Electronic Colloquium on Computational Complexity, Report No. 29 (2015)

Inherent Logic and Complexity

Stanislav Žák[*]

Institute of Computer Science, Academy of Sciences of the Czech Republic,
P. O. Box 5, 182 07 Prague 8, Czech Republic, `stan@cs.cas.cz`

Abstract. The old intuitive question "what does the machine think" at different stages of its computation is examined. Our paper is based on the formal definitions and results which are collected in the branching program theory around the intuitive question "what does the program know about the contents of the input bits" [1],[2],[3],[4],[5].

We further develop these results above and we present a formal counterpart of the intuitive notion "what does the program think " at different stages of its computation on the processed input word. The definition is constructed as the logical consequences of the definition of the knowledge about the contents of input bits above.

Our formal definition is in a good relation to the world of intuitive ideas. We prove the theorem saying that the programs which are allowed to compute (think) in a more sophisticated way can compute more effectively. We also demonstrate an example that for some programs a small enrichment of their inherent logical possibilities implies a dramatic complexity drop. So, our definition lives up to the expectations inspired by intuition.

The present paper opens a large field of possible investigations of relations between logic on one hand and complexity on the other hand.

Key words: branching programs, complexity, logic

[*] S.Ž.'s research was partially supported by the projects GA ČR P202/10/1333 and RVO: 67985807.

ISSN 1433-8092

1 Introduction

In the last century many computational models (e.g Turing machines, circuits, neural networks etc.) were introduced. Whenever a new computational device was defined the following intuitive question was insistently present: What does the computational device mean (know) about the processed input at different stages of its computation? Briefly, how does the device think? This intuitive question is a very simple one and primarily it is a very burning one for our imagination. To formalize this simple intuitive question (and the corresponding answer, of course) seems to be a very difficult problem on one hand but on the other hand it is a very fascinating challenge.
This question has become even more acute in the early 1980s when branching programs as a counterpart of memory limited Turing machines were introduced. Due to the simplicity of their definition the solution of the question "what does the branching program think at different places along different computations?" seemed to be reachable. Within our non-mathematical world of imagination the tests of variables (input bits) along the computation in a branching program are considered as the sources of atomic information. This is an indisputable axiom. We can intuitively work with the idea that along each computation some atomic information is acquired (by tests of variables), transformed to a (partially) global information and stage by stage forgotten, and finally the resulting global knowledge YES/NO remains at the end of the computation.
Since the general question to define a formal counterpart of the intuitive notion "what does the program think" was too difficult a more simple question to find a formal counterpart of the intuitive notion "what does the program think (know) about the contents of the input bits at different stages of the computation" was investigated [1], [2]. We have defined a mathematical construct - so-called window - which corresponds to intuitive knowledge about the input bits. If n is the length of inputs words then the window is a word of length n over three letter alphabet 0,1,+ where "+" stands for "at this moment unknown". Along the computation to each node and to each edge a corresponding window (depending on the computation in question) is assigned. The windows are variable along the computation. The key point of the definition of window is based on the intuitive turn that "the bit which will be tested in the future (i. e. below a node or below an edge) is actually (now) unknown (at this node or at this edge)." Consequently, at one node the different computations may have different knowledge about the input bits since in the future below this node they may test different bits. So, in our approach the idea that any node represents the same information for each computation reaching it is false. This is the key difference with the well-known approaches where information or knowledge is simply represented by a node alone (in a graph of computations).
The definition of window has allowed us to formulate a lower bound method applicable to the general branching programs [2]. Moreover, we have been able to define a class of restricted branching programs much larger than the class of read-once programs, e.g. using these programs it is possible to compute many "witness" functions superpolynomially hard for different classes of restricted pro-

grams within polynomial size. By using our lower bound method we have proven a superpolynomial lower bound on a Boolean function for this very large class of programs [3], [5]. Further, some other lower bounds are proven [4]. This demonstrates the strength of the developed method (and the usefulness of our intuitive questions, of course).
In the present research we further follow our original intuition and we make the second step in defining thinking of the program. In Section 4 we expand our previous formal definition "what does the machine know about the input bits" to the formal definition what "does the machine really mean" at different stages of its computation. The new definition is simply a logic scaffolding over windows. (The fact that the knowledge is not given by the node alone remains valid.) In Section 5 we prove the key theorem which confirms the intuitive idea that the programs which are allowed to think more sophistically (i.e. with more logical possibilities) can compute more effectively (i. e. with less need of sources). Moreover, in Section 6 we demonstrate an example where a small enlarging in inherent logic possibilities of the programs in question implies a dramatic drop in complexity.
The theorem opens a large field of questions about the connections between the logic on one hand and the complexity on the other hand. We hope that we are at the starting points of many possible research efforts. Some such possible starting points are indicated in Section 7.

2 Technical Preliminaries

By a branching program (b.p.) P (over binary inputs of length n) we mean a finite, oriented, acyclic graph with one source (in-degree = 0) where all nodes have out-degree = 2 (so-called branching or inner nodes) or out-degree = 0 (so-called sinks). The branching nodes are labeled by variables x_i, $i = 1, ..., n$, one out-going edge is labeled by 0 and the other by 1, the sinks are labeled by 0 or by 1. If a node v is labeled by x_i we say that x_i is tested at v. For an input $a = a_1...a_n \in \{0,1\}^n$ by the computation on a ($comp(a)$) we mean the sequence of nodes (and edges) starting at the source of P and ending in a sink. In the sequence, for each i, $1 \leq i \leq n$, at any node with label x_i the next node is pointed by the edge with label a_i. By the length of a computation we mean the number of its inner nodes.
For a node $v \in comp(a)$ we say that a reaches v. If a and b reach v and immediately below v they reach different nodes we say that $comp(a)$ and $comp(b)$ diverge at v (or shortly a and b diverge at v). Similarly for more than two inputs. If $comp(a)$ has a common part with a path p in P we say that a follows p (in this part). P computes function f_P which on each $a \in \{0,1\}^n$ outputs the label of the sink reached by a. We say that P computes in time $t(n)$ if its each computation is of the length at most $t(n)$.
A special case of b.p. with in-degree = 1 in each node (with exception of the source) is called decision tree. Another well-known class of restricted b.p.s are

so-called read-once branching programs in which along each computation each variable is tested once at most. Read-once b.p.s compute in time n, of course.
By a distribution we mean any mapping D of a subset of $\{0,1\}^n$ to (the set of nodes of) P with the property that for each a $D(a)$ is a node of $comp(a)$ ($D(a) \in comp(a)$). The class of the distribution at node v is the set of all as mapped to v. (Similarly, we can work with distribution to edges.)
Let v be a node of P. By the tree T_v unfolded in v according to P we mean the decision tree the branches of which are given by the paths in P starting at v and ending at sinks. Let A be a set of some (not necessarily all) inputs reaching v. By the tree $T_{v,A}$ unfolded in v according to P with respect to A we mean the decision tree which results from tree T_v after application of the following operations:

a) From T_v we omit all branches which are not followed by any input from A.

b) Each edge pointing to a node with out-degree $= 1$ (after a)) is repointed to its successor.

By the size of P we mean the number of its nodes. By the complexity of a Boolean function f we mean the size of the minimal b.p.s computing f. It is a well-known fact that superpolynomial lower bound on the size of b.p.s implies superlogarithmic lower bound for space complexity of Turing machines [6].

3 Windows

We introduce the key notion of windows. We start with some extensive comments.

For a given b.p. and for a given input a of length n we want to catch the remembered information concerning the contents of the bits of a along $comp(a)$ at its each node and at its each edge. We proceed in such a way that we assign a word w of length n over the ternary alphabet $\{0,1,+\}$ to each node and to each edge of $comp(a)$. Each such w will have the following property: for each i, $1 \leq i \leq n$, $w_i = a_i$ *or* $w_i = +$. The sign "+" will be called "a cross", and on the intuitive level of reasoning it will stand for "unknown" or "forgotten". The assigned word w will be called the window on a at the respective node or at the respective edge of $comp(a)$. By its length we shall mean the number of its non-crossed bits. On the intuitive level of our reasoning these non-crossed bits will represent the remembered information. Small warning: the window at a node or at an edge will also depend on other inputs which also reach this node or this edge.

Before creating the formal definition of windows we have two simple ideas at our disposal. Firstly, on the intuitive level, a test in b.p. means remembering (the content of) one bit. Hence, our next formal definition of windows should respect the rule "one test, (exactly) one cross is removed". Secondly, on the intuitive level it is difficult to say what is "remembered" but it is easy to say the complementary thing what is "forgotten" or "unknown". Intuitively, we see that the bit which will be tested in the future (below the node or below the edge in question) is an unknown or forgotten one now (at this node or at this edge), and it should be a crossed one, now. Our intuition is mirrored in the next formal definition of windows.

Definition 1. *Let P be a branching program, v be its node. Let A be a set of some (not necessarily all) inputs reaching v. From v we develop a tree $T_{v,A}$ according to P with respect to A.*

For each $a \in A$ we define the window $w(a,v,A)$ on a at v with respect to A in such a way that $w(a,v,A)_i = +$ if and only if in $T_{v,A}$ there is a test on bit i along the branch followed by a or there is another input $b \in A$ following the same branch as a does until the sink such that a and b differ on i. On the other -non-crossed- bits $w(a,v,A)$ equals a.

The length of a window is the number of its non-crossed bits.

The window $w(a,v,A)$ is said to be a natural one iff A is the set of all inputs reaching v.

Comments.

i) If (in the definition) we replace "node v" by "edge e" we obtain the window assigned to the edge e.

ii) For each a in a given set A comparing the window on a at v with respect to A and the window on a at an out-going edge e leaving v with respect to the subset of A corresponding to e we see that the rule "one test, (exactly) one cross is removed" mentioned above is satisfied.

iii) It is clear that the simple thing holds: "The larger A, the larger number of branches in the tree, the larger number of crosses, the shorter windows".

iv) For read-once branching programs the window is always given by the node. The idea to consider the programs in which the windows at one node may differ in a moderate way was the starting point for papers [2],[3], [5].

Another confirmation of our intuition is given by a small theorem in [1] saying that for each (general) branching program computing symmetric words it holds that during the computation on such a word each pair of symmetric positions must be non-crossed at least in one natural window (=at the same moment). In other words, branching programs computing symmetric words must compute in a human-like way.

For the theory of windows the following theorem [2], [3] is very important.

Theorem 1. *Let P be a branching program and A be a set of inputs of length n distributed in nodes $v_1, ...v_r$ of P. Let A_1, ... A_r be the classes of this distribution. Then, log_2 (size of P) $\geq log_2\, r \geq log_2\, |A| - n + avelw$ where $avelw$ is the average* *length of windows on inputs from A each at corresponding v_i with respect to A_i, $i = 1, \dots r$.*

Theorem 1 confirms our intuition that remembering a lot of information about many inputs requires a large memory, i. e. a large branching program. We see that our construct -windows- is closely related to our intuition. Moreover, Theorem 1 gives a general method for proving large lower bounds. For proving a lower bound for a Boolean function it suffices to prove that on any b.p. this function requires large windows on many inputs. We shall see an application of Theorem 1 in the proof of Theorem 4 in Section 6.

For the proof of theorem we use the following lemma.

Lemma 1. *[5] Let us have r binary trees. Let l be the average length of their branches and S be the sum of (the numbers of) their leaves. Then, $l \geq log_2\ S - log_2\ r$.*

Proof. Let us take the classes $A_1, ..., A_r$ distributed to the nodes $v_1, ..., v_r$. For each i, $1 \leq i \leq r$, in v_i let us develop the tree T_{v_i,A_i} according to P. In its each sink with at least two inputs reaching it we add an appropriate decision tree such that each sink of the resulting tree T'_i is reached by exactly one input from A_i. We obtain r binary trees and we apply lemma above. Let l, S be as in lemma. We have $log_2(size\ of\ P) \geq log_2\ r \geq log_2\ S - l \geq log_2\ |A| - l \geq log_2\ |A| - n + avelw$. $\square$

We see that windows and trees are complementary in some sense. At each node long windows are the same as short branches in the respective tree and vice versa.

4 Deductive systems for branching programs

By a deductive system (briefly system) S we mean any quadruple $\{O, Pr, Form, D\}$ where O is a set of objects, O always includes special objects $o_1, ..., o_n$ which correspond to the input bits. Pr is a set of k-ary predicates on O, $k = 0, 1, \dots$. Pr always includes unary predicates $0, 1$ applicable to the objects o_i for $i = 1, ..., n$ (formula $0(o_i)$ corresponds to the situation when the i-th bit is non-crossed and has value 0, similarly for $1(o_i)$). In the sequel we shall see how objects o_is and predicates $0, 1$ are used for intake of atomic information concerning input bits to the system of logical reasoning. Further, Pr always includes zero-predicates F ("false"),T ("true"). $Form$ is a finite set of admissible formulas over O and Pr. D is a set of deductive rules over formulas. Each deductive rule $r \in D$ is a partial mapping from $Form^{k_r}$ to $Form$.

Let P be a branching program and a be an input, $a \in \{0, 1\}^n$. To each node v of $comp(a)$ we shall assign a sequence $V(a, v)$ of formulas from $Form$. Similarly to each edge e of $comp(a)$ we shall assign a sequence $V(a, e)$. In any case in

$V(a,v)$ or in $V(a,e)$ the actual natural windows will be described by predicates $0, 1$ on objects $o_1, ..., o_n$.

The process of assigning the sequences $V(a,v)$ and $V(a,e)$ to each node and to each edge of $comp(a)$ starts in the source of the program in question, $V(a, source) =_{df} \emptyset$.

Let us have a node v testing a bit i with two outgoing edges e_0, e_1 (labeled by $0, 1$, resp.). Let a be an input such that $v \in comp(a)$. Let $V(a,v)$ be given. Let moreover $e_0 \in comp(a)$. Then, we define $V(a,e_0)$ as the set of all formulas derivable from formulas in $V(a,v) \cup \{0(o_i)\}$ by rules from D.

Let us have an edge e ending in a node v, $e \in comp(a)$. We define $V(a,v)$ as follows:

$V(a,v)$ will be a subsequence of $V(a,e)$.

Let $B_{a,v}$ be the set of inputs b which reach v via an edge e_b and which follow a from v to the same sink (following the same path).

Each $w \in V(a,e)$ such that $w \in V(b,e_b)$ for each $b \in B_{a,v}$ remains in $V(a,v)$. Further in $V(a,v)$ are all formulas $0(o_i), 1(o_i)$ for $i = 1, ..., n$ which define the window on a at v. $V(a,v)$ is completed by adding all formulas derivable from the previous formulas of both types (by rules from D).

We see that each system assigns the sets of formulas depending on computation to each node and to each edge in any branching program. The problem is that the system can derive predicates F, T at the end of computations which do not correspond to the labels $1, 0$ of the reached sinks.

Definition 2. *Let P be a program and S be a deductive system. We say that S is P-sound iff*

for each sink of P with label 0 (1, resp.) S never derives T (F, resp.) for the computation on any input.

Definition 3. *Let f be a Boolean function and S be a system. We say that S is f-sound iff S is P-sound for each P computing f.*

Theorem 2. *Let f be a Boolean function and let DT be a full decision tree computing f. Let S be a DT-sound system. Then S is an f-sound system.*

Proof. Let P be a program computing f and let a be an input. The sequence of derived formulas at the end of $comp(a)$ in P is a part of the set of formulas derived at the end of $comp(a)$ in DT. Hence, it does not contain wrong F or wrong T. $\square$

5 Compatible systems and complexity

In the next definition we introduce the notion of deductive systems compatible with a branching program, which is a basic notion for our paper. Within the imaginative part of our reasoning (our intuition) we consider compatible systems as entities which give us to understand the way how the program does compute (think), how its inherent logic works.

Definition 4. *Let P be a branching program and let S be a P-sound system. We say that P and S are (mutually) compatible if in $V(a, s)$ for each sink s of P and for each input a reaching s predicate T (F, resp.) is derived iff the label of s is 1 (0, resp.).*

Lemma 2. *Let f be a Boolean function. Then, there is a branching program P computing f and there is a system S such that S is compatible with P.*

Proof. Let P be a decision tree for f. Let S be a system with predicates $0, 1, F, T$. We see that for each input word there is a unique corresponding branch of P. At the end of this branch, a sequence w of formulas is collected which correspond to the window on the input word. Since P is a decision tree this window is of length n, all bits are non-crossed. Therefore, it suffices to include appropriate deductive rules w/T or w/F to D. $\square$

Definition 5. *Let $S_1 = (O_1, Pr_1, Form_1, D_1)$ and $S_2 = (O_2, Pr_2, Form_2, D_2)$ be systems. If $O_1 \subseteq O_2$, $Pr_1 \subseteq Pr_2$, $Form_1 \subseteq Form_2$ and $D_1 \subseteq D_2$ then we say that S_1 is a part of S_2, $S_1 \sqsubseteq S_2$.*

Lemma 3. *Let P be a program and let S_1, S_2 be systems which are P-sound. If S_1 is compatible with P and $S_1 \sqsubseteq S_2$ then also S_2 is compatible with P.*

Proof. Each proof in S_1 is also in S_2 including the proofs of F and T at sinks. $\square$

Definition 6. *Let f be a Boolean function and S be a system. Then, by S-complexity of f we mean the size of the smallest branching program P computing f and compatible with S if such P exists (otherwise formally S-complexity equals ∞).*

Theorem 3. *Let f be a Boolean function and S_1, S_2 be systems. Let S_2 be P-sound for each P computing f compatible with S_1. If $S_1 \sqsubseteq S_2$ then S_2-complexity of f is not larger than S_1-complexity of f.*

Proof. It suffices to prove that each program P compatible with S_1 is compatible also with S_2. This follows from the lemma above. So, the minimum for S_2-complexity is taken over the same or larger set of programs than in case of S_1-complexity. $\square$

The theorem confirms our intuitive idea that the branching programs which may compute in a more complicated way (i. e. using our terminology which are compatible with a richer deductive system) can compute more effectively, i. e. within a smaller complexity bound. This indicates that our choice of definitions is sound and that our small theory is the desired counterpart of our intuitive ideas. In the next section we demonstrate this fact convincingly. In our example, a small increase in the richness of deductive systems produces a dramatic drop in the need of the computation source (memory).

6 An example: Less logic, more complexity

Let f be the parity function on n input bits. We shall construct a chain $S_1 \sqsupseteq S_2 \sqsupseteq .... \sqsupseteq S_n$ of systems such that for $i < n$ S_{i+1}-complexity of f is larger than the S_{i-1}-complexity of f for all $i < n$. In other words, less logic, more complexity and viceversa.

We define $S_i =_{df} \{O_i, Pr_i, Form_i, D_i\}$ where O_i contains the obligatory objects $\{o_i | i = 1, ..., n\}$ and all subsets of cardinality at least i of the set of input bits, $O_i \supset \{A | A \subseteq \{1, ..., n\}, |A| \geq i\}$.

Pr_i contains the obligatory predicates $\{0, 1, F, T\}$ and the predicates $odd_i, even_i$ applicable to sets of input bits of cardinality at least i.

$Form_i$ contains F, T and all formulas of type $p(o)$ where $p \in Pr_i$, $o \in O_i$ and p is applicable on o.

D_i contains the deduction rules as follows:

Rule I).

For any $o_{j_1}, ..., o_{j_i}$ for any choice of predicates $a_{j_k} = 0$ or $a_{j_k} = 1$ for $k = 1, ..., i$ there is a rule $a_{j_1}(o_{j_1}), \dots , a_{j_i}(o_{j_i})$ / $Par(\{j_1, ..., j_i\})$ where $Par = odd_i$ or $Par = even_i$ according to the parity of the number of 1s in the chain $a_{j_1}, ..., a_{j_i}$.

Rule II).

$Par_L(A), par(o_j)/Par_R(A \cup \{o_j\})$ where A is a set of input bits, $0 < j < n+1$ and $j \notin A$, Par_L is odd_i or $even_i$, par is 0 or 1 and Par_R is odd_i or $even_i$ depending on Par_L, par in the obvious way which is given by the properties of the parity function.

Rule III).

Moreover, in D_i there are two special rules

$odd_i(\{1, ..., n\})/T$ and $even_i(\{1, ..., n\})/F$.

Theorem 4. *For $i \in \{1, ..., n\}$, S_i-complexity of the parity function is at least 2^{i-1}.*

Proof. Let P be any branching program computing parity function which is compatible with S_i. We want to prove that $size(P)$ is at least 2^{i-1}, this will be sufficient. From the compatibility P and S_i follows that at the sink of its computation each input has activated predicates F or T. Hence, during its computation each input has activated predicates odd_i or $even_i$ on the set $\{1, ..., n\}$ (cf. Rule III).

The unique way in which an input may have activated a parity predicate on a set of input bits of cardinality n is such that it has activated this predicate on

the set of cardinality $n - 1$ and used Rule II. Repeatedly till the cardinality i. For each input let us take into account the edge of its computation where the predicates odd_i or $even_i$ are activated on a set of input bits of size i for the first time (cf. Rule I). Left-hand side of this rule is activated only in the case when the predicates $0, 1$ are defined on i input bits. This is the moment where the input has the natural window of length i.
Now we distribute each input a to the edge of its computation where a has the natural window of length i for the first time. The length of windows according to this distribution is of course of length at least i. According to Theorem 1 the number of classes of the distribution and hence of edges in P is at least 2^i. Hence, according to the fact that out-degree of nodes in P is at most 2 we have $size(P)$ that is at least 2^{i-1}. (Here we see how Theorem 1 works in practice.) $\square$

Lemma 4. *For i, $i = 1, ..., n$, there is a program P which computes the parity function and which is compatible with system S_i such that $size(P) \leq 2^i + 2.(n - i + 1)$.*

Proof. P starts as a decision tree of depth i on the first i bits. On the remaining $n - i$ levels P is of width 2. The nodes are arranged in two columns, one column represents the value "even" and the other represents the value "odd". The zero-edge outgoing any node always preserves the column while the one-edge always changes the column.

We see that P indeed computes the parity function and that its size is below the desired bound. It remains to prove that P is compatible with S_i. On the level of leaves of the initial tree of depth i in question we have for the first time derived the formula $Par(A)$ where A is the set of cardinality i and Par is odd_i or $even_i$. Below in two column chain the cardinality of A is increasing step by step, by one in each step (cf. Rule II).

(In each step at the level l of our two columns four edges start which end in two nodes on the next level $l+1$ of columns. Each edge has assigned a formula of type $Par(A)$ which is assigned also in the starting node, and a formula $Par'(A')$ where the cardinality of A' is equal to the cardinality of A plus 1. Small analysis says that in the target nodes only formulas of type $Par'(A')$ are assigned. This is an example of forgetting relatively global information during the computation - cf. the creation of $V(a, v)$ in Section 4.)

On the level of sinks the cardinality of A is n, and, therefore, correct F, T are derived here - cf. Rule III. So, P is compatible with S_i. $\square$

We see that for $i > log\ n$ the upper bound $2^{i-1} + 2.(n - i + 2)$ for S_{i-1}-complexity is less than the lower bound 2^i for S_{i+1}-complexity. Indeed, adding to S_{i+1} the possibility to express parity value for the sets of input bits of cardinality i and $i - 1$ considerably decreases complexity (in comparison with S_{i+1}-complexity).

7 Problems

We have started a research which seems to be new. Hence, we have to perform the initial recognition of our terrain. We are going to do this by formulating a series of questions which indicate some starting points of possible research directions.

1. At this moment we do not know whether for each branching program P there is a system S which is compatible with P.
2. Is there a Boolean function f and a system S compatible with all (minimal) programs computing f?
3. Are there programs P_1, P_2 computing the same function , and systems S_1, S_2 such that S_1 is compatible with P_1 but not with P_2 and vice versa for S_2?
4. Many questions are arising in connection with the quality of the deductive system S on one hand and the respective S-complexity on the other hand for different (types of) functions.
a) How is the S-complexity of a function influenced by the morphology of S? I.e. if different types of objects, different predicates, different way of constructions of formulas and different deductive rules are allowed or forbidden in S?
b) Given a function f and a system S compatible with at least one program computing f. Preserving soundness, we may enlarge S in four directions - we may enlarge either the set of objects, either the set of predicates, either the set of admissible formulas, either the set of deductive rules. Along each direction the S-complexity is monotone non-increasing. The question is when S becomes compatible with a minimal program computing f (and, therefore, the S-complexity of f will be the same as normal complexity).
c) Is it possible to substitute an enlarging of S in one direction (see b)) by an enlarging in another direction?
5. To find some functions f which have simple non-complicated systems compatible with their minimal programs. To classify all such functions.
6. To find a minimal program and the corresponding compatible system for a concrete Boolean function - e.g. s-t connectivity in oriented graphs. The same for functions which figure in proofs of lower bounds in the b.p. theory.
7. Let us have some class of systems. Let us take the class of all programs which are compatible with at least one of the systems in question. In fact, we obtain a restriction on programs. May such a restriction be in some relation with classical restrictions, e. g. read-once branching programs etc.?
8. For a given function f to find a system S compatible with each 1-bp computing f (if S exists). Similarly for other restrictions.

The world of branching programs observed from the point of inherent logic seems to be more beautiful and wild and posing more challenges than the classical theory. And more understandable and, therefore, more human.

References

1. Žák, S.: Information in Computation Structures. Acta polytechnica. Vol. 20, no. 4 (1983), pp. 47-54. ISSN 1210-2709
2. Žák, S.: A Subexponential Lower Bound for Branching Programs Restricted with Regard to Some Semantic Aspects. Electronic Colloquium on Computational Complexity. Report Series 1997. ECCC TR97-50. Trier, 1997, http://www.eccc.uni-trier.de/report/1997/050
3. Jukna, S., Žák, S.: On Branching Programs with Bounded Uncertainty. Automata, Languages and Programming. Proceedings. Berlin : Springer, 1998 - (Larsen, K.; Skyum, S.; Winskel, G.), pp. 259-270 ISBN 3-540-64781-3. - (Lecture Notes in Computer Science. 1443). [ICALP'98 International Colloquium /25./. Aalborg (DK), 13.07.1998-17.07.1998]
4. Jukna, S., Žák, S.: Some Notes on the Information Flow in Read-Once Branching Programs. SOFSEM'2000: Theory and Practice of Informatics. Berlin : Springer, 2000 - (Hlav, V.; Jeffery, K.; Wiedermann, J.), pp. 356-364 ISBN 3-540-41348-0. ISSN 0302-9743. - (Lecture Notes in Computer Science. 1963).
5. Jukna, S., Žák, S.: On Uncertainty versus Size in Branching Programs. Theoretical Computer Science. 290 (2003), pp. 1851-1867.
6. Wegener, I.: Branching Programs and Binary Decisions Diagrams, SIAM Monographs on Discrete Mathematics and Applications, pp. 408, 2000.
7. Žák, S.: A Lower Bound Method for Branching Programs and Its Application. Prague : ICS AS CR, 2012. 19 pp., Technical Report, V-1171

ECCC ISSN 1433-8092

http://eccc.hpi-web.de