Interval-valued computations and its connection with PSPACE

Benedek Nagy [a,b], Sándor Vályi [a]

[a] *Department of Computer Science, Faculty of Informatics, University of Debrecen Hungary H-4010 Debrecen, P.O. Box 12*

[b] *Research Group on Mathematical Linguistics, Rovira i Virgili University Tarragona, Spain*

Abstract

At the conference CiE2005, B. Nagy introduced a new model for analog computations, namely interval-valued computations. In this model, computations work on the so-called interval-valued bytes, which are special subsets of the interval [0,1) rather than a finite sequence of bits. The question was posed there, which complexity is needed to solve $PSPACE$-complete problems in this paradigm. In this paper, after formalizing the computational model, we answer this question. We show that the validity problem of quantified propositional formulae is decidable by a linear interval-valued computation. As a consequence, all polynomial space problems are decidable by a polynomial interval-valued computation. Furthermore, it is proven that PSPACE coincides with the class of languages which are decidable by a restricted polynomial interval-valued computation.

Key words: Continuous space machine, Analog computation, Computing with interval-values, Computational complexity, Massively parallel computing
1991 MSC: 68Q10, 68Q15

1 Introduction

Based on the theory of the universal Turing machines the principle of classical computers were developed by John von Neumann. From theoretical point of view, these computers can compute everything that is Turing computable. There are some significant features of these wide-spread computers, such as

Email addresses: nbenedek@inf.unideb.hu (Benedek Nagy), valyis@inf.unideb.hu (Sándor Vályi).

the sequential run and the usage of binary system that allows to use classical Boolean logic.

Although Neumann-type computers work sequentially, they work with cells (fixed length sequences of bits) and some operations, such as the Boolean operations, are performed parallely on the bits of the cells (inner parallelism). The 'level' of the CPU is measured by the numbers of bits in a cell it uses ([18]). (It also can be related to the size of the alphabet of Turing machines, it is the information unit of the sequential process.) In the last decades the number of bits of cells of computers has permanently increased. Considering the increasing bit-size of cells, increasing Boolean algebras are needed to formalize the behaviour of computations. Many-valued Boolean algebras with increasing number of elements can describe this situation. Instead of taking different Boolean algebras for different number of bits in cells, one can employ an infinite Boolean algebra for a uniform representation of all the cases.

The most important idealization in the Turing model of traditional computers is that the memory (built up by cells of a given number of bits) can be linearly extended in an unrestricted way. This is a straightforward model of everyday practice when one can use as much memory as one needs to solve a given problem.

In [7], Nagy proposed another computational model, the so-called interval-valued computing. It involves another type of idealization – the density of the memory can be raised unlimitedly instead of its length. This new paradigm keeps some of the features of the traditional Neumann-Turing type computations. It works on specific subsets of the interval $[0, 1)$, more specifically, on finite unions of $[)$-type subintervals. In a nutshell, interval-valued computations start with $\left[0, \frac{1}{2}\right)$ and continue with a finite sequence of operator applications. It works sequentially in a deterministic manner. The allowed operations are motivated by the operations of the traditional computers: Boolean operations and shift operations. There is only an extra operator, the product. The role of the introduced product is to connect interval-values on different 'resolution levels'. Essentially, it shrinks interval-values. So, in interval-valued computing systems, an important restriction is eliminated, i.e. there is no limit on the number of bits of a cell in the system; we have to suppose only that we always have a finite number of bits. Of course, in the case of a given computation an upper bound (the bit height of the computation sequence) always exists, and it gives the maximum number of bits the system needs for that computation process. Hence our model still fits into the framework of the Church-Turing paradigm, but it faces different limitations than the classical Turing model. Although the computation in this model is sequential, the inner parallelism is extended. One can consider the system without restriction on the size of the information coded in an information unit (interval-value). It allows to increase the size of the alphabet unlimitedly in a computation.

As our results will show, interval-valued computations are suitable for dealing with polynomial space problems. In Section 2, the interval-valued computations are formalized based on [7]. In that paper, the problem SAT was solved by a linear interval-valued computation and the question was posed, whether there are $PSPACE$-complete problems decidable by linear interval-valued computations. In Section 3, we answer this question in the affirmative. A class of interval-valued computations such that the class of languages decidable by them coincides with $PSPACE$ is presented in Section 4. Concluding remarks and some interesting open questions close the paper in Section 6.

2 Interval-valued computations

In this section we formalize the interval-valued computing system given in [7]. First we define what an interval-value means. Then we present the allowed operations which can be used to build and evaluate computation sequences. Finally, we give the notions concerning decidability and computational complexity.

2.1 Interval-values

We note in advance that we do not distinguish interval-values (specific functions from [0,1) into $\{0,1\}$) from their subset representations (subsets of [0,1)) and we use always the more convenient notation.

Definition 1 *The set* $\mathbb{V}$ *of* interval-values *coincides with the set of finite unions of* $[)$*-type subintervals of* $[0,1)$.

Definition 2 *The set* $\mathbb{V}_0$ *of* specific interval-values *coincides with* $\left\{ \bigcup_{i=1}^{k} \left[\frac{l_i}{2^m}, \frac{1+l_i}{2^m}\right) \ : m \in \mathbb{N}, k \leq 2^m, 0 \leq l_1 < \ldots < l_k < 2^m \right\}$.

We note that the set of finite unions includes the empty set ($k = 0$), that is, $\emptyset$ is also an allowed interval-value. Essentially, the notion of interval-value coincides with the notion of generalized interval ([2]). In interval temporal logic ([1]), these intervals represent occuring, non-contiguous events. The main difference between the proposed interval-valued computational model and the existing interval logic approach is that the latter deals with problems about interval-values while the proposed system computes classical decision problems with the help of computations on such interval-values. For example, the proposed fractalian product is an operation that cannot be expressed by usual interval logic relations. However, generalized interval logic can provide tools

for reasoning about interval-valued computations.

2.2 Operators on interval-values

Similarly to traditional computers working on bytes, of course, we allow bit-wise Boolean operations. If we consider interval-values as subsets of [0,1) then the corresponding operations coincide with the set-theoretical operations of complementation ($\overline{A}$), union ($A \cup B$) and intersection ($A \cap B$). $\mathbb{V}$ forms an infinite Boolean set algebra with these operations. $\mathbb{V}_0$ is an infinite subalgebra of the last algebra.

Before we add some other operators, we introduce a function assisting the formulation of the following definition. Intuitively, it provides the length of the left-most component (included maximal subinterval) of an interval-value A.

Definition 3 *We define the function* $Flength : \mathbb{V} \to \mathbb{R}$ *as follows. If there exist* $a, b \in [0,1]$ *satisfying* $[a,b) \subseteq A$, $[0,a) \cap A = \emptyset$ *and* $[a,b') \not\subseteq A$ *for all* $b' \in (b,1]$, *then* $Flength(A) = b - a$, *otherwise* $Flength(A) = 0$.

$Flength$ helps us to define the binary shift operators on $\mathbb{V}$. The *left-shift* operator will shift the first interval-value to the left by the first-length of the second operand and remove the part which is shifted out of the interval $[0,1)$. As opposed to this, the *right-shift* operator is defined in a circular way, i.e. the parts shifted above 1 will appear at the lower end of $[0,1)$. In this definition we write interval-values in their original, "characteristic function" notation.

Definition 4 *The binary operators* $Lshift$ *and* $Rshift$ *on* $\mathbb{V}$ *are defined in the following way. If* $x \in [0,1]$ *and* $A, B \in \mathbb{V}$ *then*

$$Lshift(A,B)(x) = \begin{cases} A(x + Flength(B)), & \text{if } 0 \leq x + Flength(B) \leq 1, \\ 0 \text{ in other cases.} \end{cases}$$

$$Rshift(A,B)(x) = \begin{cases} A(frac(x - Flength(B))), & \text{if } x < 1, \\ 0 \text{ if } x = 1. \end{cases}$$

Here the function frac gives the fractional part of a real number, i.e., $frac(x) = x - \lfloor x \rfloor$, *where* $\lfloor x \rfloor$ *is the greatest integer which is not greater than* x.

In Figure 1 some examples can be seen for both operations $Rshift$ and $Lshift$. The second (ancillary-) operands are shown in grey to assist understanding, but they are not the real parts of the resulted interval-values. Now we explain

the so-called *fractalian product* on intervals.

Definition 5 *Let A and B be general interval-values and $x \in [0,1)$. Then the fractalian product $A*B$ includes x if and only if $B(x) = 1$ and $A\left(\frac{x-x_B}{x^B-x_B}\right) = 1$, where x_B denotes the lower end-point of the B-component including x and x^B denotes the upper end-point of this component, that is, $[x_B, x^B)$ is the maximal subinterval of B containing x.*

We can give this operation in a more descriptive manner. If A contains k interval components with ends $a_{i,1}, a_{i,2}$ $(1 \leq i \leq k)$ and B contains l components with ends $b_{i,1}, b_{i,2}$ $(1 \leq i \leq l)$, then we determine the value of $C = A * B$ as follows: we set the number of components of C to be $k \cdot l$. For this process we can use double indices for the components of C. The starting- and end points of the ij-th component are $a_{i1} + b_{j1}(a_{i2} - a_{i1})$ and $a_{i1} + b_{j2}(a_{i2} - a_{i1})$, respectively.

The idea and the role of this operation is similar to that of unlimited shrinking of 2-dimensional images in [19]. It will be used to connect interval-values of different resolution. We note, that for the results of the present paper, it would be enough to introduce a restricted version of product operation, taking products by only with $\left[0, \frac{1}{2}\right)$ as an operand. For future extensions of this research, we keep the binary product in the definition.

As we can observe in Figure 2, as well, the fractalian product of two interval-values is the result of shrinking the first operand to each component of the second one.

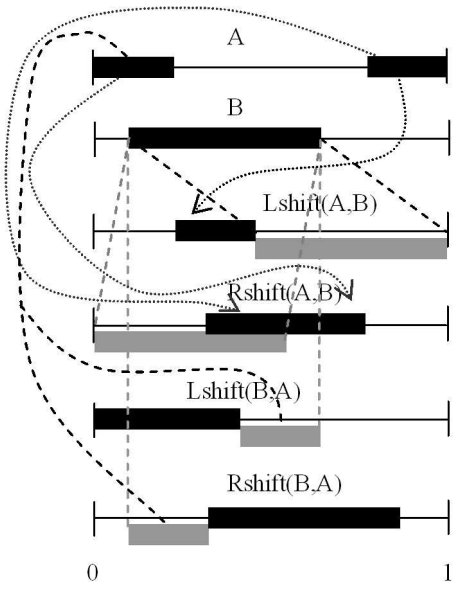

Fig. 1. Examples of shift operators with interval-values

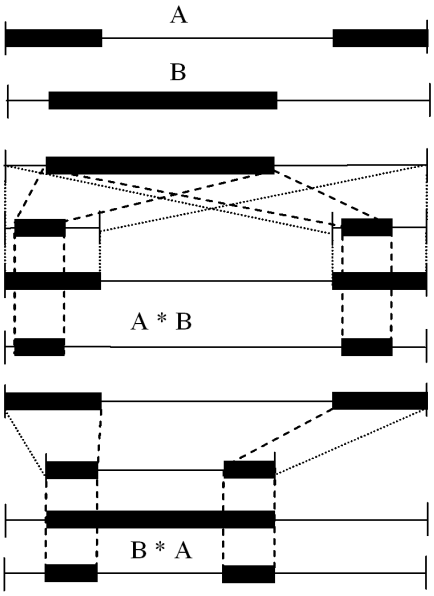

Fig. 2. Examples for product of interval-values

2.3 Syntax and semantics of computation sequences

In this subsection, we formalize the interval-valued computations of [7]. This formalisation is of Boolean network style, since equality or similar tests do not seem to be easily implementable for interval-values, just like in the case of optical computing (no tests for equalities on images). As usual, the length of a sequence S is denoted by $|S|$ and its i-th element by S_i. If $j \leq |S|$ then the j-length prefix of S is denoted by $S_{\rightarrow j}$.

Definition 6 *An* interval-valued computation sequence *is a nonempty finite sequence S satisfying $S_1 = FIRSTHALF$ and further, for any $i \in \{2, \ldots, |S|\}$, S_i is (op, l, m) for some $op \in \{AND, OR, LSHIFT, RSHIFT, PRODUCT\}$ or S_i is (NOT, l) where $\{l, m\} \subseteq \{1, \ldots, i-1\}$. The* bit height *of a computation is the number of the applied $PRODUCT$ operators in it.*

The semantics of interval-valued computation sequences is defined by induction on the length of the sequences. The *interval-value* of such a sequence S is denoted by $\|S\|$ and defined by induction on the length of the computation sequence, as follows.

Definition 7 *First, we fix $\|(FIRSTHALF)\|$ as $\left[0, \frac{1}{2}\right)$. Second, if S is an interval-valued computation sequence and $|S|$ is its length, then*

$$\|S\| = \begin{cases} \|S_{\to j}\| \cap \|S_{\to k}\|, \ \textit{if } S_{|S|} = (AND, j, k), \\ \|S_{\to j}\| \cup \|S_{\to k}\|, \ \textit{if } S_{|S|} = (OR, j, k) \\ \|S_{\to j}\| * \|S_{\to k}\|, \ \textit{if } S_{|S|} = (PRODUCT, j, k) \\ Rshift(\|S_{\to j}\|, \|S_{\to k}\|), \ \textit{if } S_{|S|} = (RSHIFT, j, k) \\ Lshift(\|S_{\to j}\|, \|S_{\to k}\|), \ \textit{if } S_{|S|} = (LSHIFT, j, k) \\ \overline{\|S_{\to j}\|}, \ \textit{if } S_{|S|} = (NOT, j). \end{cases}$$

One can notice, that in this formulation of interval-valued computations, only *specific* interval-values (cf. Definition 2) appear as values of computation sequences. However, this observation only strengthens our main result (Theorem 15) and makes it more likely to find implementations.

2.4 Decidability

In this subsection, we give the definitions concerning interval-valued computability and complexity.

Definition 8 *Let Σ be a finite alphabet and let $L \subseteq \Sigma^*$ be a language. We say that L is* decidable by an interval-valued computation *if there is an algorithm A that for each input word $w \in \Sigma^*$ constructs an appropriate computation sequence $A(w)$ such that $w \in L$ if and only if $\|A(w)\|$ is nonempty. Furthermore, we consider $\overline{L}$ also decidable in this case.*

This last remark makes it possible to test emptiness and, by applying set-theoretical operators, also to test whether $\|A(w)\| = [0, 1)$. The following statement is straightforward, since, algorithm A can be abitrary, on the one hand, and by the obvious fact, that if a language is decidable by an interval-valued computation then one can calculate and track the sequence of the limiting points (rationals in this model) of all the components of the actual interval-values, on the other.

Fact 9 *The class of languages decidable by an interval-valued computation coincides with the class of recursive languages.*

This fact shows that we have to narrow down the notion of acceptable interval-valued computations. In [7], SAT was solved by a linear interval-valued computation in the following meaning.

Definition 10 *We say that a language $L \subseteq \Sigma^*$ is* decidable by a linear interval-valued computation *if and only if there is a positive constant c and a logarithmic space algorithm A with the following properties. For each input*

word $w \in \Sigma^*$, A constructs an appropriate interval-valued computation sequence $A(w)$ such that $|A(w)|$ is not greater than $c \cdot (|w|)$ and $w \in L$ if and only if $\|A(w)\|$ is nonempty. Again, deciding $\overline{L}$ instead of L itself is allowed.

In this operator network style formulation of interval-valued computations, the size of the network is constrained. The question was raised in [7] whether there exists a $PSPACE$-complete language decidable by a linear interval-valued computation. We will answer this question in the next section. To solve all the problems in $PSPACE$ by interval-valued computations, it is useful to introduce the following notions.

Definition 11 *We say that a language $L \subseteq \Sigma^*$ is* decidable by a restricted polynomial interval-valued computation *if and only if there is a polynomial P and a logarithmic space algorithm A with the following properties. For each input word $w \in \Sigma^*$, A constructs an appropriate interval-valued computation sequence $A(w)$ containing product operators only of the form $(PRODUCT, 1, n)$ such that $|A(w)|$ is not greater than $P(|w|)$, further, $w \in L$ if and only if $\|A(w)\|$ is nonempty. Again, deciding $\overline{L}$ instead of L itself is allowed. If we omit the condition on the $PRODUCT$ operators, we obtain the notion of* polynomial interval-valued computations*.*

Having this restriction on products, one can take products of an interval-value only by the starting interval-value $\left[0, \frac{1}{2}\right)$. As the main result of the paper we will show that this restriction leads to a class of interval-valued computations that decide exactly the languages of $PSPACE$.

3 A linear interval-valued computation to decide a PSPACE-complete problem

3.1 The language of true quantified propositional formulae (QSAT)

We recall now the definition of (a suitable variant of) the language $QSAT$ of *true quantified propositional formulae*. It is a subset of satisfiable propositional formulae, say, built from the propositional variables $\{x_1, x_2, \ldots\}$, by the logical operators $\neg, \wedge, \vee$. We *do not explicitly put the quantifier prefix to the propositional formulae*, only the definition of the language is given this way. Variables with odd indices are meant to quantify universally while those with even indices to quantify existentially. It can be shown by renaming of variables and using fictive quantifiers that this variant is equally PSPACE-complete as the original $QSAT$ ([13]). Before we define $QSAT$, we have to make some preparations.

Definition 12 *A* valuation *is a function with range* $\{0,1\}$ *on the domain* $\{x_1,\ldots,x_n\}$ *for some positive integer* n. *If* $t_1,\ldots,t_n$ *are truth values then we write* $(t_1,\ldots,t_n)$ *for the valuation* v *that* $v(x_1)=t_1,\ldots,v(x_n)=t_n$ *and* $dom(v)=\{x_1,\ldots,x_n\}$. *For a quantifier-free formula* ϕ, $[[\phi v]]$ *denotes the* truth value *of* ϕ *by the valuation* v. *For any positive integer* i, the quantifier Q_i *is* $\forall$ *if* i *is odd otherwise it is* $\exists$.

Definition 13 *For any propositional formula* ϕ, ϕ *belongs to* $QSAT$ *if and only if there exists a positive integer* n *such that the propositional variables in* ϕ *are exactly* $x_1,\ldots,x_n$ *and* $(\forall t_1\in\{0,1\})(\exists t_2\in\{0,1\})\ldots(Q_n t_n\in\{0,1\}):[[\phi(t_1,\ldots,t_n)]]=1$ *holds.*

Example 14 $\phi=(((x_1\equiv x_2)\wedge\neg x_4))\vee(x_3\wedge((\neg x_1\wedge x_2\wedge\neg x_4)\vee(x_1\wedge\neg x_2\wedge x_4)))$ *is in* $QSAT$, *since* $(\forall t_1\in\{0,1\})(\exists t_2\in\{0,1\})(\forall t_3\in\{0,1\})(\exists t_4\in\{0,1\}):[[\phi(t_1,t_2,t_3,t_4)]]=1$ *holds. (Here* $\equiv$ *is the usual abbreviation of the logical connective 'equivalence'.) The index of a propositional variable determines if it is universally or existentially quantified.*

3.2 A linear interval-valued computation to decide QSAT

Theorem 15 *QSAT is decidable by a linear interval-valued computation.*

We give an algorithm to construct the computation sequence $K_1,\ldots,K_{11n+m-1}$ for any input formula ϕ that contains exactly the variables $x_1,\ldots,x_n$ and the number of its subformulae is m. The length of this list is less than $13\cdot|\phi|$, where $|\phi|$ is the length of ϕ. The algorithm provides the above computation sequence in such a way that its interval-value will be empty if and only if $\phi\in QSAT$.

Let K_1 be $FIRSTHALF$. For all positive integers $k\le n$, we define $K_{3k-1}=(PRODUCT,1,3k-2)$, $K_{3k}=(RSHIFT,3k-1,3k-2)$ and $K_{3k+1}=(OR,3k,3k-1)$.

By induction on k one can establish the following statement.

Lemma 16 *For all positive integer* k, *if* $k\le n$ *then*

$$\|K_{\to 3k-2}\|=\bigcup_{l=0}^{2^{k-1}-1}\left[\frac{2l}{2^k},\frac{2l+1}{2^k}\right).$$

The n independent truth values of $x_1,\ldots,x_n$ will be represented by the interval-values $\|K_{\to 1}\|,\|K_{\to 4}\|,\ldots,\|K_{\to 3n-2}\|$. See the example for $n=4$ in the first

four lines of Figure 3. Now we establish some further properties of $\|K_{\to 1}\|$, $\|K_{\to 4}\|, \ldots, \|K_{\to 3n-2}\|$.

Lemma 17 *For every $r \in [0,1)$ and positive integer $j \leq n$ hold the following conditions.*

(1) if $r \in \|K_{\to 3j-2}\|$ then for all $i < j$, $r + \frac{1}{2^j} \in \|K_{\to 3i-2}\|$ if and only if $r \in \|K_{\to 3i-2}\|$,
(2) if $r \notin \|K_{\to 3j-2}\|$ then for all $i < j$, $r - \frac{1}{2^j} \in \|K_{\to 3i-2}\|$ if and only if $r \in \|K_{\to 3i-2}\|$,
(3) $r + \frac{1}{2^j} \in \|K_{\to 3j-2}\|$ if and only if $r \notin \|K_{\to 3j-2}\|$.

Let $\phi_1, \ldots, \phi_m$ be an enumeration of all the subformulae of ϕ such that any formula is preceded by its subformulae (consequently, $\phi_m = \phi$). The algorithm gives the next part of the computation sequence $(K_{3n-2+1}, \ldots, K_{3n-2+m})$ in the following way. For each $i \in \{1, \ldots, m\}$,

$$K_{3n-2+i} = \begin{cases} (AND, 3n-2+j, 3n-2+k) \text{ if } \phi_i = \phi_j \wedge \phi_k, \\ (OR, 3n-2+j, 3n-2+k) \text{ if } \phi_i = \phi_j \vee \phi_k, \\ (NOT, 3n-2+j) \text{ if } \phi_i = \neg\phi_j, \\ (AND, 3j-2, 3j-2) \text{ if } \phi_i = x_j. \end{cases}$$

By induction on j the following statement can be verified.

Lemma 18 *For each $j \in \{1, \ldots, m\}$, $\|K_{\to 3n-2+j}\| =$ $\{r \in [0,1) : [[\phi_j(r \in \|K_{\to 1}\|, r \in \|K_{\to 4}\|, \ldots, r \in \|K_{\to 3n-2}\|)]] = 1\}$ holds.*

So far, we have obtained a linear size computation sequence to decide the satisfiability of $\phi (= \phi_m)$ by the validity of the following equivalence: ϕ is satisfiable if and only if $\|K_{\to 3n-2+m}\|$ is nonempty. This can be concluded from the fact, that for each n-tuple $(t_1, \ldots, t_n)$ of truth values there is an $r \in [0,1)$ such that $(\forall i \in \{1, \ldots, n\})$: $t_i = r \in \|K_{\to 3i-2}\|$.

The computation sequence continues with $K_{3n-2+m+1}, \ldots, K_{3n-2+m+8n}$ so that for each integer $j < n$, the following holds. $\|K_{\to 3n-2+m+8(j+1)}\| =$ $((Lshift(\|K_{\to 3n-2+m+8j}\|, \|K_{\to 3(n-j)-2}\|) \cap \|\overline{K_{\to 3(n-j)-2}}\|) \cup \|K_{\to 3n-2+m+8j}\|)$ $\cup((Rshift(\|K_{\to 3n-2+m+8j}\|, \|K_{\to 3(n-j)-2}\|) \cap \overline{\|K_{\to 3(n-j)-2}\|}) \cup \|K_{\to 3n-2+m+8j}\|)$, if $n-j$ is even, and $(Lshift(\|K_{\to 3n-2+m+8j}\|, \|K_{\to 3(n-j)-2}\|) \cap \|K_{\to 3(n-j)-2}\| \cap$ $\|K_{\to 3n-2+m+8j}\|) \cup (Rshift(\|K_{\to 3n-2+m+8j}\|, \|K_{\to 3(n-j)-2}\|) \cap \overline{\|K_{\to 3(n-j)-2}\|} \cap$ $\|K_{\to 3n-2+m+8j}\|)$ in the other case. In this definition, we do not specify all the intermediate expressions between $K_{3n-2+m+8j}$ and $K_{3n-2+m+8(j+1)}$, they are the subexpressions of $K_{3n-2+m+8(j+1)}$ needed to express $K_{3n-2+m+8(j+1)}$ from $K_{3n-2+m+8j}$ and $K_{3(n-j)-2}$.

To make the next lemma more readable, we assume without any further mention, that variables $t_1, t_2, \ldots, t_n$ range over the truth values. We recall that the quantifier sequence $Q_1, Q_2, Q_3, \ldots$ is defined as $\forall, \exists, \forall, \ldots$, respectively.

Lemma 19 *For each $j \in \{0, \ldots, n\}$ and for all $r \in [0,1)$:*
$r \in \|K_{\to 3n-2+m+8j}\|$ *if and only if*
$Q_{n-j+1}t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-j)-2}\|, t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1.$

Proof. The proof goes by induction on j from 0 up to n. For $j = 0$, Lemma 18 implies the needed equivalence, which is $r \in \|K_{\to 3n-2+m}\|$ if and only if $[[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3\cdot n-2}\|)]] = 1$.

Assume $j < n$. Let the induction hypothesis be the following. For any $r \in [0,1)$, $r \in \|K_{\to 3n-2+m+8j}\|$ if and only if
$Q_{n-j+1}t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-j)-2}\|, t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1.$

We have to show that $r \in \|K_{\to 3n-2+m+8(j+1)}\|$ if and only if
$Q_{n-j}t_{n-j} \ldots Q_n t_n$
$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-(j+1))-2}\|, t_{n-j}, \ldots, t_n)\Big]\Big] = 1,$
for arbitrary $r \in [0,1)$.

As a proof, we write a sequence of equivalent conditions starting with $r \in \|K_{\to 3n-2+m+8(j+1)}\|$ and closing with the right side of the equivalence. We prove the case when $n-j$ is even and Q_{n-j} is $\exists$, the proof, when $n-j$ is odd, can be constructed analogously.

(i) $r \in \|K_{\to 3n-2+m+8(j+1)}\|$;
(ii) $r \in \|K_{\to 3n-2+m+8j}\|$ or
$(r \in Lshift(\|K_{\to 3n-2+m+8j}\|, \|K_{\to 3(n-j)-2}\|) \wedge r \in \|K_{\to 3(n-j)-2}\|)$ or
$(r \in Rshift(\|K_{\to 3n-2+m+8j}\|, \|K_{\to 3(n-j)-2})\| \wedge r \in \overline{\|K_{\to 3(n-j)-2}\|})$;
(iii) $\forall t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-j)-2}\|, t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1$
or
$(r \in \|K_{\to 3(n-j)-2}\| \wedge \forall t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r + \frac{1}{2^{n-j}} \in \|K_{\to 3\cdot 1-2}\|, \ldots, r + \frac{1}{2^{n-j}} \in \|K_{\to 3(n-j)-2}\|, t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1)$
or
$(r \notin \|K_{\to 3(n-j)-2}\| \wedge \forall t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r - \frac{1}{2^{n-j}} \in \|K_{\to 3\cdot 1-2}\|, \ldots, r - \frac{1}{2^{n-j}} \in \|K_{\to 3(n-j)-2}\|, t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1)$;
(iv) $\forall t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-j)-2}\|, t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1$ or
$\forall t_{n-j+1} \ldots Q_n t_n$

$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-(j+1))-2}\|, r \notin \|K_{\to 3(n-j)-2}\|,$
$t_{n-j+1}, \ldots, t_n)\Big]\Big] = 1;$

(v) $\exists t_{n-j} \forall t_{n-j+1} \ldots Q_n t_n$
$\Big[\Big[\phi(r \in \|K_{\to 3\cdot 1-2}\|, \ldots, r \in \|K_{\to 3(n-(j+1))-2}\|, t_{n-j}, \ldots, t_n)\Big]\Big] = 1.$

The equivalence of (i) and (ii) is due to the definition of $K_{3n-2+m+8(j+1)}$. The equivalence of (ii) and (iii) follows from the following three properties: $Flength(\|K_{\to 3(n-j)-2}\|) = \frac{1}{2^{n-j}}$ (cf. Lemma 16); for every $r \in [0,1)$ and interval-values A, B: $r \in Lshift(A,B)$ if and only if $r + Flength(B) \in A$ and an analogous fact concerning $Rshift$. The equivalence of (iii) and (iv) can be shown by the propositions (1)–(3) of Lemma 17. Finally, the equivalence of (iv) and (v) can be shown by considering that only two possible truth values exist. The proof of the lemma is finished. Now we are ready to finish the proof of Theorem 15.

Proof of Theorem 15. Letting $j = n$, the above lemma ensures that $r \in \|K_{\to 3n-2+m+8n}\|$ if and only if $Q_1 t_1 \ldots Q_n t_n \ : [[\phi(t_1, \ldots, t_n)]] = 1$ holds for any $r \in [0,1)$. Since the right side of the last equivalence is independent from r, we can state that $Q_1 t_1 \ldots Q_n t_n \ : [[\phi(t_1, \ldots, t_n)]] = 1$ if and only if $\|K_{\to 3n-2+m+8n}\|$ is equal to $[0,1)$. Finally, by setting $K_{3n-2+m+8n+1}$ to $(NOT, 3n-2+m+8n)$ the algorithm constructs a computation sequence whose interval-value is empty if and only if $\phi \in QSAT$. Q.E.D.

Corollary 20 *Every language in $PSPACE$ is decidable by a restricted polynomial interval-valued computation.*

This can be proved in a way similar to that of proving the transitivity of log-space reducibility. One should only observe one more thing: the given interval-valued computation for $QSAT$ also satisfies the restriction on the applications of product.

Figure 3 gives an example of the computation on a formula. $(((x_1 \equiv x_2) \wedge \neg x_4)) \vee (x_3 \wedge ((\neg x_1 \wedge x_2 \wedge \neg x_4) \vee (x_1 \wedge \neg x_2 \wedge x_4)))$ is shown to be in $QSAT$.

4 Interval-valued computations that characterize $PSPACE$

The second achievement of the present paper is the following.

Theorem 21 *The class of languages decidable by a restricted polynomial interval-valued computation is equal to $PSPACE$.*

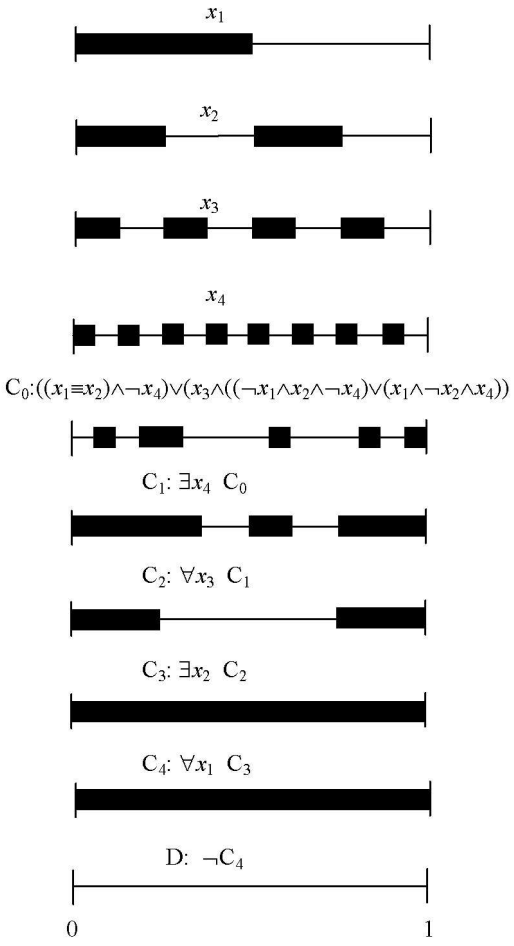

Fig. 3. $[\forall x_1 \exists x_2 \forall x_3 \exists x_4](((x_1 \equiv x_2) \wedge \neg x_4)) \vee (x_3 \wedge ((\neg x_1 \wedge x_2 \wedge \neg x_4) \vee (x_1 \wedge \neg x_2 \wedge x_4))) \in QSAT$ holds.

One direction of this class equation is already achieved in Corollary 20. For the reverse direction, we will construct a quadratic space algorithm which decides whether the value of an input interval-valued computation sequence is equal to the full [0,1). First we give a recursive algorithm to decide this problem. This guarantees only that the problem is solvable in exponential time. We also show how the execution of this recursive program can be equipped by a back-track like control in such a way that the needed memory is limited by a quadratic function of the length of the input computation sequence.

Lemma 22 *For any interval-valued computation sequence S of bit height m, $x, y \in \mathbb{R}$ and nonnegative integer l such that $l < 2^{m+1}$, if $\{x, y\} \subseteq \left[\frac{l}{2^{m+1}}, \frac{l+1}{2^{m+1}}\right)$ then $x \in \|S\|$ if and only if $y \in \|S\|$. In other words, $\left[\frac{l}{2^{m+1}}, \frac{l+1}{2^{m+1}}\right) \subseteq \|S\|$ or $\left[\frac{l}{2^{m+1}}, \frac{l+1}{2^{m+1}}\right) \cap \|S\| = \emptyset$.*

Proof. The proof can be formulated for an induction on $|S|$.

Below we introduce a notation naming some subintervals of $[0,1)$ that occur as values of computational sequences.

Definition 23 *We define a subinterval $i(w)$ for an arbitrary word $w \in \{0,1\}^*$ in the following way. Let us denote the length of w by $m = |w|$ and the k-th element of this sequence by w_k. If $v = \sum_{k=1}^{m} w_k 2^k$ then $i(w)$ is $\left[\frac{v}{2^m}, \frac{v+1}{2^m}\right)$. Under these circumstances, we call $i(w)$ an* m-elementary subinterval. *We denote the set of m-elementary subintervals by $\mathbb{E}_m$, that is, $\mathbb{E}_m = \left\{\left[\frac{l-1}{2^m}, \frac{l}{2^m}\right) \ : \ l \in \{1, \ldots, 2^m\}\right\}$. Furthermore, let $\mathbb{E}$ be $\bigcup_{m\in\mathbb{N}} \mathbb{E}_m$.*

Remark 24 *$i(\lambda) = [0,1)$ holds, also $\bigcup_{w\in\{0,1\}^m} i(w) = [0,1)$, if $m \geq 0$. Moreover, i is a bijection from $\{0,1\}^m$ onto $\mathbb{E}_m$.*

Now we can continue describing the algorithm, let us denote it by $\mathcal{B}$. It takes a computation sequence S as input and decide whether $\|S\| = [0,1)$. Hence one of the non-basic data types of this algorithm is the type of the computation sequences, or, specified in the narrowest sense, all the nonempty prefixes of S. The set of these prefixes is denoted by Seq. Clearly, its elements can be identified with positive integers not greater than $|S|$. The other type of data structure is given by the set of elementary intervals $\mathbb{E}$, which we represent by i as words in $\{0,1\}^*$. Let m denote the bit height of the input computation sequence. All the words while $\mathcal{B}$ is running on this sequence correspond to m-elementary subintervals, that is, elements of $\mathbb{E}_m$.

The algorithm uses both recursively and non-recursively definable functions.

Definition 25 *The functions of $\mathcal{B}$ computable in a recursive way are the following:*

$\sqsubset\colon \mathbb{E} \times Seq \to \{TRUE, FALSE\}$,

$\lhd : \mathbb{E} \times Seq \to \mathbb{E} \times (\mathbb{E} \cup \{\lambda\})$,

$\prec\colon \mathbb{E} \times Seq \to \mathbb{E} \times (\mathbb{E} \cup \{\lambda\})$.

The meaning of $(w \sqsubset S)$ is $i(w) \subset \|S\|$, $(w \lhd S)$ returns the starting and ending m-elementary subintervals of the maximal connected component of $\|S\|$ containing $i(w)$ where m is the bit height of S if such a component exists, (w, λ) otherwise. Finally, $(w \prec S)$ returns the starting and ending m-elementary subinterval of the maximal connected component of $[0,1) \setminus \|S\|$ containing $i(w)$ if such a component exists, (w, λ) otherwise.

Definition 26 *The directly, non-recursively definable (partial) functions of $\mathcal{B}$ are the following. (m is the bit height of the input computational sequence.)*

$bitheight : Seq \to \mathbb{N}$,

$< : \mathbb{E}_m \times \mathbb{E}_m \to \{TRUE, FALSE\}$,

$min, max : \mathbb{E}_m \times \mathbb{E}_m \to \mathbb{E}_m$,

$rotate\_left, rotate\_right : \mathbb{E}_m \times (\mathbb{E}_m \times \mathbb{E}_m) \to \mathbb{E}_m$,

$pred, succ : \mathbb{E}_m \to \mathbb{E}_m$,

$center, upper\_center : \mathbb{E}_m \times \mathbb{E}_m \to \mathbb{E}_m$.

The meaning of bitheight is straightforward. $w_1 < w_2$ *holds if and only if* w_1 *is strictly left to* w_2*. min and max works with respect to the just defined liner ordering* $<$*.* $rotate\_left(w, w_1, w_2)$ *returns the result of the shifting of* w *towards left by the length of the subinterval starting point* w_1 *and ending point* w_2*. If overflow occurs then the result or a part of it appears right to* w*. If* (w_1, w_2) *is empty then no shifting occurs. rotate_right is interpreted analogously.* $pred(w)$ *determines the left neighbour of* w *among the* m*-elementary intervals,* $pred(0^{|w|}) = \lambda$*,* $pred(\lambda)$ *is undefined. succ is the mirror of pred moving right,* $succ(1^m)$ *is undefined.* $center(w_1, w_2)$ *is the central* $|w|$*-elementary subinterval between* w_1 *and* w_2 *if it is unambiguous, that is, there is an odd number of* $|w_1|$*-elementary subintervals strictly between* w_1 *and* w_2*.* $upper\_center(w_1, w_2)$ *returns the bigger of the two central* $|w|$*-elementary subintervals between* w_1 *and* w_2*.*

If m is the bit height of S, then by Lemma 22, instead of deciding $\|S\| = [0, 1)$, it is enough to decide in polynomial space that $i(w) \subset \|S\|$ for every $w \in \{0, 1\}^m$. It is clear that for this purpose it is enough to decide $i(w) \subset \|S\|$ one by one, for each $w \in \{0, 1\}^m$, in a uniformly sized quadratic space. So $\mathcal{B}$ has to answer $(w \sqsubset S)$, for each $w \in \{0, 1\}^m$.

We give the recursive algorithm in a self-explaining pseudo-code in which w, $w_1, \ldots, w_9$ denote (i-codes of) m-elementary subintervals while K, K_1, K_2 denote prefixes of S. For the sake of simplicity we write $op(K)$ for $(K_{|K|})_1$ and $arg_{j-1}(K)$ for the $(K_{|K|})_j$-length prefix of K if $j \in \{2, 3\}$. Let $op(K)$ be $FIRSTHALF$ if $K = (FIRSTHALF)$. We omit conditions on some of the cases, since there can be constructed analogously to the cases given. Further, we exclude the case $\lhd PRODUCT$ due to lack of space. To compensate for that, we include the case of $\prec PRODUCT$ which is no less complex.

We establish the following recursive conditions on $\sqsubset$, $\lhd$ and $\prec$.

$(w \sqsubset K) =$
$(K_1, K_2) := (arg_1(K), arg_2(K))$;
case $op(K)$

$=FIRSTHALF \rightarrow$ return $(first\_character\_of(w) = 0)$;
$=NOT \rightarrow$ return the negation of $w \sqsubset K_1$;
$=AND \rightarrow$ return the conjunction of $w \sqsubset K_1$ and $w \sqsubset K_2$;
$=LSHIFT \rightarrow$
$(w_1, w_2) := 0^{|w|} \prec K_2$,
if $w_2 = 1^{|w|}$ then return $(w \sqsubset K_1)$,
% not a real shift, $\|K_2\| = \emptyset$
if $w_2 = \lambda$ then $w_2 := 0^{|w|}$ else $w_2 := succ(w_2)$,
% now w_2 is the first m-elementary subinterval
% included in $\|K_2\|$
$(w_3, w_4) := w_2 \lhd K_2$,
% the first component of K_2 starts with w_3
% and ends with w_4
$w_5 := rotate\_right(w, w_3, w_4)$,
if $w < w_5$ then return $rotate\_right(w) \sqsubset K_1$ else $FALSE$.
% $RSHIFT$ is slightly different since
% it is cyclic
$=PRODUCT \rightarrow$
$(w_1, w_2) := w \lhd K_2$,
if (w_1, w_2) is empty then return $FALSE$,
% by Statement 22, the number of
% $|w|$-elementary subintervals is even
return $(w < upper\_center(w_1, w_2))$.
% remember K_1=$FIRSTHALF$

$(w \lhd K) =$
$(K_1, K_2) := (arg_1(K), arg_2(K))$;
case $op(K)$
$=FIRSTHALF \rightarrow$
if $first\_character\_of(w) = 1$ then
return $(0^{|w|}, \lambda)$
else return $(0^{|w|}, 01^{|w|-1})$;
$=NOT \rightarrow$ return $w \prec K_1$;
$=OR \rightarrow$
$(w_1, w_2) := w \lhd K_1$,
$(w_3, w_4) := w \lhd K_2$,
if $w_2 = \lambda$ then return (w_3, w_4)
else if $w_4 = \lambda$ then return (w_1, w_2)
else return $(min(w_1, w_3), max(w_2, w_4))$;
$=LSHIFT \rightarrow$
$(w_1, w_2) := 0^{|w|} \prec K_2$,
if $w_2 = 1^{|w|}$ then return $(w \sqsubset K_1)$,
% not a real shift, $\|K_2\| = \emptyset$
if $w_2 = \lambda$ then $w_2 := 0^{|w|}$ else $w_2 := succ(w_2)$,
$(w_3, w_4) := w_2 \lhd K_2$,

$w_5 := rotate\_right(w, (w_3, w_4))$,
if $w_5 < w$ then return (w, λ),
% w `is shifted out from` $[0,1)$ `by` $LShift(K_1, K_2)$
$(w_6, w_7) := w_5 \lhd K_2$,
if $w_7 = \lambda$ then return (w, λ),
$w_8 := rotate\_left(w_6, (w_3, w_4))$,
if $w_6 < w_8$ then $w_8 := 0^{|w|}$,
$(w_9, w_1 0) := (w_8, rotate\_left(w_7, (w_3, w_4)))$,
return $(w_9, w_1 0)$.
% `The idea is to move our interval right, find`
% `out` $\lhd K_1$ `and transform it back to the left`

$(w \prec K) =$
$K_1 := arg_1(K)$,
case $op(K) = STAR \rightarrow$

$(K_1, K_2) := (arg_1(K), arg_2(K))$,
$(w_1, w_2) := w \lhd K_2$,
if (w_1, w_2) is empty then
$(w_3, w_4) := w \prec K_2$,
if $w_3 = 0^{|w|}$ then return $(0^{|w|}, w_4)$,
else $(w_5, w_6) := w \lhd pred(w_3)$,
return $(upper\_center(w_5, w_6), w_4)$,
else % `the case when` $w \lhd K_2$ `is nonempty`
$w_3 := upper\_center(w_1, w_2)$,
if $w < w_3$ then return (w, λ),
% $w \sqsubset FIRSTHALF * K_2$
else
if $w_2 = 1^{|w|}$ then return (w_3, w_2),
% (w_1, w_2) `is the last component`
$(w_8, w_9) := succ(w_2) \prec K_2$,
% (w_8, w_9) `is the next component of` $\neg K_2$
return (w_3, w_9).

The given set of recursive conditions describes a terminating recursive algorithm. This can be shown by observing that each recursive call operates on a shorter computation sequence and that the cases of $FIRSTHALF$ are directly given. The correctness of the conditions can be proved by examining the various cases.

Unfortunately, the existence of a recursive algorithm deciding a problem guarantees only its solvability in exponential time. Hence we have to proceed further. We equip this recursive algorithm with a back-track type control. The memory use of the resulting equipped algorithm is quadratic in the input interval-valued computation sequence S. The expression $c \cdot |S| \cdot bitheight(S) \leq$

$c \cdot |S|^2$ describes a sufficient space limit. First we realize that the non-recursive functions are all computable in linear space. To carry out these computations the same memory can always be recycled.

For the organization of back-track type control, the algorithm stores the following data additionally to the input computational sequence.

- ◇ An integer $j \leq log|S|$ stores which prefix of S is actually under processing;
- ◇ for each prefix of S, the index of its caller prefix is stored;
- ◇ for each prefixes of S, the the actual task is stored by a word of length $bitheight(S)$ and an element of $\{\sqsubset, \lhd, \prec\}$;
- ◇ for each prefixes of S, the whole cumulative information that is needed to answer the actual task is stored.

This amount of data fits into the mentioned quadratic space since no description of the gathered information per prefix (*local description of the process of the stored task*) exceeds the size $10 * bitheight(S)$. This can be proved by examining the various cases. We give these local descriptions only in one case when the actual task is $(w, \lhd)$ on a prefix K whose last operation is $LSHIFT$. It is clear, that the full description fits into the given space limit. We use the concept *anti-component* containing w in the following sense: the component of the complement set containing w.

The cumulative information about the stored task can be:

1 There is no information about $w \lhd K$ yet.
2 The anti-component around $0^{|w|}$ is already known and it is (w_1, w_2).
3 In addition to 2, it is known that the anti-component is the whole [0,1).
4 In addition to 2, it has turned out that the anti-component of $0^{|w|}$ is empty.
5 In addition to 2, the anti-component of $0^{|w|}$ is neither empty nor the whole [0,1).
6 In addition to 2 the values (w_3, w_4) are known (they determine the first component of $arg_2(K)$.
. and so on ...
. at last, the the answer is known, it is stored in (w_9, w_{10}).

The notion of local descriptions can be described in a more formal manner. We introduce a relation called *local comparison* between the local descriptions of the states of the computation at the same stored task, based on their amount of gathered information. The local comparison is a partial ordering with two (in case of a task of type $w \sqsubset K$) or one (in the other two cases) maximal element(s). The maximal element(s) belong(s) to the finished, answered task. A *global description* for a state of a computation of $\mathcal{B}$ for S is a sequence $(L_1, \ldots, L_n)$ where each L_i is a local description belonging to $S \rightarrow j$ if $i \in \{1, \ldots, n\}$ or L_i is $\emptyset$ satisfying that $L_i = \emptyset$ and $j < i$ always imply $L_j = \emptyset$.

The execution of the algorithm (equipped by the back-track type control) is as follows. While it works on the task of the actual prefix, there are two possible types of steps. If the answer to the actual task is already known, then the actual task is terminated, the answer returns to the caller prefix. Another possible step is to gather further information to answer the actual task. This is done by calling another task belonging to a prefix of a less index. Practically, it means that we take a step in the execution in the part of the algorithm answering the actual task. This organization guarantees that at most one task has to be stored per prefix. Every task has to be executed as many times as it is called.

One can observe that if the control goes back to the caller prefix then the global amount of gathered information is strictly grows, in the following sense. If G_1 and G_2 are two global descriptions then $G_1 < G_2$ if and only if there exists a positive index $j < |S|$ such that G_1 and G_2 agree on $S_{j+1}, \ldots, S_{|S|}$, G_1 and G_2 have the same actual tasks at S_j but G_2 has more information about it. Intuitively speaking, G_2 is closer to answering of the original question than G_1. We can ascertain that if the actual task finishes then the caller's information will increase. So, in this sense, the global amount of gathered information is always – at each return to the caller – strictly increasing. At the same time, it has an upper bound, since we know the answer to the original question $w \sqsubset S$. Earlier we have established that the algorithm always halts. Moreover, it terminates with the answer to the original question.

The previous arguments complete the proof of Theorem 21.

5 The place of interval-valued computations among new computing paradigms

In this section we recall some unconventional ways of computations and their connections to our system. We do not want to intend to overview all important paradigms known in the literature.

It is a popular research tradition to develop computing paradigms which go beyond the *computational or tractability barriers* of Turing machines and so, Neumann-type computers or other, recently implemented computing devices. In the first case, these new paradigms (often named hypercomputations) would break the resistance of the Church–Turing thesis. For a state-of-art discussion of this direction, see e.g. a recent special issue of Appl. Math. and Computation on hypercomputations ([4]) or [10].

An equally important idea is to look for new paradigms of computation which are intended to tackle intractable (say NP-complete or more complex) prob-

lems. Several branches of unconvential computing paradigms have been rapidly developed in the last 10–15 years. For example, DNA-computing and membrane computing are based on *massive parallelism* observable in nature. In these two models, data is represented by discrete words ([11], [12]). Quantum computing promises also faster solutions to classically hard problems ([16]). These computations are parallel ones opposite to the Neumann-Turing type computations.

Another way is to employ *analog computations* in which data is represented in a non-discrete, continuous form. For example, analog recursive neural networks show higher practical computational performance in some image processing tasks than digital computers ([14]). In [17], analog recurrent neural networks are proved to have computing capabilities above Turing-machines – allowing arbitrary-precision calculations with real numbers. Also in the Blum–Shub–Smale computation model ([3]), computations work on arbitrarily precise real numbers.

Analogic wave computers are introduced in [15]. They work on analog data represented by waves (e.g., in 2-dimensional case, by continuous flows of real-valued images). An *optical model* for analog computing appeared in article [19], in which data is stored in 2-dimensional complex-valued images of arbitrary resolution, and the allowed operators on these images are inspired of the theory of Fourier optics. Important features of this approach are allowing unlimited magnification and shrinking on continuous space data but not allowing test-and-branching control.

The interval-valued computing system is another discrete time / continuous space computational model. It works on 1-dimensional continuous data. The product operator is similar to the shrinking operator in optical computing and it makes it possible to produce unlimited high resolution continuous data, as in [19]. No test-and-branching operator is allowed, that is also a similarity to the optical model.

This model is a natural computational extension of the topic "Reasoning by generalized intervals", as well. Our set of interval-values are closely related to the notion of generalized intervals of the paper [2], see also [6] and [9]. The selection of computational operations on interval-values are motivated above. It was explained in [7] that this set of operations is enough to compute all the tasks of digital computers. In our formalisation this fact reads as follows: every recursive language can be decided by an interval-valued computation. (See Fact 9.)

6 Concluding remarks and open problems

We have proved the solvability of a $PSPACE$-complete problem by a linear interval-valued computation, furthermore, that all the problems in $PSPACE$ can be decided by a restricted polynomial interval-valued computation. Moreover, it is shown that the reverse direction also holds, that is, $PSPACE$ is exhausted by languages decidable by such computations. We can also describe our results in terms of a $PSPACE$-completeness: the equality problem of the closed terms of the structure $(\mathbb{V}_0, \cap, \overline{\|S\|}, Lshift, Rshift, PbF, Firsthalf)$ is $PSPACE$-complete, where PbF is is a unary operation Product by $Firsthalf$ and $Firsthalf$ is the only constant. This approach leads to new questions - how to describe higher logical theories of this structure, for example, is its set of equalities or its first-order theory is decidable?

If we do not restrict the product operator, by the method of Section 4 we can prove that the class of languages decidable by polynomial interval-valued computations is included in $EXPSPACE$. We do not know whether equality holds.

We have formalized the notion of general and linear interval-valued computations. Our definition is quite specialized – it works only on specific (cf. Definition 2 interval-values. Our next question is, what happens if one modifies and generalizes the concept of interval-valued computations – for example, when $\left[0, \frac{1}{2}\right)$ is not the only starting point, or when other operations are also considered.

Another model should be worked out and analyzed where we let the interval-valued mechanism work more, in the following sense. A digital-to-(interval-value) converter translates the input into an interval-value, and then interval-values are processed by the presented network-style computations.

Acknowledgements

This work has been supported by the grant of the Hungarian National Foundation for Scientific Research OTKA F043090, T049409 and T043242, by a grant of the Hungarian Ministry of Education and by the Öveges programme of the Agency for Research Fund Management and Research Exploitation (KPI) and National Office for Research and Technology 

. The authors are grateful to Tamás Mihálydeák and to the anonymous referee for his/her valuable remarks.

References

[1] J. Allen, Maintaing information about temporal intervals, Communications of the ACM, 26(1983):832–843.

[2] P. Balbiani, J.-F. Condotta, L. Farinas del Cerro, A. Osmani, Reasoning about generalized intervals, in: F. Giunchiglia (ed), Artificial Intelligence: Methodology, Systems and Applications, Lecture Notes in Artificial Intelligence 1480(1998): 50–61, Springer.

[3] L. Blum, M. Shub, S. Smale, On a theory of computation and complexity over the real numbers, Bull. AMS (New Series) 21/1(1989):1–46.

[4] F. A. Doria, J. F. Costa, (editors) Special issue on hypercomputation, Appl. Math. and Computation 178/1(2006).

[5] M. Hogarth, Does general relativity allow an observer to view an eternity in a finite time?, Foundations of Physics Letters 5(1992):173–181.

[6] Z. Kulpa, A diagrammatic approach to investigate interval relations, Journal of Visual Languages and Computing, 17(2006): 466-502.

[7] B. Nagy, An Interval-valued Computing Device, in: Computability in Europe 2005: New Computational Paradigms, (eds. S. B. Cooper, B. Löwe, L. Torenvliet), ILLC Publications no. X-2005-01, Amsterdam, pp. 166–177.

[8] B. Nagy, S. Vályi, Solving a PSPACE-complete problem by a linear interval-valued computation, in: Proc. of Conf. "Computability in Europe 2006: Logical Approaches to Computational Barriers" CiE2006, Swansea, Report no. CSR-7-2006, (eds. A. Beckmann, U. Berger, B. Löwe and J.F. Tucker).

[9] B. Nagy, Reasoning by Intervals, 4th International Conference on the Theory and Applications of Diagrams, Stanford(CA), USA, 2006, pp. 145-147. (LNCS-LNAI 4045.)

[10] I. Németi, G. Etesi, Non-Turing computations via Malament–Hogarth spacetimes, International Journal of Theoretical Physics 41(2002):341–370.

[11] Gh. Paun, G. Rozenberg, A. Salomaa, DNA Computing - New Computing Paradigms, Springer, 1998.

[12] Gh. Paun, Membrane Computing. An Introduction, Springer, 2002.

[13] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[14] T. Roska and L. O. Chua, The CNN Universal Machine: An analogic array computer, IEEE Transactions on Circuits and Systems-II, 40(1993):163-173.

[15] T.Roska, Computational and Computer Complexity of Analogic Cellular Wave Computers, Journal of Circuits, Systems and Computers, 12(2003):539-562.

[16] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Comput. 26(1997):1484-1509.

[17] H. T. Siegelmann, E. D. Sontag, Analog computation via neural networks, Theoretical Computer Science, 131/2(1994):331–360.

[18] Tanenbaum, Andrew S.: Structured Computer Organization, Prentice-Hall, 1984.

[19] D. Woods, T. Naughton, An optical model of computation, Theoretical Computer Science 334(2005):227-258.