal{F})$ which are mapped under $\mathcal{F}$ into $\mathrm{dom}(\mathcal{P})$ and if for any such object a' and any object a of $\boldsymbol{I}$ isomorphic to $\mathcal{F}(a')$, every object in the class $\mathcal{P}'(a')$ is a representing object of some object of $\mathcal{P}(a)$.

If furthermore a restriction of $\mathcal{F}$ defines a representation of the domain of $\mathcal{P}$ (in other words, if every object of $\mathrm{dom}(\mathcal{P})$ is isomorphic to an object of the form $\mathcal{F}(a)$), we say that $\mathcal{P}'$ *computes* $\mathcal{P}$.

Remark 89 In the definition, for a fixed a', one just has to require the condition for one a; it then follows that it holds for all a as given.

Remark and Definition 90 Let $\mathcal{P}$ be a partial computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$, and let partial representations $\mathcal{F}$ of $\boldsymbol{I}$ by $\boldsymbol{I}'$ and $\mathcal{G}$ of $\boldsymbol{T}$ by $\boldsymbol{T}'$ be fixed.

We can then associate to these data a partial computational problem $\mathcal{P}'$ from $\boldsymbol{I}'$ to $\boldsymbol{T}'$ as follows:

The domain of $\mathcal{P}'$ consists of the objects a' of $\mathrm{dom}(\mathcal{F})$ which are mapped under $\mathcal{F}$ into $\mathrm{dom}(\mathcal{P})$ and for which there exists an object of $\boldsymbol{T}'$ which is mapped to an object which is isomorphic to an object of $\mathcal{P}(\mathcal{F}(a'))$. For such an object we define $\mathcal{P}'(a')$ as the class of objects of $\boldsymbol{T}'$ which are mapped under the representation to objects which are isomorphic to objects in $\mathcal{P}(\mathcal{F}(a'))$.

We call the computational problem just described the *induced partial computational problem.*

By Definition 88, if furthermore a restriction of $\mathcal{F}$ defines a representation of $\mathrm{dom}(\mathcal{F})$ then $\mathcal{P}'$ computes $\mathcal{P}$.

Remark 91 If $\mathcal{G}$ is a representation of $\boldsymbol{T}$ by $\boldsymbol{T}'$, $\mathcal{P}'$ is a partial computational problem.

The definition of partial computational problem in an obvious way gives rise to a definition of a partial computational problem from some type $\boldsymbol{I}$ to some type $\boldsymbol{T}$ relative to some type $\boldsymbol{B}$. In addition, we define:

Definition 92 Given a type $\boldsymbol{B}$ and two types $\boldsymbol{C}$ and $\boldsymbol{D}$ over $\boldsymbol{B}$ as well as partial representations of $\boldsymbol{C}$ over $\boldsymbol{B}$ by $\boldsymbol{C}'$ over $\boldsymbol{B}'$ and $\boldsymbol{D}$ over $\boldsymbol{B}$ by $\boldsymbol{D}'$ over $\boldsymbol{B}'$, we say that the partial representations are *consistent* with each other over $\boldsymbol{B}$ if they induce the same partial representation of $\boldsymbol{B}$ by $\boldsymbol{B}'$.

We now have the following obvious definition.

Definition 93 Let $\mathcal{P}$ be a partial computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$ over $\boldsymbol{B}$. Furthermore, let $\mathcal{F}$ and $\mathcal{G}$ be partial representations of $\boldsymbol{I}$ and $\boldsymbol{T}$ by $\boldsymbol{I}$ and $\boldsymbol{T}$ respectively which are consistent with each other over $\boldsymbol{B}$. Now a partial computational problem $\mathcal{P}'$ from $\boldsymbol{I}'$ to $\boldsymbol{T}'$ relative to $\boldsymbol{B}'$ which partially computes $\mathcal{P}$ is said to partially compute $\mathcal{P}$ *relatively* to the given data.

We also give an analogous definition for induced computational problem in the relative situation:

Remark and Definition 94 Let $\mathcal{P}$ be a partial computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$ relative to $\boldsymbol{B}$, and let partial representations $\mathcal{F}$ of $\boldsymbol{I}$ over $\boldsymbol{B}$ by

$\boldsymbol{I}'$ over $\boldsymbol{B}'$ and $\mathcal{G}$ of $\boldsymbol{T}$ over $\boldsymbol{B}$ by $\boldsymbol{T}'$ over $\boldsymbol{B}'$ be fixed which are consistent with each other over $\boldsymbol{B}$.

We can then associate to these data a partial computational problem $\mathcal{P}'$ from $\boldsymbol{I}'$ to $\boldsymbol{T}'$:

The domain of $\mathcal{P}'$ consists of the objects a' of $\mathrm{dom}(\mathcal{F})$ which are mapped under $\mathcal{F}$ into $\mathrm{dom}(\mathcal{P})$ and for which there exists an object b of $\boldsymbol{T}'$ which is mapped under $\mathcal{G}$ to an object which is isomorphic to an object of $\mathcal{P}(\mathcal{F}(a))$ and which restricts to $\mathrm{res}^{\boldsymbol{I}'}_{\boldsymbol{B}'}(a')$. For such an object a' we define $\mathcal{P}'(a')$ as the class of objects c' of $\boldsymbol{T}'$ which are mapped under $\mathcal{G}$ to an object which is isomorphic to an object of $\mathcal{P}(\mathcal{F}(a'))$ and which satisfy $\mathrm{res}^{\boldsymbol{T}'}_{\boldsymbol{B}'}(c') = \mathrm{res}^{\boldsymbol{I}'}_{\boldsymbol{B}'}(a')$.

Remark 95 With the above notations, let us assume that $\mathcal{G}$ has the property that for every object a' of $\boldsymbol{T}$ and every object b' of $\boldsymbol{B}'$ mapping to an object isomorphic to a' there exists an object a' of $\boldsymbol{T}$ with $\mathrm{res}^{\boldsymbol{T}'}_{\boldsymbol{B}'}(a') = b'$ which is mapped under $\mathcal{G}$ to an object isomorphic to a. Then $\mathcal{P}'$ is defined on the whole domain of $\mathcal{F}$.

4.3 Algorithms and representations

We study now the idea of representation in the context of algorithms.

First, in the light of Definition 87, we define:

Definition 96 A *partial algorithm* is defined as in Definition 58 except that in the **let**- and **choose**-commands all partial computational problems rather than just computational problems can occur.

The application to an input instance is defined as in subsection 3.3 with the following change: When in a **let**-command a partial computational problem $\mathcal{P}$ is applied to an object complex for which the problem is not defined, the algorithm enters the state of "failure".

Similarly to Definition 66 we define:

Definition 97 Let $\mathcal{P}$ be a non-randomized partial computational problem from $\boldsymbol{I}$ to $\boldsymbol{T}$, and let $\boldsymbol{\mathcal{A}}$ be an algorithm with input type $\boldsymbol{I}$ and output type $\boldsymbol{T}$. We say that $\boldsymbol{\mathcal{A}}$ *computes* $\mathcal{P}$ if for all a in the domain of $\mathcal{P}$ for no computation path the algorithm fails and the class of possible outputs or the class of possible outputs with input modification (depending on the mode of the algorithm) is non-empty and contained in $\mathcal{P}(a)$.

We adapt Definition 67 in the same way.

Let now a computational problem $\mathcal{P}$ from $\boldsymbol{I}$ to $\boldsymbol{T}$ relative to $\boldsymbol{B}$ and algorithm $\boldsymbol{\mathcal{A}}$ for a machine of type $\boldsymbol{C}$ and of dimension d be given which computes $\mathcal{P}$.

Recall that by definition $\boldsymbol{C}$ has a fixed vertex v_0 to which the elementary type $\boldsymbol{N}_0$ is associated. Let a second type $\boldsymbol{C}'$ be given with a fixed vertex v_0' whose associated type is also $\boldsymbol{N}_0$. Let now a partial representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$ be given which maps v_0' to v_0 and restricts to the identity on the associated elementary type $\boldsymbol{N}_0$. According to the last sentence of Definition 82 we speak here of a representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$ over $\boldsymbol{N}_0$.

This partial representation defines a representation of $\boldsymbol{I}$ by some type $\boldsymbol{I}'$, of $\boldsymbol{T}$ by some type $\boldsymbol{T}'$ and of $\boldsymbol{B}$ by some type $\boldsymbol{B}'$. Here, both $\boldsymbol{I}'$ and $\boldsymbol{T}'$ restrict to $\boldsymbol{B}'$.

Let Δ be the maximum of the the differences $d_{\boldsymbol{C}',v} - d_{\boldsymbol{C},\varphi(v)}$ for vertices v of V, and let $d' := d + \Delta + 1$.

Let the notation of the representation be as in Definition 77.

We have the induced computational problem $\mathcal{P}'$ from $\boldsymbol{I}'$ to $\boldsymbol{T}'$ relative to $\boldsymbol{B}'$. It is now nearly immediate that $\boldsymbol{\mathcal{A}}$ defines an algorithm $\boldsymbol{\mathcal{A}}'$ which computes $\mathcal{P}'$: If in a **let**-command occurs some computational problem we substitute it by the induced computational problem. The assignments are transferred by first applying φ to the vertices. Note that here we use that we pad the addresses with leading $1'$s if necessary. Note also that if $\boldsymbol{\mathcal{A}}$ is non-randomized so is $\boldsymbol{\mathcal{A}}'$.

We immediately obtain the following proposition.

Proposition 98 Let us assume that the partial representation of $\boldsymbol{C}$ by $\boldsymbol{C}'$ is a strong representation. Then the partial algorithm $\boldsymbol{\mathcal{A}}'$ computes the partial computational problem $\mathcal{P}'$.

In fact, the condition in Definition 85 on strong representations only needs to hold for types $\boldsymbol{T}$ and $\boldsymbol{B}$ as in the definition which occur in **let**-commands in the algorithm.

4.4 Specification

Let now an algorithm $\boldsymbol{\mathcal{A}}$ for a machine of type $\boldsymbol{C}$ and of dimension d be given, and let us say that in line l of the algorithm there is a command of the form

let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$

or

let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$ **with update** .

Now, let $\boldsymbol{\mathcal{B}}$ be an algorithm which computes $\mathcal{P}$. Then intuitively, we can *specify* the line l of algorithm $\boldsymbol{\mathcal{A}}$ via algorithm $\boldsymbol{\mathcal{B}}$. Expressed differently, the reference *implements* the given line in algorithm $\boldsymbol{\mathcal{A}}$.

Let us now in addition suppose that the type $\boldsymbol{C}$ is represented by some type $\boldsymbol{C}'$ over $\boldsymbol{N}_0$ – as in the previous subsection. In is now reasonable to allow that $\boldsymbol{\mathcal{B}}$ operates on the representing objects from $\boldsymbol{C}'$ rather than on the represented objects from $\boldsymbol{C}$.

Example 99 Say in a particular line of an algorithm we use the computational problem of addition on an elliptic curve as described in Example 36: $(k; E; (P, Q))$, where k is a field, E an elliptic curve over k and $P, Q \in E(k)$ is mapped to $(k; E, P + Q)$. By specification we want to give information on the inner workings of this operation which up to now has been treated as a black box.

For this, we first fix a representation of the input and output types $\boldsymbol{X}$ and $\boldsymbol{Y}$. Here the complex type $\boldsymbol{Y}'$ representing $\boldsymbol{Y}$ is of second order, has one bottom vertex and three above it. The type for the bottom vertex is the type of fields. The objects of the elementary type defined by the first vertex at the top are $(k; f)$, where k is a field and $f \in k[x, y]$ is a Weierstraß-polynomial. The objects of the elementary type defined by the second top vertex are $(k; x)$, where k is a field and $x \in \mathbb{P}^1(k) = k \dot{\cup} \infty$, The objects of the third elementary type defined by the third top vertex are $(k; y)$, where k is a field and $y \in k$. Therefore, the objects of type $\boldsymbol{T}'$ are of the form $(k; (f, x, y))$ with the definitions as given.

The definition of the type $\boldsymbol{X}'$ is similar with the difference that there are five vertices on top and the objects are then $(k; (f, x_1, y_1, x_2, y_2))$. We now have an obvious representation of $\boldsymbol{X}$ by $\boldsymbol{X}'$ and of $\boldsymbol{Y}$ by $\boldsymbol{Y}'$.

Given this representation, we might now implement the addition with a second algorithm with input type $\boldsymbol{X}'$ and output type $\boldsymbol{Y}'$.

We state the general situation we have just described in the example:

We start with a partial algorithm $\boldsymbol{\mathcal{A}}$ for a machine of type $\boldsymbol{C}$ and of dimension d. Suppose in line l of $\boldsymbol{\mathcal{A}}$ there is a **let**-command based on a partial computational problem $\mathcal{P}$ with input type $\boldsymbol{X}$ and output type $\boldsymbol{Y}$ over $\boldsymbol{Z}$.

Let a partial representation of $\boldsymbol{C}$ by a complex type $\boldsymbol{C}'$ be fixed. This induces a partial representation of $\boldsymbol{X}$ by some complex type $\boldsymbol{X}'$, of $\boldsymbol{Y}$ by some complex type $\boldsymbol{Y}'$ and of $\boldsymbol{Z}$ by some complex type $\boldsymbol{Z}'$. We have the induced partial computational problem $\mathcal{P}'$ from $\boldsymbol{X}'$ to $\boldsymbol{Y}'$ over $\boldsymbol{Z}'$.

Then we can consider the specification by an algorithm with input type $\boldsymbol{X}'$ and output type $\boldsymbol{Y}'$ over $\boldsymbol{Z}'$ computing $\mathcal{P}'$.

What we just described is merely a special case of the situation we want to capture with a definition. The generalizations are as follows:

- We do not merely want to implement one line of an algorithm $\boldsymbol{\mathcal{A}}$ with

a **let**-command but an arbitrary number of lines. For each specialization / implementation we want to be able to use other representations.

- All these representations should however be consistent with one representation of the type $\boldsymbol{C}$.
- We want to be able to iterate the process of specification, leading to a *specification tree*.

These ideas are captured by the following definition.

In the definition, we use the following notations:

If a partial algorithm $\boldsymbol{\mathcal{A}}$ is given, the corresponding complex type is denoted by $\boldsymbol{C}_{\boldsymbol{\mathcal{A}}}$ and the corresponding dimension by $d_{\boldsymbol{\mathcal{A}}}$. The input type of the algorithm is denoted by $\boldsymbol{I}_{\boldsymbol{\mathcal{A}}}$ and the output type by $\boldsymbol{T}_{\boldsymbol{\mathcal{A}}}$. For each algorithm $\boldsymbol{\mathcal{A}}$ under consideration we fix a representation of $\boldsymbol{C}$ by some complex type we denote by $\boldsymbol{C}'$.

If $\boldsymbol{C}$ is a complex type for a partial algorithm and in the algorithm there is a line with a let-command based on a partial algorithm with input type $\boldsymbol{X}$ and output type $\boldsymbol{Y}$, this representation restricts to a representation of $\boldsymbol{X}$ by some type $\boldsymbol{X}'$ and of $\boldsymbol{Y}$ by some type $\boldsymbol{Y}'$.

Definition 100 A *complex of partial algorithms* consists of

i) a multigraph on a finite set of non-randomized or functorial partial algorithms without loops with labeled edges, where
   - the label of an edge from a partial algorithm $\boldsymbol{\mathcal{A}}$ to an algorithm $\boldsymbol{\mathcal{B}}$ is a line number of $\boldsymbol{\mathcal{A}}$ in which there is a **let**-command or a **choose**-command,
   - for each vertex (=partial algorithm) and each line number, there is at most one edge starting at the vertex with the given line number as label,

ii) a fixed vertex, called *partial input algorithm*,

iii) for each vertex (=partial algorithm) $\boldsymbol{\mathcal{A}}$,
   - a partial representation
   $$\mathcal{F}_{\boldsymbol{\mathcal{A}}} : \boldsymbol{C}'_{\boldsymbol{\mathcal{A}}} \longrightarrow \boldsymbol{C}_{\boldsymbol{\mathcal{A}}}$$
   such that for each edge from some vertex $\boldsymbol{\mathcal{A}}$ to some vertex $\boldsymbol{\mathcal{B}}$ labeled by a line number l of $\boldsymbol{\mathcal{A}}$,
     - if the command is preserving and the corresponding computational problem at line l of $\boldsymbol{\mathcal{A}}$ is $\mathcal{P}$ from $\boldsymbol{X}$ to $\boldsymbol{Y}$ over $\boldsymbol{Z}$, or

– if the command is updating and the corresponding computational problem at line l of $\boldsymbol{\mathcal{A}}$ is $\mathcal{P}$ from $\boldsymbol{X}$ to $\boldsymbol{X} \times_{\boldsymbol{Z}} \boldsymbol{Y}$ over $\boldsymbol{Z}$,

the types $\boldsymbol{X}'$ and $\boldsymbol{I}'_{\boldsymbol{\mathcal{B}}}$ are essentially equal and the types $\boldsymbol{Y}'$ and $\boldsymbol{T}'_{\boldsymbol{\mathcal{B}}}$ are essentially equal,

- partial representations as indicated in the following diagrams such that the diagrams

$$\begin{array}{ccc} \boldsymbol{X}' & \longrightarrow & \boldsymbol{X} \\ \| & & \uparrow \\ \boldsymbol{I}'_{\boldsymbol{\mathcal{B}}} & \longrightarrow & \boldsymbol{I}_{\boldsymbol{\mathcal{B}}} \end{array} \qquad \begin{array}{ccc} \boldsymbol{Y}' & \longrightarrow & \boldsymbol{Y} \\ \| & & \uparrow \\ \boldsymbol{T}'_{\boldsymbol{\mathcal{B}}} & \longrightarrow & \boldsymbol{T}_{\boldsymbol{\mathcal{B}}} \end{array} ,$$

where the upper rows are the induced partial representations, commute, and such that both partial representations induce the same representations in the diagram

$$\begin{array}{ccc} \boldsymbol{Z}' & \longrightarrow & \boldsymbol{Z} \\ \| & & \uparrow \\ \boldsymbol{B}'_{\boldsymbol{\mathcal{B}}} & \longrightarrow & \boldsymbol{B}_{\boldsymbol{\mathcal{B}}} \end{array}$$

such that for each edge from a partial algorithm $\boldsymbol{\mathcal{A}}$ to a partial algorithm $\boldsymbol{\mathcal{B}}$ labeled by line number l of $\boldsymbol{\mathcal{A}}$ as above,

- $\boldsymbol{\mathcal{B}}$ computes the partial computational problem induced by $\mathcal{P}$ and the partial representations of $\boldsymbol{X}$ by $\boldsymbol{I}_{\boldsymbol{\mathcal{B}}}$, $\boldsymbol{Y}$ by $\boldsymbol{T}_{\boldsymbol{\mathcal{B}}}$ and $\boldsymbol{Z}$ by $\boldsymbol{B}_{\boldsymbol{\mathcal{B}}}$
- if moreover the line in algorithm $\boldsymbol{\mathcal{A}}$ is

choose a_1 **relative to** a_0

based on an elementary type $\boldsymbol{A}$ be implemented by an algorithm $\boldsymbol{\mathcal{B}}$, then for any input instance a either all computation paths fail or the image of the output in $\boldsymbol{A}$ (which is then a random variable) is uniformly randomly distributed.

If such a complex of partial algorithms is given then if there is a line from algorithm $\boldsymbol{\mathcal{A}}$ to algorithm $\boldsymbol{\mathcal{B}}$ with line number l of $\boldsymbol{\mathcal{A}}$, we say that the command in line l of $\boldsymbol{\mathcal{A}}$ is *specified* by $\boldsymbol{\mathcal{B}}$ or *implemented* by $\boldsymbol{\mathcal{B}}$.

Definition 101 If in the above definition in iii) all partial representations are the identity (such that the data given reduce to the ones in i) and ii)), we speak of a complex of partial algorithms *without representation*.

Remark and Definition 102 If a complex of partial algorithms as in Definition 100 is given, one obtains, by considering the induced partial algorithms on the representing types defined in iii), a complex of partial algorithms without representation. We call this complex of partial algorithms the *complex of partial algorithms induced by the representations*. If the original complex of partial algorithms is given, we denote this complex by $\boldsymbol{C}'$.

Definition 103 A complex of partial algorithms is called *non-randomized* if all occurring algorithms are non-randomized.

In comparison, the definition of a *functorial* complex of partial algorithms is more difficult:

Definition 104 A complex of partial algorithms is called *functorial* if

- all occurring partial algorithms are functorial,
- all partial algorithms induced by the representations are functorial.

We now discuss the operation of a complex of partial algorithms. For this, we first pass to the associated complex of partial algorithms induced by the representations. After all, we introduced the representations in iii) and iv) in order to be able to pass from one algorithm to another one.

So let now a complex of partial algorithms without representations $\boldsymbol{C}$ be given. The intuitive idea is now as follows: Each algorithm $\boldsymbol{\mathcal{A}}$ has its own registers $R_{\underline{i}}$. Every time such an algorithm is initiated, all these registers are set to $0 \in \mathbb{N}_0$. Moreover, whenever an algorithm $\boldsymbol{\mathcal{A}}$ is called in a line with a command

$$\textbf{let } a_1 \longleftarrow \mathcal{P} \textbf{ applied to the registers given by } a_0 \textbf{ relative to } \boldsymbol{Z} \quad (1)$$

or

let $a_1 \longleftarrow \mathcal{P}$ **applied to the registers given by** a_0 **relative to** $\boldsymbol{Z}$ **with update**

we identify the registers pointed to by a_0 with the input registers $\mathrm{I}_{v,\underline{i}}$ of $\boldsymbol{\mathcal{A}}$.

The following definition generalizes Definition 102.

Definition 105 A *state* of a complex of partial algorithms without representation $\boldsymbol{\mathcal{C}}$ consists of

- a sequence of natural numbers $(l_1, \ldots, l_r)$, where l_1 is a label of an edge in the multigraph of $\boldsymbol{\mathcal{C}}$ starting at the input algorithm and ending at some algorithm $\boldsymbol{\mathcal{B}}$ (which means that line l_1 of $\boldsymbol{\mathcal{A}}$ is implemented by $\boldsymbol{\mathcal{B}}$), similarly l_2 is a the label of an edge in the multigraph starting at $\boldsymbol{\mathcal{B}}$ and so on until $r-1$ and l_r is an arbitrary line of the last algorithm called,

- for each of the algorithms involved in the path just described a content as defined in Definition 50 such that the identifications of input registers with registers $\mathrm{R}_{\boldsymbol{\mathcal{A}},\underline{i}}$ as described above are made.
- an object o from the output type of the input algorithm or one of two other objects called "running" and "failure". (In the first case, we say that the algorithm has terminated with output o.)

Now, given a state, we obtain in an obvious way a next state. It follows that given an input instance to the input algorithm, we obtain in an obvious way *computation paths* and the *class of possible outputs*. Just as for a single algorithm, we can say what it means that a non-randomized or a functorial complex of partial algorithms without representation *computes* a computational problem – the definition is again just as above.

The following lemma is immediate.

Lemma 106 Let a complex of algorithms without representation be non-randomized or functorial. If then the input algorithm computes $\mathcal{P}$ then the complex of algorithms without representation computes $\mathcal{P}$.

Definition 107 Let $\mathcal{P}$ be a partial computational problem and $\boldsymbol{\mathcal{C}}$ be a non-randomized or a functorial complex of partial algorithms with the appropriate input and output types. We say that $\boldsymbol{\mathcal{C}}$ *computes* $\mathcal{P}$ if the complex of partial algorithms induced by the representation computes the induced partial computational problem $\mathcal{P}'$ and $\mathcal{P}'$ computes $\mathcal{P}$ (which means that the induced representation of the input type of the input algorithm restricts to a representation of the domain of $\mathcal{P}$).

Finally, we briefly discuss the notion of *complexity* in the context of complexes of partial algorithms, continuing with the discussion for a single algorithm in subsection 3.5.

Let a complex type $\boldsymbol{\mathcal{C}}$ be given. Then for each induced partial computational problem which occurs in a **let**-command which is *not* specified, we fix a complexity measure. Now, we proceed just as for a single algorithm with the obvious difference that for each occurrence of a **let**-command which refers to another algorithm, we loop into this algorithm.

Discussion

It might seem to be reasonable to remove the condition in the definition of "complex of partial algorithms" that the multigraph on set of vertices has no loops. If this condition was removed, one could also apply recursion.

With this alternative definition, the previous lemma would be false: Indeed, one can then consider a complex of partial algorithms consisting of

only one algorithm for a computational problem $\mathcal{P}$ which refers to itself for computing $\mathcal{P}$. This algorithm would then never terminate, but of course, the input algorithm (being the algorithm itself) would compute $\mathcal{P}$.

The reason why we do not allow loops in the multigraph is that we want to study the idea "specification" of algorithms and also allow for the possibility of representation. This is a goal which is different from the goal to study recursion.

4.5 Bit oriented computations

In this section, we give the link between the very general computational model presented so far and usual RAM-models. The guiding idea is that the following should be conveniently possible: To give an algorithm in a RAM-model which operates on bit-strings representing certain mathematical objects, it should be possible to start out with some "abstract algorithm", followed by repeated specifications (including representations, as given in Definition 100) until one finally obtains an algorithm with essentially the commands of that of a usual RAM-model operating on a multi-dimensional storage. As already explained in the introduction, there is no essential difference between such an algorithm an an algorithm in a RAM-model for the usual one-dimensional storage.

We begin with the following definition, referring to Definition 33:

Definition 108 A complex type for which for all elementary types the corresponding fibers are trivially fibered with trivial fiber or with fiber $\boldsymbol{N}_0$ is called *bit-oriented*.

Remark 109 The category of a bit-oriented complex type is (canonically isomorphic to) a first order type of the form $\mathrm{array}^{d_1}(\boldsymbol{N}_0) \times \cdots \times \mathrm{array}^{d_r}(\boldsymbol{N}_0)$. The object complexes can therefore be given by tuples of arrays of non-negative integers.

Definition 110 A partial representation of a complex type is *bit-oriented* if the representing type is bit-oriented.

To give the link to the classical addition RAM model, we now define what we mean by a *bit-oriented algorithm*. For this, we restrict the computational problems which are allowed in the **let**-commands:

- For each type $\boldsymbol{C}$ we have a computational problem $\mathrm{id}_{\boldsymbol{C}}$ which, as the name suggests, is given by the identities. Obviously, this is a computational problem over each type to which $\boldsymbol{C}$ restricts.

- For each non-negative integer i we have a computational problem c_i from the trivial type to the type $\boldsymbol{N}_0$ which gives i.

- For each trivially fibered elementary type $\boldsymbol{A}$ with trivial fiber we have a computational problem $\mathsf{init}_{\boldsymbol{A}}$ (from initialize) from $\mathrm{Base}(\boldsymbol{A})$ to $\boldsymbol{A}$ which is given by $a \mapsto (a; *)$ if $*$ is the unique object of the fiber $\boldsymbol{F}$. This is a computational problem over $\mathrm{Base}(\boldsymbol{A})$ (and therefore over each complex type to which $\mathrm{Base}(\boldsymbol{A})$ restricts).

- For each fibered elementary type $\boldsymbol{A}$ with fiber $\boldsymbol{F}$ we have a computational problem $\mathsf{fromtop}$ from the elementary type $\boldsymbol{A}$ to $\boldsymbol{F}$ which is given by $(a; b) \mapsto b$. This is again a computational problem over $\mathrm{Base}(\boldsymbol{A})$.

- For each fibered elementary type $\boldsymbol{A}$ with fiber $\boldsymbol{F}$ we have a computational problem totop from the complex type $\mathrm{Base}(\boldsymbol{A}) \times \boldsymbol{F}$ to $\boldsymbol{A}$ which is given by $(b, n) \mapsto (b; n)$. This is also a computational problem over $\mathrm{Base}(\boldsymbol{A})$.

- We have a computational problem *Add* which is defined in the obvious way: The input type $\boldsymbol{I}$ is the first order complex type $\boldsymbol{N}_0 \times \boldsymbol{N}_0$ consisting of two vertices each of which is labeled by $\boldsymbol{N}_0$. The output type $\boldsymbol{T}$ is the elementary type $\boldsymbol{N}_0$. Now, to $(a, b) \in \mathrm{Cat}(\boldsymbol{I}) = \boldsymbol{N}_0$ the number $a + b \in \boldsymbol{T} = \boldsymbol{N}_0$ is assigned.

- Similarly, we have a computational problem *Sub* which is given by assigning to $(a, b) \in \mathrm{Cat}(\boldsymbol{I}) = \boldsymbol{N}_0$ the number $\max\{a-b, 0\} \in \boldsymbol{T} = \boldsymbol{N}_0$

We remark here that the last two problems, addition and subtraction, are exactly the commands of the classical addition RAM model. They can be substituted by other commands if one wants to consider the computation in other RAM models.

Furthermore only the computational problem of computing either 0 or 1 is allowed in a **choose**-command. Formally, the problem is as follows: We consider the elementary type $\boldsymbol{N}_0$ and the problem given by assigning to the trivial complex the set $\{0, 1\}$.

Definition 111 A partial algorithm is called *bit-oriented* if its type $\boldsymbol{C}$ is bit-oriented and the **let**-commands are based on computational problems as given above.

Note that in the description of the allowed computational problems we did not impose that the types be bit-oriented but this requirement is nonetheless present in the definition above.

Definition 112 A complex of partial algorithms is called *bit-oriented* if all representing types are bit-oriented and for the algorithms which are not specified the representations are trivial and the algorithms are bit-oriented.

We assign to all the relative computational problems in a bit-oriented algorithm the following time complexity: The complexity of a problem from $\boldsymbol{X}$ to $\boldsymbol{Y}$ over $\boldsymbol{Z}$ applied to some input is given the sum of the bit-length of the numbers for the vertices in $\boldsymbol{X}$ not in $\boldsymbol{Z}$.

We now strive for an efficient simulation of a bit-oriented algorithm by an addition RAM.

For this, we consider a computational problem $\mathcal{P}$ with input type $\boldsymbol{I}$ and output type $\boldsymbol{T}$.

As already stated in the introduction, up to logarithmic factors, one can simulate a "multi-dimensional addition RAM" by a usual addition RAM. We therefore obtain:

Proposition 113 If a bit-oriented non-randomized partial computational problem can be computed in time T with a bit-oriented partial algorithm, it can be computed in time $\tilde{O}(T)$ with a deterministic addition RAM.

This implies:

Proposition 114 Let a partial computational problem be given, and let a bit-oriented non-randomized complex of partial algorithms be given which computes the partial problem in a time of T. Then with respect to the given partial representations of the input and output types, the problem can be computed in a time $\tilde{O}(T)$ with a deterministic addition RAM.

And:

Proposition 115 Let a partial computational problem be given, and let a bit-oriented functorial complex of partial algorithms be given which computes the partial problem in an expected time of T. Then with respect to the given partial representations of the input and output types, the problem can be computed in an expected time $\tilde{O}(T)$ with an addition RAM.

5 Perspectives

We mention some perspectives for expansion of the model.

Further computational models. In the previous subsection, we discussed what we called bit-oriented computations. The computational model discussed there is just one of the computational models which might be

used. In addition, as stated in the introduction, it is now also classical to regard computational models in which exact operations with field elements can be performed. In this spirit, one can – within the framework of abstract algorithm – regard a computational model relying on the storage of fields, elements in arbitrary fields and of non-negative integers. Besides the usual field operations, it might also be of interest to allow commands based on the computational problem of factorization of polynomials.

Automated construction of complex types and representations. Complex types are one of the bases for the concepts presented. In the presentation one interesting aspect about types is however absent: The types form themselves a category. One can therefore also regard transformations of types, or more accurately, one can regard functors from a full subcategory of the category of types to the category of types. In fact, a particular family these functors was frequently used: The functor array^n for first order elementary types.

If $\boldsymbol{A}$ is a first order elementary type with a fixed representation by a first order elementary type $\boldsymbol{B}$, we obtain immediately a representation of $\text{array}^n(\boldsymbol{A})$ by $\text{array}^n(\boldsymbol{B})$. Maybe of greater importance than the automated generated of types is the generated of types with representations. We give another example for this:

Say for a *fixed* commutative ring R we consider the type of elements of R: $\boldsymbol{R}$. Say we have a representation of $\boldsymbol{R}$ by some set S fixed. (Concretely, this means that we have a surjective map $S \longrightarrow R$ fixed.) We then have an obvious representation of $\boldsymbol{R}[\mathbf{x}]$ by $\text{array}^1(\boldsymbol{S})$. If we now let R and the representation vary, we obtain an assignment from commutative rings with representation by sets to commutative rings with representation. With the appropriate definitions, this assignment gives also rise to a functor between categories.

Axiomatic semantics. Axiomatic semantics provide a formalized way to argue about algorithms via formal propositions. Just the same is possible here. Recall here that in the commands of abstract algorithms computational problems occur, and that these computational problems are themselves mathematical objects, generalizing functors. Of course, types and computational problems should then also occur in the propositions. This means that the propositions are not mere "formulae".

Metalanguage. An abstract algorithm is a mathematical object. Just as any mathematical object, it can be described in different ways. In particular, one need not write down the commands in the way written down above. It

is the content which is important, not the form. We give some examples for this:

- One can introduce variables with arbitrary names. These variables are then simply identified with the registers. These variables can be introduced in an ad hoc manner, as is usual in a description via pseudo-code. We remark here that in the analysis of algorithms, it is appropriate to distinguish between a variable and its content at a specific time. This can be achieved by using different symbols for variables and values. For example, we used $\mathrm{R}_{\underline{i}}$ for a register and $R_{\underline{i}}$ for its content.

- Also, one need not follow the syntax for the commands suggested above. For example, if a, b, c are variables and in the context of the algorithm always store integers, it is perfectly clear to write "Let $c \longleftarrow ab$".

- To give another example, say that the type of pointed groups is used. Recall again that a pointed group is nothing but a group with an element of the group. Now, as remarked above, a "group element" is just the same as a "pointed group". It therefore causes no harm if the intuition is changed to group elements.

- Continuing with the previous example, say two elements in an abelian group shall be added. In the context of abstract algorithms, this is given by a functor from the complex type of groups with two elements to the elementary type of groups with one element. Say now that a and b are variables which always store pointed groups. Then it is perfectly well defined to simply write, e.g., "Let $a \longleftarrow b + c$".

- One can also use further control commands in the meta-language. Examples are **if ... then ... else**, **while ... repeat** or **repeat ... until**.

6 Discussion

In the introduction we have stated that the uses of the word "algorithm" are divers. Here we give a more detailed discussion on the notion of *algorithm*, its different uses and the scientific implication of its uses.

We begin by this question: How is the word "algorithm" used? More specifically, we ask how the word is used in a scientific context, in particular in scientific literature. Even more precisely, we ask how the word is used in the mathematics, to which we also include mathematically oriented computer science. We exclude thus any potential use of the word associated to physical operations.

The two meanings of "algorithm". As already sated, broadly speaking, the word "algorithm" is used in two different ways: The first way is quite informal. Here, usually, some kind of procedure is given in a rather intuitive way, using pseudo-code. The second use of the word is formal. Here, it is defined that the word "algorithm" should mean a particular mathematical object with a particular rigorous definition. This definition might for example be "Turing machine" with respect to a particular definition of Turing machine or "RAM machine" with respect to a particular definition of RAM machine.[5] So, one can find the same word "algorithm" both in a theorem in which formally the existence of a mathematical object like a Turing machine with certain properties is postulated and in an informal description of the computation performed by the said Turing machine. Quite often, one can find both uses in the same scientific work.[6]

One aspect of the formal application of the word is that a representation of the mathematical objects under consideration, by specific objects which can be manipulated by the algorithm have to be fixed. Usually, these specific objects are bit-strings / natural numbers, however, in some formal computational models some other objects are considered. For example, in "algebraic complexity theory" as in [BCS91] one considers computations with elements of some field. In contrast, for an algorithm in the informal sense, one can argue about computations with arbitrary mathematical objects.

Now, of course it would be wrong to say that one can prove something about an algorithm in the informal sense by the very meaning of "to prove" within mathematics: One cannot prove anything if no clear definitions have been given to start with, and an informally given "algorithm" by the word "informal" itself does not have a proper definition. On the other hand, arguments about informally given algorithms are really the key to the analysis of algorithms. How is this possible? Two possible answers come to mind here:

The first possibility is that really statements about formally defined algorithms are made. So every statement on an algorithm in the informal sense should be interpreted as a statement on an algorithm in the formal

[5]We would like to stress that – as already stated – such a "machine" is indeed a mathematical object. In fact, it is an object within set-theory in just the same way as objects like fields, vector spaces or manifolds. So to claim that such a "machine" with certain properties exist is just as much a statement of (pure) mathematics as a statement that a field, a vector space or a manifold with certain properties exists.

[6]In addition, there is no consensus on whether an algorithm should by definition always terminate. For example, Doland Knuth in [Knu97] formulates this condition and advises to talk about a "computational method" for the more general concept. Generally speaking, this condition is however not imposed. In fact, often one reads a phrase similar to "The algorithm always terminates", which implicitly implies that even before termination had been established, it was reasonable to talk about an "algorithm". We have not imposed the condition of termination throughout the work and also do not impose it here.

sense. For example, in algorithmic number theory the computational model one uses is usually an addition RAM model. Here, implicitly a representation of the objects under consideration by bit-strings is fixed and for all the commands in pseudo-code "implementations" in the model are fixed. But depending on the algorithm given, even then there might be quite a bit of ambiguity concerning the complete machine, for example concerning the data storage.

The second possibility is that one really argues about the mathematical objects actually present in the algorithm in the informal sense, however one implicitly uses an induction argument. This means that without stating this explicitly, one really talks about a (possibly finite) sequence of mathematical objects defined by a rule laid out in the algorithm, and one proves statements on this sequence. The fact that one should analyze algorithms by induction (but without reference to sequences) is emphasized in standard textbooks like [CLRS01].

To give an example of this approach: Say a particular "complete Gaussian elimination algorithm" shall be analyzed. It shall be proven that the algorithm terminates with a reduced row normal form. The structure of an argument for this usually has about this form:

> Suppose that at some time the matrix A considered in the algorithm has this form. Then after the update the new matrix A' has this "nicer" form. One therefore sees that after some steps, a reduced row normal form is reached.

Such an argument (if properly stated) is surely valid, precisely because it is in fact nothing but a proof by induction on a particular sequence of matrices. So the argument on the termination of the algorithm (which we recall is only informally given and on which we cannot prove anything just because it is given informally) is turned into a proof on a sequence of matrices. However, intuitively the situation is in fact the opposite: Every matrix is right-equivalent to a matrix in row normal form *because* the complete Gaussian elimination method terminates.

Now, in the particular example just given, it is already now possible to give a formally defined algorithm because as mentioned above there are computational models based on computations in fields. The already mentioned model in [BCS91] is an example. This model is non-uniform, but there are also uniform models which are used in "real complexity theory" – here the computations are always in the field of real numbers, but this can be easily generalized. However, computations with elements of a particular field are just a very particular example of "abstract computations" one might like to perform. To stay with the particular example, in another algorithms (in the informal sense) computations with matrices might be performed, and in

this algorithm a command "Let B be the reduced row normal form of A" or – what is the same – "Compute the row normal from B of A" might occur. If one now wants to argue about the correctness of an algorithm in which such a command occurs, it is natural to regard matrices as the objects one performs computations with and to postulate that a command as the one indicated is available in the computational model. Or with other words: It is not necessary to think of the matrices as being represented by something else (for example tuples of field elements) and it is not necessary to think about how one would perform the computation with the help of some more "basic" commands. However, when one argues about complexity, one might want to perform the analysis with field operations, and at this point one might be interested how the matrices are represented and how the computation is "implemented" with field elements. But one might not even want to stop there. One might, for example, then consider computations over finite fields and analyze the computation in a RAM model.

Two words for two meanings. Even though the word "algorithm" is used in the two different ways indicated, and often it is used so in the same work, it is also common that different words or phrases are used to denote different levels of "abstraction".

One possibility is to only only speak of an "algorithm" if a certain degree of concreteness is reached and otherwise to speak for example of an "algorithm scheme". So, an algorithm scheme can be "specialized" to an algorithm. From a systematic point of view this terminology is however not convincing: What is in the end called an "algorithm" is often just as informally described as the "algorithm scheme" one started with; the difference is just that it is "more concrete". We note here that even an innocently looking command like "Let $c := ab$" for variables a, b which can take values in the natural numbers still has to be implemented if one performs computations with in the addition RAM model.

Another related but different possibility is to reserve the work "algorithm" strictly for the formal meaning. Starting from this, one possibility is then to always say that one "outlines" an algorithm when one gives an algorithm in the informal sense. Another possibility is to use another word for an algorithm in the informal sense. The word "algorithm scheme" is not appropriate for the informal meaning – who would for example talk about a "Gaussian elimination algorithm scheme" just because not every operation is fixed with respect to a particular model of computation? But one might use for example the word "procedure" or "method" for the informal meaning.

Just another possibility is to use the word "algorithm" for the informal meaning and to use another word for the formal meaning. Here, evidently,

the word "machine" suggests itself for the formal meaning. This approach is for example taken in the work [Die13] by the author.

We remark that this discussion is only about problems associated with the use of the word "algorithm". As stated, the problem is however deeper: If a situation is only informally described one cannot prove anything about it, no matter if what one argues about is called "algorithm", "algorithm scheme" or "procedure".

To conclude a personal comment

This work was difficult to write. A lot of effort was necessary for what I consider – at least for the moment – the "right" definitions. Are they the "right" ones? Of course, this can be argued. Naturally, there is no monopoly on the "right definitions" for an informal concept.The definitions given should not be considered to be set in stone and other definitions starting with the same ideas are possible. Having said this, I am however convinced that starting from the idea that one should take a "categorial approach" to algorithms, some underlying ideas and some definitions are at least in principle indeed the "right ones". This includes, for example:

- The emphasis on large groupoids: Isomorphisms are important and should be respected. Further morphisms are not important.
- The idea of a computational problem and the fact that each computational problem can itself be the basis of a command in an algorithm.
- The definition of representations of first order elementary types via partial functors.

It would be presumptuous to assume that from now on "abstract algorithms" will be written down in the framework laid out. It is however my hope that at least the fundamental idea of the application of category theory to provide a framework for algorithms will inspire other researchers. Already the fact that a formal theory of "abstract algorithms" from a categorial point of view can be developed and that one can therefore reason about such "abstract algorithm" might be of interest and might serve as a basis for further developments.

Science in general is an open ended endeavor based on individual effort and interpersonal communication. In this sense I end this work by saying:

Comments are welcome!

References

[BCFS11] W. Bosma, J. Cannon, C. Fieker, and A. Steel, editors. *Handbook of Magma functions, Edition 2.17.* 2011.

[BCS91] P. Bürgisser, M. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory.* Springer-Verlag, 1991.

[BS03] E. Börger and R. Stark. *Abstract State Machines.* Springer, 2003.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill and The MIT Press, 2001. Second Edition.

[Die13] C. Diem. On the complexity of some computational models in the turing model. Preprint, 2013.

[DJB] D.J. Bernstein. The devil's guide to choosing new mathematical terminology. http://cr.yp.to/bib/devil-name.html.

[Gur99] Y. Gurevich. The Sequential ASM Thesis. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 1999.

[Gur00] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Trans. Comput. Log.*, 1:77–111, 2000.

[Knu97] D. Knuth. *The Art of Computer Programming Volume 1 (Fundamental Algorithms), Third Edition.* Addison-Wesley, 1997.

[Sch79] A. Schönhage. On the power of random access machines. In H. Maurer, editor, *Proc. 6th Internat. Coll. on Automata, Languages and Programming*, volume 71 of *LNCS*. Springer, 1979.

[SSB01] R. Stärk, J. Schmidt, and E. Bröger. *Java and the Java Virtual Machine.* Springer, 2001.

[Wik] Wikipeda article on *Pairing function.*

[WW86] K. Wagner and G. Wechsung. *Computational Complexity.* VEB Verlag der Wissenschaften, Berlin, 1986.

Claus Diem
Universität Leipzig
Mathematisches Institut
Johannisgasse 26
04103 Leipzig
Deutschland
diem@math.uni-leipzig.de