Pure Pointer Programs with Iteration

Martin Hofmann and Ulrich Schöpp

Ludwig-Maximilians-Universität München
D-80538 Munich, Germany

Abstract. Many LOGSPACE algorithms are naturally described as programs that operate on a structured input (e.g. a graph), that store in memory only a constant number of pointers (e.g. to graph nodes) and that do not use pointer arithmetic. Such "pure pointer algorithms" thus are a useful abstraction for studying the nature of LOGSPACE-computation.

In this paper we introduce a formal class PURPLE of pure pointer programs and study them on locally ordered graphs. Existing classes of pointer algorithms, such as Jumping Automata on Graphs (JAGs) or Deterministic Transitive Closure (DTC) logic, often exclude simple programs. PURPLE subsumes these classes and allows for a natural representation of many graph algorithms that access the input graph by a constant number of pure pointers. It does so by providing a primitive for iterating an algorithm over all nodes of the input graph in an unspecified order.

Since pointers are given as an abstract data type rather than as binary digits we expect that logarithmic-size worktapes cannot be encoded using pointers as is done, e.g. in totally-ordered DTC logic. We show that this is indeed the case by proving that the property "the number of nodes is a power of two," which is in LOGSPACE, is not representable in PURPLE.

1 Introduction

One of the central open questions in theoretical computer science is whether LOGSPACE equals PTIME and more broadly an estimation of the power of LOGSPACE computation.

While these questions remain as yet inaccessible, one may hope to get some useful insights by studying the expressive power of programming models or logics that are motivated by LOGSPACE but are idealised and thus inherently weaker. Examples of such formalisms that have been proposed in the literature are *Jumping Automata on Graphs* (JAGs) [2] and *Deterministic Transitive Closure (*DTC*) logic* [3]. Both are based on the popular intuition that a LOGSPACE computation on some structure, e.g. a graph, is one that stores only a constant number of graph nodes. Many usual LOGSPACE algorithms obey this intuition and are representable in those formalisms. Interestingly, there are also natural LOGSPACE algorithms that do not fall into this category. Reingold's algorithm for st-connectivity in undirected graphs [13], for example, uses not only a constant number of graph nodes, but also a logarithmic number of boolean variables, which are used to exhaustively search the neighbourhood of nodes up to a logarithmic depth.

Indeed, Cook & Rackoff [2] show that st-connectivity is not computable with JAGs. On the other hand, JAGs cannot compute some other problems either, for which there

M. Kaminski and S. Martini (Eds.): CSL 2008, LNCS 5213, pp. 79–93, 2008.

does exist an algorithm obeying the above intuition, such as a test for acyclicity in graphs. Thus, important though this result is, we cannot see it evidence that "constant number of graph nodes" is strictly weaker than LOGSPACE.

Likewise, DTC logic without a total order on the graph nodes is fairly weak and brittle [8,5], being unable to express properties such as that the input graph has an even number of nodes.

By assuming a total order on the input structure these deficiencies are removed, but DTC logic becomes as strong as LOGSPACE, because the total order can be used to simulate logarithmically sized work tapes [3].

We thus find that neither JAGs nor DTC logic adequately formalise the intuitive concept of "using a constant number of graph variables only." In this paper we introduce a formalism that fills this gap. Our main technical result shows that full LOGSPACE does not enter through the backdoor by some encoding, as is the case if one assumes a total order on the input structure.

One reader remarked that it was known that LOGSPACE is more than "constant number of pointers", but up until the present contribution there was no way of even rigorously formulating such a claim because the existing formalisms are either artificially weak or acquire an artificial strength by using the total order in a "cheating" kind of way.

To give some intuition for the formalism introduced in this paper, let us recall that a JAG is a finite automaton accepting a locally ordered graph (the latter means that the edges emanating from any node are uniquely identified by numbers from 1 to the degree of that node). In addition to its finite state a JAG has a finite (and fixed) number of pebbles that may be placed on graph nodes and may be moved along edges according to the state of the automaton. The automaton can check whether two pebbles lie on the same node and obtains its input in this way. Formally, one may say that the input alphabet of a JAG is the set of equivalence relations on the set of pebbles.

JAGs cannot visit all nodes of the input graphs and therefore are incapable of evaluating DTC formulas with quantifiers. To give a concrete example, the property whether a graph contains a node with a self-loop is not computable with JAGs.

Rather than as an automaton, we may understand a JAG as a `while`-program whose variables are partitioned into two types: boolean variables and graph variables. For boolean variables the usual operations are available, whereas for graph variables one only has equality test and, for each i, a successor operation to move a graph variable along the i-th edge.

Our proposal, which we call PURPLE for "PURe Pointer LanguagE", consists of adding to this programming language theoretic version of JAGs a `forall`-loop construct ($\texttt{forall}\ x\ \texttt{do}\ P$) whose meaning is to set the graph variable x successively to all graph nodes in some arbitrary order and to evaluate the loop body P after each such setting. The important point is that the order is arbitrary and will in general be different each time a `forall`-loop is evaluated. A program computes a function or predicate only if it gives the same (and correct) result for all such orderings.

The `forall`-loop in PURPLE can be used to evaluate first-order quantifiers and thus to encode DTC logic on locally ordered graphs. Moreover, PURPLE is strictly more expressive than that logic. The following PURPLE-program checks whether the input

graph has an even number of nodes: $(b := \mathrm{true};\ \texttt{forall}\ x\ \texttt{do}\ b := \neg b)$. It is known that this property is not expressible in locally-ordered DTC logic, which establishes strictness of the inclusion.

Beside the introduction of PURPLE, the main technical contribution of this paper is a proof that PURPLE is not as powerful as all of logarithmic space and that thus in particular one cannot use the `forall`-loop to somehow simulate counting, as one can in totally ordered DTC logic [3]. We do this by showing that the property "the number of nodes is a power of two" is not computable in PURPLE. We believe that st-connectivity in undirected graphs is not computable in PURPLE either.

In order to justify the naturality of PURPLE we can invoke, besides the fact that formulas of locally-ordered DTC logic may be easily evaluated, the fact that iterations over elements of a data structure in an unspecified order are a common programming pattern, being made available e.g. in the Java library for the representation of sets as trees or hash maps. The Java API for the `iterator` method in the interface `Set` or its implementation `HashSet` says that "the elements are returned in no particular order".

Efficient implementations of such data structures, e.g. as splay trees, will use a different order of iteration even if the contents of the data structure are the same. Thus, a client program should not depend upon the order of iteration. A spin-off of PURPLE is a rigorous formalisation of this independence.

This research was supported by the DFG project *programming language aspects of sublinear space complexity* (ProPlatz).

2 Pointer Structures

We define the class of structures that serve as input to pure pointer programs.

Definition 1. *Let $L = \{l_1, \ldots, l_n\}$ be a finite set of operation labels. A* pointer structure *on L, an L-model for short, is a set U with n unary functions $l_1, \ldots, l_n \colon U \to U$.*

An L-model can be viewed as the current state of a program with pointers pointing uniformly to records whose fields are labelled $\{l_1, \ldots, l_n\}$. For example, if L is $\{car, cdr\}$ then an L-model is a heap layout of a LISP-machine. We show in the next section how various kinds of graphs can be represented as L-models.

A *homomorphism* $\sigma \colon U \to U'$ between L-models U and U' is a function $\sigma \colon U \to U'$ such that $l(\sigma(x)) = \sigma(l(x))$ holds for all $l \in L$ and $x \in U$. A bijective homomorphism is called an *isomorphism*.

3 Pure Pointer Programs

Pure pointer programs take L-models as input. Unlike general programs with pointers they are not permitted to modify the input, but only to inspect it using a constant number of variables holding elements of U. In addition, pure pointer programs have a finite local state represented by a constant number of boolean variables. The pointer language PURPLE is parameterised by a finite set of *operation labels* $L = \{l_1, \ldots, l_n\}$.

Terms. There are two types of terms, one for boolean values and one for pointers into the universe U. Fix countably infinite sets Vars and Vars_B of pointer variables and

boolean variables. We make the convention that x, y denote pointer variables and b, c denote boolean variables. The terms are then generated by the grammars below.

$$t^B ::= \text{true} \mid \text{false} \mid b \mid \neg t^B \mid t^B \wedge t^B \mid t^B \vee t^B \mid t^U = t^U$$
$$t^U ::= x \mid t^U.l_1 \mid \cdots \mid t^U.l_n$$

The intention is that pointer terms denote elements of the universe U of an L-model and that the term $x.l$ denotes the result of applying the unary operation l in this model to x. The only direct observation about pointers is the equality test $t = t'$.

Programs. The set of PURPLE programs is defined by the following grammar.

$$P ::= \texttt{skip} \mid P; P \mid x := t^U \mid b := t^B \mid \texttt{if}\ t^B\ \texttt{then}\ P\ \texttt{else}\ P \\ \mid \texttt{while}\ t^B\ \texttt{do}\ P \mid \texttt{forall}\ x\ \texttt{do}\ P$$

We abbreviate $(\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip})$ by $(\texttt{if}\ b\ \texttt{then}\ P)$. The intended behaviour of $(\texttt{forall}\ x\ \texttt{do}\ P)$ is that the pointer variable x iterates over U in some unspecified order, visiting each element exactly once, and P is executed after each setting of x.

On certain classes of L-models the power of PURPLE coincides with LOGSPACE. This is in particular the case if one of the functions l_i is the successor function induced by a total ordering on U. In general, however, PURPLE fails to capture all of LOGSPACE, as we show in Sect. 5. Since we are interested mainly in pointer programs on locally ordered graphs, let us now discuss different possible choices of operation labels for working with locally ordered graphs.

Graphs of constant degree. Pointer algorithms on locally ordered graphs of some fixed out-degree d are most easily represented as an L-model with L being $\{succ_1, \ldots, succ_d\}$ and U being the set of nodes in G. We write $\mathcal{A}_d(G)$ for this L-model.

Graphs of unbounded degree. For graphs of unbounded degree, it is more suitable to use pointer programs with three labels $succ$, $next$ and $prec$. Each locally ordered graph G determines a model $\mathcal{A}(G)$ of these labels. The universe of $\mathcal{A}(G)$ is the set $U = \{\langle v, i\rangle \mid v \in V,\ 0 \le i \le deg(v)\}$, where V denotes the node set of G. A pair $\langle v, i\rangle \in U$ with $i > 0$ represents an outgoing edge from v. Such pairs are often called *darts*, especially in the case of undirected graphs, where each undirected edge is represented by two darts, one for each direction in which the edge can be traversed. We include objects of the form $\langle v, 0\rangle$ to model the nodes themselves; thus the universe consists of the disjoint union of the nodes and the darts. The operation labels are interpreted by

$$succ(v, i) = \begin{cases} \langle succ_i(v), 0\rangle & \text{if } i \ge 1, \\ \langle v, 0\rangle & \text{if } i = 0, \end{cases}$$
$$next(v, i) = \langle v, \min(i + 1, deg(v))\rangle,$$
$$prec(v, i) = \langle v, \max(i - 1, 0)\rangle.$$

Using $next$ and $prec$ one can iterate over the darts on a node and using $succ$ one can follow the edge identified by a dart.

The presentation of graphs by darts and operations on them is commonly used in the description of LOGSPACE-algorithms [1,13,10], but it is also prevalent in other contexts, see [7] and the discussion there.

We note that with the dart representation, the `forall`-loop iterates over all darts and not the nodes of the graph. Iteration over all nodes can nevertheless be implemented easily. Since a program can recognise if x is a dart of the form $\langle v, 0\rangle$ by testing whether $x = x.prec$ is true, one can implement a `forall`-loop that visits only darts of the form $\langle v, 0\rangle$ and this amounts to iteration over all nodes.

While we have now introduced two representations for graphs of bounded degree, it is not hard to see that it does not matter which representation we use, as each program with labels $succ_1, \ldots, succ_d$ can be translated into a program with labels $succ, prec, next$ that recognises the same graphs, and vice versa.

3.1 Examples

We give two examples to illustrate the use of PURPLE. The first simple example program decides the property that all nodes have even in-degree. First we define a program $E(x, y, b)$ with variables x, y and b, such that after evaluation of $E(x, y, b)$ the boolean variable b is true if and only if there is an edge from the node given by x to that given by y. Such a program may be defined as:

$$
\begin{array}{l}
b := \mathrm{false};\ x' := x \\
\texttt{while}\ \neg(x' = x'.next)\ \texttt{do}\ x' := x'.next; \\
\texttt{while}\ \neg(x' = x'.prec)\ \texttt{do}\ (b := b \vee (y = x'.succ);\ x' := x'.prec)
\end{array}
$$

Herein, x' should be chosen afresh for each occurrence of $E(x, y, b)$ within a larger program. The following program then computes if the in-degree of all nodes is even.

$$
\begin{array}{l}
even := \mathrm{true}; \\
\texttt{forall}\ x\ \texttt{do} \\
\quad c := \mathrm{true}; \\
\quad \texttt{forall}\ y\ \texttt{do}\ (\texttt{if}\ y = y.prec\ \texttt{then}\ (E(y, x, b);\ \texttt{if}\ b\ \texttt{then}\ c := \neg c)); \\
\quad even := even \wedge c
\end{array}
$$

In the inner `forall`-loop we have a test for $(y = y.prec)$, so that the body of this loop is executed once for each graph node, rather than for each dart. In this way, the inner `forall`-loop is used to iterate over all nodes that have an edge to x. In the body we then make the assignment $c := \neg c$ for all nodes y that have an edge to x.

For a second, more substantial, example we show that acyclicity of undirected graphs can be decided in PURPLE, which is a well-known LOGSPACE-complete problem [1]. We next describe the LOGSPACE-algorithm of Cook & McKenzie [1] and then show that it can be written directly in PURPLE. The fact that PURPLE can express a LOGSPACE-complete problem does not conflict with PURPLE being strictly weaker than LOGSPACE, since not all reductions can be expressed in PURPLE.

Let G be an undirected locally ordered graph with node set V and let U be the universe of $\mathcal{A}(G)$. Let σ be the permutation on U such that $\sigma(v, 0) = \langle v, 0\rangle$ holds and such that $\sigma(v, i) = \langle w, j\rangle$ implies both $succ(v, i) = w$ and $succ(w, j) = v$. Note that in an undirected graph each edge between v and w is given by two half-edges, one from v to w and one from w to v. Thus, if the dart $\langle v, i\rangle$ represents one half of

an edge in G, then $\sigma(v,i)$ represents the other half of the same edge. Let π be the permutation on U satisfying $\pi(v,0) = \langle v, 0\rangle$, $\pi(v,i) = \langle v, i+1\rangle$ for $1 \leq i < \mathit{deg}(v)$ and $\pi(v, \mathit{deg}(v)) = \langle v, \min(1, \mathit{deg}(v)\rangle$.

Consider now the composite permutation $\pi \circ \sigma$ on U. It implements a way of exploring the graph G in depth-first-search order. That is, the walks in depth-first-search order are the obtained as the sequences of darts $x_0, x_1, \ldots$ defined by $x_{i+1} = (\pi \circ \sigma)(x_i)$. Since these sequences are generated by the composite permutation $\pi \circ \sigma$, it is easy to see that they can be generated in logarithmic space.

Being able to construct walks in depth-first-search order, one can use the following characterisation of acyclicity of undirected graphs to decide this property in LOGSPACE. An undirected graph is acyclic, if it does not have self-loops and if, for any node v and any integer i, the walk that starts by taking the i-th edge from v and proceeds in depth-first-search order does not visit v again until it traverses the i-th edge from v in the opposite direction. This is formulated precisely in the following lemma, a proof of which can be found in [1].

Lemma 2. *The undirected graph G is acyclic if and only if the following property holds for all $x_0 \in \{\langle v, i\rangle \mid v \in V,\ 1 \leq i \leq \mathit{deg}(v)\}$: If the walk $x_0, x_1, \ldots$ is defined by $x_{i+1} = (\pi \circ \sigma)(x_i)$ and $k > 0$ is the least number such that x_0 and x_k are darts on the same node, then both $k > 1$ and $\sigma(x_{k-1}) = x_0$ hold.*

It now only remains to show that the property in this lemma can be decided in PURPLE. A program $P_{\pi\circ\sigma}(x)$ implementing the permutation $\pi \circ \sigma$ can be written easily, since the `forall`-loop allows one to iterate over all the neighbours of any given node. Moreover, it is easy to write a program $P_{=}(x, y, b)$ that sets the boolean variable b to true if x and y are darts on the same node and to false otherwise. With these programs, the property of the above lemma can be decided in PURPLE as follows:

$\mathit{acyclic} := \mathrm{true}$
`forall` x `do`
  `if` $\neg(x = x.\mathit{prec})$ `then`
    $\mathit{keq0} := \mathrm{true};\ \mathit{kleq1} := \mathrm{true};\ x_0 := x;\ \mathit{returned} := \mathrm{false};$
    `while` $\neg \mathit{returned}$ `do`
      $(\mathit{kleq1} := \mathit{keq0};\ \mathit{keq0} := \mathrm{false};\ x' := x;\ P_{\pi\circ\sigma}(x);\ P_{=}(x_0, x, \mathit{returned}));$
    $P_\sigma(x');\ \mathit{acyclic} := \mathit{acyclic} \wedge \neg \mathit{kleq1} \wedge (x' = x)$

3.2 Operational Semantics

PURPLE is defined with the intention that an input must be accepted or rejected regardless of the order in which the `forall`-loops are run through. In this section we give the operational semantics of PURPLE, thus making this intention precise.

The operational semantics of PURPLE with operation labels in L is parameterised by an L-model $\mathcal{A} = (U, \boldsymbol{l})$ and is formulated in terms of a small-step transition relation $\longrightarrow_{\mathcal{A}}$. To define this transition relation, we define a set of *extended programs* that have annotations for keeping track of which variables have already been visited in the

Assignment

$$\langle x := t^U, q, \rho\rangle \longrightarrow_{\mathcal{A}} \langle \texttt{skip}, q, \rho[x \mapsto [\![t^U]\!]_{q,\rho}]\rangle$$
$$\langle b := t^B, q, \rho\rangle \longrightarrow_{\mathcal{A}} \langle \texttt{skip}, q[b \mapsto [\![t^B]\!]_{q,\rho}], \rho\rangle$$

Composition

$$\langle \texttt{skip}; P,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle P,\ q,\ \rho\rangle \qquad \frac{\langle P,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle P',\ q',\ \rho'\rangle}{\langle P; Q,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle P'; Q,\ q',\ \rho'\rangle}$$

Conditional

$$\langle \texttt{if}\ t\ \texttt{then}\ P\ \texttt{else}\ Q,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle P,\ q,\ \rho\rangle \text{ if } [\![t]\!]_{e,\rho} = \text{true}$$
$$\langle \texttt{if}\ t\ \texttt{then}\ P\ \texttt{else}\ Q,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle Q,\ q,\ \rho\rangle \text{ if } [\![t]\!]_{e,\rho} = \text{false}$$

`while`-loop

$$\langle \texttt{while}\ t\ \texttt{do}\ P,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle \texttt{skip},\ q,\ \rho\rangle \text{ if } [\![t]\!]_{q,\rho} = \text{false}$$
$$\langle \texttt{while}\ t\ \texttt{do}\ P,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle P;\ \texttt{while}\ t\ \texttt{do}\ P,\ q,\ \rho\rangle \text{ if } [\![t]\!]_{q,\rho} = \text{true}$$

`for`-loop

$$\langle \texttt{for}\ x \in \emptyset\ \texttt{do}\ P,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle \texttt{skip},\ q,\ \rho\rangle$$
$$\langle \texttt{for}\ x \in W\ \texttt{do}\ P,\ q,\ \rho\rangle \longrightarrow_{\mathcal{A}} \langle P;\ \texttt{for}\ x \in W \setminus \{v\}\ \texttt{do}\ P,\ q,\ \rho[x \mapsto v]\rangle \text{ for any } v \in W$$

Fig. 1. Operational Semantics

computation of the `forall`-loops. The set of extended programs consists of PURPLE-programs in which the `forall`-loops are not restricted to an iteration over the whole universe U, but where ($\texttt{for}\ x \in W\ \texttt{do}\ P$) is allowed for any subset W of U. We identify ($\texttt{forall}\ x\ \texttt{do}\ P$) with ($\texttt{for}\ x \in U\ \texttt{do}\ P$).

The transition relation $\longrightarrow_{\mathcal{A}}$ is a binary relation on configurations. A *configuration* is a triple $\langle P, q, \rho\rangle$, where P is an extended program, $q\colon Vars_B \to 2$ is an assignment of boolean variables and $\rho\colon Vars \to U$ is an assignment of pointer variables. The inference rules defining $\longrightarrow_{\mathcal{A}}$ appear in Fig. 1. In this figure, we denote by $[\![t]\!]_{q,\rho}$ the evident interpretations of terms with respect to the variable assignments q and ρ. The operational semantics is standard for all but the `for`-loop. We note, in particular, that the rules for the `for`-loop make the transition system non-deterministic.

We say that a program P is *strongly terminating* if for all $\mathcal{A}$ the computation of P on $\mathcal{A}$ always terminates, i.e. for all q and ρ there is no infinite reduction sequence of $\longrightarrow_{\mathcal{A}}$ starting from $\langle P,\ q,\ \rho\rangle$ and in particular there are ρ' and q' such that $\langle P,\ q,\ \rho\rangle \longrightarrow^*_{\mathcal{A}} \langle \texttt{skip}, q',\ \rho'\rangle$ holds.

To define what it means for an L-model to be recognised by a program, we choose a distinguished boolean variable *result* that indicates the outcome of a computation.

Definition 3 (Recognition). *A set X of L-models is* recognised *by a program P, if P is strongly terminating and, for all L-models $\mathcal{A}$ and all ρ, ρ', q and q' satisfying $\langle P,\ q,\ \rho\rangle \longrightarrow^*_{\mathcal{A}} \langle \texttt{skip}, q',\ \rho'\rangle$, one has $q'(result) = \text{true}$ if and only if $\mathcal{A} \in X$.*

Our notion of recognition should not be confused with the usual definition of acceptance for existentially (nondeterministic) or universally branching Turing machines; in contrast to those concepts it is completely symmetrical in X vs. $\overline{X}$. If the input is in X then all runs must accept; if the input is not in X then all runs must reject. In particular, not even for strongly terminating P can we *in general* define "the language of P". A program whose result depends on the traversal order does not recognise any set at all.

3.3 Basic Properties

In contrast to formalisms that depend on a global ordering, PURPLE is closed under isomorphism. This is formulated by the following lemma, in which we write σP for the program obtained from P by replacing each occurrence of $(\texttt{for}\ x \in W\ \texttt{do}\ P)$ by $(\texttt{for}\ x \in \sigma W\ \texttt{do}\ P)$. Note that if P is a PURPLE-program proper, i.e., not an extended one, then $\sigma P = P$ holds. Its proof is a straightforward induction.

Lemma 4. *Let $\sigma\colon U \to V$ be an isomorphism of L-models. Then $\langle P, q, \rho\rangle \longrightarrow_U \langle P', q', \rho'\rangle$ implies $\langle \sigma P, q, \sigma \circ \rho\rangle \longrightarrow_V \langle \sigma P', q', \sigma \circ \rho'\rangle$.*

The straightforward proof of the following lemma is based on the fact that the number of global configurations is polynomial in the input size.

Lemma 5. *For any program P with labels in L, there exists a* `while`*-free program P' that recognises the same sets of finite L-models.*

4 Related Models of Computation

Based on the intuition that computation with logarithmic space amounts to computation with a constant number of pointers, a number of formalisms of pure pointer algorithms have been proposed as approximations of LOGSPACE.

4.1 Jumping Automata on Graphs

Cook & Rackoff [2] introduce *Jumping Automata on Graphs* (JAGs) in order to study space lower bounds for reachability problems on directed and undirected graphs. Jumping automata on graphs are a model of pure pointer algorithms on locally ordered graphs. Each JAG may be described as a `forall`-free PURPLE-program over the operation labels $succ_1$, $succ_2$, ... and vice versa. Therefore, a JAG may move on the graph only by traversing edges and by jumping one graph variable to the position of another variable. As a result, JAGs can only compute local properties of the input graph. If, for instance, all the graph variables are in some connected component of the input graph then they will remain in it throughout the whole computation.

Cook & Rackoff show that it is possible to prove upper bounds on the expressivity of JAGs [2]. They show that both on directed and on undirected graphs reachability cannot be solved by them. Together with the local character of JAG computations, this can be used to show that many natural LOGSPACE-properties of graphs cannot be computed by JAGs. For instance, JAGs cannot compute the parity function and they cannot decide whether or not the input graph is acyclic.

While PURPLE is more expressive than JAGs, we hope that nevertheless separation results along the lines of the existing ones for JAGs, e.g. [2,4], could be achievable for PURPLE by further elaborating the pumping techniques used to establish those.

One criticism of Jumping Automata on Graphs as a computation model is that JAGs are artificially weak on directed graphs. Since, with the operations $succ_1$, $succ_2$, ..., edges can only be traversed in the forward direction, there is no way for a JAG to reach a node that has only outgoing edges, for example. One solution to this problem is to work with graphs having a local ordering both on the outgoing and on the incoming edges of each node, so that edges can be traversed in both directions [5]. The `forall`-loop of PURPLE represents another possible solution, since we can use it to iterate over all the nodes that have an edge to a given node, as we have shown in Sect. 3 above.

4.2 Deterministic Transitive Closure Logic

In the context of descriptive complexity theory DTC-logic was introduced as a logical characterisation of LOGSPACE on ordered structures [9]. The formulae of this logic are built from the connectives of first-order logic and a connective DTC for deterministic transitive closure. The formula $\mathrm{DTC}[\varphi(\boldsymbol{x},\boldsymbol{y})](\boldsymbol{s},\boldsymbol{t})$ expresses that, for all variable assignments ν, the pair $([\![\boldsymbol{s}]\!]_\nu, [\![\boldsymbol{t}]\!]_\nu)$ is in the transitive closure of the relation $\{(\boldsymbol{u},\boldsymbol{v}) \mid \mathcal{A} \models_\nu \varphi[\boldsymbol{u},\boldsymbol{v}] \wedge \forall \boldsymbol{z}.\, \varphi[\boldsymbol{u},\boldsymbol{z}] \Rightarrow \boldsymbol{z} = \boldsymbol{v}\}$, see e.g. [9].

While on structures with a totally ordered universe DTC-logic captures all of LOGSPACE, it is strictly weaker on unordered structures. A typical example of a property that cannot be expressed without an ordering is whether or not the universe has an even number of elements. If graphs are represented without any ordering by an edge relation $E(-,-)$, then DTC logic on graphs is very weak indeed. Grädel & McColm [8] have shown that there exist families of graphs on which DTC without any ordering is no more expressive than first-order logic.

Unordered DTC logic is nevertheless interesting, since on locally ordered graphs it captures an interesting class of pure pointer algorithms. Locally ordered graphs may be used in the logic by allowing, in addition to the binary edge relation $E(-,-)$, a ternary relation $F(-,-,-)$, such that $F(v,-,-)$ is a total ordering on $\{w \mid E(v,w)\}$ [5], for any v. With such a graph representation, DTC can encode JAGs and it is strictly more expressive, since it allows first-order quantification [5]. With suitable restrictions on the formulae, it is furthermore possible to characterise smaller classes of pointer algorithms on locally ordered graphs, such as the class given by Tree Walking Automata [11].

We next observe that PURPLE subsumes DTC logic on locally ordered graphs.

Proposition 6. *For each closed* DTC *formula φ for locally ordered graphs there exists a program P_φ such that, for any finite locally ordered graph G, $G \models \varphi$ holds if and only if P_φ recognises $\mathcal{A}(G)$.*

First-order quantifiers can be evaluated directly using the `forall`-loop. To see that $\mathrm{DTC}[\varphi(\boldsymbol{x},\boldsymbol{y})](\boldsymbol{a},\boldsymbol{b})$ can be evaluated, note that using the `forall`-loop we can iterate over all tuples $\boldsymbol{y}$, so that we can compute the unique $\boldsymbol{y}$ such that $\varphi(\boldsymbol{x},\boldsymbol{y})$ holds, if such a unique $\boldsymbol{y}$ exists, and we can recognise when such a unique $\boldsymbol{y}$ does not exist.

The converse of this proposition is not true, of course, since there is no DTC-formula that expresses that the input graph has an even number of nodes [5].

Although DTC-logic on locally ordered graphs is interesting, there are still many open questions regarding its expressive power. As far as we know, it is not known if DTC-logic on locally ordered graphs can express directed or undirected reachability. The best result we know is that of Etessami & Immerman [5], who show that undirected reachability cannot be expressed by a formula of the form $\mathrm{DTC}[\varphi(\boldsymbol{x},\boldsymbol{y})](\boldsymbol{s},\boldsymbol{t})$, where φ is a first-order formula (without a total ordering not every formula can be expressed in this way).

One reason for the lack of results on the expressive power of DTC on locally ordered graphs may be that at present there are no simple Ehrenfeucht-Fraïssé games for it; and such games are the main tool for proving inexpressivity results in finite model theory. Most of the existing results have been proved either directly or by reduction to a proof that uses automata-theoretic techniques. Etessami & Immerman, for example, obtain their inexpressivity result for undirected reachability by reduction to the corresponding result of Cook & Rackoff for JAGs. The relative success of automata-theoretic methods is part of the motivation for studying the programming language PURPLE.

Furthermore, when viewed as a model of pointer algorithms, DTC logic is somewhat unnaturally restricted. To implement universal quantification, say on a LOGSPACE Turing Machine, one needs to have a form of iteration over all possible pointers. If it is possible to iterate over all pointers, then it should also be possible to write a program for the parity function, even without any knowledge about a total ordering of the pointers. But this cannot be done in DTC. If we view the universal quantifier as a form of iteration that works without a total ordering, then it is more restricted than it needs to be.

The problem that a logic cannot express counting properties such as parity is often addressed in the literature by extending the logic with a totally ordered universe of numbers (of the same size as the first universe) and perhaps also counting quantifiers, see e.g. [5,9]. Such an addition appears to be quite a jump in expressivity. For instance, in view of Reingold's algorithm for undirected reachability, it is likely that undirected reachability becomes expressible in such a logic [Ganzow & Grädel, personal communication]. However, we believe that this problem is not expressible in PURPLE and in view of the Prop. 6 also not in DTC.

Another option of increasing the expressive power of DTC to include functions such as parity is to consider order-independent queries [9]. An order-independent query is a DTC formula that has access to a total ordering on the universe, but whose value does not depend on which particular ordering is chosen. Superficially, there appears to be a similarity to the `forall`-loop in PURPLE. However, order-independent queries are strictly stronger than PURPLE. They correspond to the version of PURPLE, in which each program is guaranteed that all `forall`-loops iterate over the universe in the same order, even though this order may be different from run to run. Of course, in either system (PURPLE with fixed traversal order and order-independent queries) one can use the order to capture all of LOGSPACE.

5 Counting

Our goal in this section is to show that the behaviour of an arbitrary program on the discrete graph with n nodes can be described abstractly and independently of n. From

this it will follow that PURPLE-programs are unable to detect whether n has certain arithmetic properties such as being a power of two.

Write G_n for the discrete graph with n nodes and write $V_n = \{1, \ldots, n\}$ for its set of nodes. Since this graph has constant degree 0, the L-model with universe V_n induced by it does not have any operations.

Fix a finite set M of graph variables. We show that no program with graph variables in M can recognise the set of all graphs G_n where n is a power of two. Since M is arbitrary, this will be enough to show the result for all PURPLE-programs.

The proof idea is to show that whether or not a (while-free) program P accepts G_n for sufficiently large n depends only on the initial value of boolean variables, the initial incidence relation of the pointer variables and the remainder modulo l of the graph size n for some l. In order to prove this by induction on programs we associate to each program P an abstraction $[\![P]\!]$, which given the initial valuation of the boolean variables q_0, the initial incidence relation E_0 and the size n modulo l yields a triple $(q, E, f) = [\![P]\!](q_0, E_0, n \bmod l)$ that characterises the final configuration of a computation as follows: q is the final valuation of the boolean variables, E is the final incidence relation of pointer variables, and $f\colon M \to M + \{\mathit{fresh}\}$ is a function that tells for each variable x whether it moves to a "fresh" node, i.e. one that was not occupied at the start of the computation, or assumes the position that some other variable $f(x) \in M$ had in the initial configuration. The exact position of the "fresh" variables will of course depend on the order in which `forall`-loops are being worked off. In fact, we will show that with an appropriate choice of ordering any position of the "fresh" variables can be realised, so long as it respects E.

For example, the abstraction of the program $(z := x;\ \texttt{forall}\ x\ \texttt{do}\ y := x)$ would map (q_0, E_0, l) to (q_0, E, f), where E is the equivalence relation generated by (x, y), and f is given by $f(x) = f(y) = \mathit{fresh}$ and $f(z) = x$. This means that for any n large enough, the program has a run on G_n that ends in a state where z assumes the position of x in the start configuration and where x and y lie on a node not occupied in the start configuration. Moreover, E specifies that x and y must lie on the same node.

Notice that the abstraction characterises the result of *some* run of the program. In the example, there also exists a run in which the last node offered by the `forall`-loop happens to be the (old) value of x, in which case x, y, z are all equal. The purpose of the abstraction is to show that certain sets cannot be recognised. Since for a set to be recognised the result must be the same (and correct) for all runs it suffices to exhibit (using the abstraction) a single run that yields a wrong result. This existential nature of the abstraction is made more precise in Def. 10 and Lemma 11.

Definition 7. *Let Σ denote the set of equivalence relations on M. For each environment ρ we write $[\rho] \in \Sigma$ for the equivalence relation given by $x[\rho]y \iff \rho(x) = \rho(y)$. Since the meaning of a boolean term t depends only on the induced equivalence relation, we define $[\![t^B]\!]_{q,E}$ as $[\![t^B]\!]_{q,\rho}$ for any ρ with $[\rho] = E \in \Sigma$.*

Definition 8. *The set F of* moves *is given by $F := M \to M + \{\mathit{fresh}\}$.*

The intention of a *move* $f \in F$ is that if $f(x) = y \neq \mathit{fresh}$ holds then variable x is set to the (old value of) variable y and if $f(x) = \mathit{fresh}$ holds then x is moved to a fresh location. This is formalised by the next definition.

Definition 9. *Let $\rho, \rho' \colon Vars \to V_n$ for some n and let $E' \in \Sigma$ and $f \in F$. We say that ρ' is* compatible with (E', ρ, f) *if $[\rho'] = E'$ holds and for all $x \in M$ we have*

- $f(x) = y \neq \mathit{fresh}$ *implies* $\rho'(x) = \rho(y)$*; and*
- $f(x) = \mathit{fresh}$ *implies* $\rho'(x) \notin \mathrm{im}(\rho)$.

In the rest of this section, we write Q for the set $Vars_B \to 2$ of boolean states.

Definition 10. *Let P be a program, $k, l > 0$ and*

$$B : Q \times \Sigma \times \mathbb{Z}/l\mathbb{Z} \longrightarrow Q \times \Sigma \times F$$

be a function. Say that (B, k, l) represents the behaviour of P on discrete graphs *if for all $n > k$, $q \in Q$ and $\rho \colon Vars \to V_n$ there exists $\rho' \colon Vars \to V_n$ with*

$$\langle P, q, \rho\rangle \longrightarrow^*_{G_n} \langle \mathtt{skip}, q', \rho'\rangle,$$

such that ρ' is compatible with (E', ρ, f) whenever $B(q, [\rho], n \bmod l) = (q', E', f)$.

Notice that in contrast to the definition of recognition we only require that *for some* evaluation of $\langle P, q, \rho\rangle$ the predicted behaviour is matched. This is appropriate because the intended use of this concept is negative: in order to show that no program can recognise a certain class of discrete graphs we should exhibit for each program a run that defies the purported behaviour. Of course, this also helps in the subsequent proofs since it is then us who can control the order of iteration through `forall`-loops.

Lemma 11. *Suppose (B, k, l) represents the behaviour of P on discrete graphs. Then whenever $n \geq k$ and $q \in Q$ and $\rho \colon Vars \to V_n$ and $B(q, [\rho], n \bmod l) = (q', E', f)$ then $\langle P, q, \rho\rangle \longrightarrow^*_{G_n} \langle \mathtt{skip}, q', \rho_1\rangle$ holds for all ρ_1 compatible with (E', ρ, f).*

Proof. Choose ρ' compatible with (E', ρ, f) that satisfies $\langle P, q, \rho\rangle \longrightarrow^*_{G_n} \langle \mathtt{skip}, q', \rho'\rangle$. Such ρ' must exist by the definition of "represents." We have $\rho'(x) = \rho(f(x)) = \rho_1(x)$ whenever $f(x) = y \neq \mathit{fresh}$ holds and $\rho'(x), \rho_1(x) \notin \mathrm{im}(\rho)$ whenever $f(x) = \mathit{fresh}$ holds. Hence we can find an automorphism $\sigma : G_n \to G_n$ satisfying $\sigma \circ \rho = \rho$ and $\sigma \circ \rho' = \rho_1$. The claim then follows from Lemma 4. □

Theorem 12. *There exist numbers k, l (depending on the number of variables in M) such that each* `while`*-free program P with graph variables in M is represented by $(\llbracket P\rrbracket, k, l)$ for some function $\llbracket P\rrbracket$.*

Proof. Put $N = |Q| \cdot |\Sigma| \cdot 2^{|M|\cdot|M|}$ and $k = 3|M| + N$ and $l = N!$.

We prove the claim by induction on P. For basic programs the statement is obvious.

Case $P = P_1; P_2$**.** Suppose we are given (q, E, t) where $t \in \mathbb{Z}/l\mathbb{Z}$. Write $\llbracket P_1\rrbracket(q, E, t) = (q_1, E_1, f_1)$ as well as $\llbracket P_2\rrbracket(q_1, E_1, t) = (q_2, E_2, f_2)$. Define $f \in F$ by

$$f(x) = f_1(f_2(x)), \text{ if } f_2(x) \in M;$$
$$f(x) = \mathit{fresh}, \text{ if } f_2(x) = \mathit{fresh}$$

Put $\llbracket P\rrbracket(q, E, t) = (q_2, E_2, f)$. Fix some n with $n \bmod l = t$ and $\rho \colon Vars \to V_n$ with $[\rho] = E$ and, using the induction hypothesis, choose ρ_1 compatible with (E_1, ρ, f_1)

and ρ_2 compatible with (E_2, ρ_1, f_2). Invoking Lemma 11 we may assume without loss of generality that $f_2(x) = \mathit{fresh}$ implies $\rho_2(x) \notin \mathrm{im}(\rho)$. We now claim that ρ_2 is compatible with (E_2, ρ, f), which establishes the current case. To see this claim pick $x \in M$ and suppose that $f_2(x) = y$ and $f_1(y) = z \in M$. Then $f(x) = z$ and $\rho_2(x) = \rho_1(y) = \rho(z)$ as required. If $f_2(x) = \mathit{fresh}$ then $f(x) = \mathit{fresh}$ and $\rho_2(x) \notin \mathrm{im}(\rho)$ by assumption on ρ_2. If, finally, $f_2(x) = y \in M$ and $f_1(y) = \mathit{fresh}$ then $\rho_2(x) = \rho_1(y) \notin \mathrm{im}(\rho)$ by compatibility of ρ_1.

Case `if` s^B `then` P_1 `else` P_2**.** Define $[\![P]\!](q, E, t) = [\![P_1]\!](q, E, t)$ when $[\![s]\!]_{q,E} = \mathrm{true}$ and $[\![P]\!](q, E, t) = [\![P_2]\!](q, E, t)$ when $[\![s]\!]_{q,E} = \mathrm{false}$.

Case `forall` x `do` P_1**.** We note that $k \geq 3|M|$ holds. Now, given (q, E, t) choose n minimal with $n \geq k$ and $n \bmod l = t$ and some $\rho\colon \mathit{Vars} \to V_n$ with $[\rho] = E$ and assume w.l.o.g. that $\mathrm{im}(\rho) \subseteq \{1, \ldots, |M|\}$. We then iterate through the graph nodes $\{1, \ldots, n\}$ in ascending order. Fresh nodes are chosen from $\{|M|+1, \ldots, 3|M|\}$. Since there are only $|M|$ variables we have enough space in this interval as to satisfy any request for fresh nodes possibly arising during the evaluation of P_1. Formally, we choose sequences ρ_i and q_i in such a way that

1. $\rho_0 = \rho$, $q_0 = q$;
2. $\langle P_1, q_i, \rho_i[x \mapsto i{+}1]\rangle \longrightarrow^*_{G_n} \langle \mathtt{skip}, q_{i+1}, \rho_{i+1}\rangle$;
3. for all $y \in M$, $\rho_{i+1}(y) \notin \mathrm{im}(\rho_i[x \mapsto i{+}1])$ implies $\rho_{i+1} \in \{|M|+1, \ldots, 3|M|\}$.

That such sequences exist follows from the induction hypothesis and Lemma 11.

For $I \in \Sigma$ define $I^+ = I \setminus x \cup \{(x, x)\}$, where $I \setminus x$ is I with all pairs involving x removed.

Putting $E_i = [\rho_i]$ we then get $[\rho_i[x \mapsto i{+}1]] = E_i^+$ for all $i > 3|M|$ and thus, again for $i > 3|M|$:

$$(q_{i+1}, E_{i+1}, f_{i+1}) = [\![P_1]\!](q_i, E_i^+, t)$$

for some sequence f_i.

Thus, for $i > 3|M|$ the incidence relations E_{i+1} no longer depend on ρ_i itself but only on the previous incidence relation E_i (and the valuation of the boolean variables).

Choose now f such that ρ_n is compatible with (E_n, ρ, f) and define $[\![P]\!](q, E, t) = (q_n, E_n, f)$.

Now we have to show that indeed $([\![P]\!], k, l)$ represents the behaviour of P on discrete graphs. To this end fix $m > n \geq k$ with $m \bmod l = t = n \bmod l$ and $\chi \in \mathit{Vars} \to V_m$ with $[\chi] = E$.

In view of Lemma 4 we may assume $\chi(x) = \rho(x)$ for all $x \in M$ so that in particular $\mathrm{im}(\chi) \subseteq \{1, \ldots, |M|\}$. We can now iterate through G_m in ascending order in exactly the same fashion yielding sequences I_i, χ_i, r_i such that

1. $\chi_0 = \chi$, $r_0 = q$;
2. $\langle P_1, r_i, \chi_i[x \mapsto i{+}1]\rangle \longrightarrow^*_{G_m} \langle \mathtt{skip}, r_{i+1}, \chi_{i+1}\rangle$;
3. $\chi_i(y) = \rho_i(y)$ and $q_i = r_i$ for all $y \in M$ and $i \leq n$.
4. for all $y \in M$, $\chi_{i+1}(y) \notin \mathrm{im}(\chi_i[x \mapsto i{+}1])$ implies $\chi_{i+1} \in \{|M|+1, \ldots, 3|M|\}$.

We put $I_i = [\chi_i]$ and find $(r_{i+1}, I_{i+1}, g_{i+1}) = [\![P_1]\!](r_i, I_i^+, t)$ for some function sequence g_i. Now consider the restriction of χ_i to $\{1, \ldots, |M|\}$, i.e., formally define

$\xi_i = \{(x, \chi_i(x)) \mid x \in M,\ \chi_i(x) \in \{1, \ldots, |M|\}\}$. We note that ξ_{i+1} does not depend upon χ_i but only on the incidence relation I_i (and q_i and t, of course). Indeed,

$$\xi_{i+1} = \{(y, v) \mid g_{i+1}(y) = z, (z, v) \in \xi_i\}.$$

In view of the choice of N there must exist indices $3|M| < t < t' \leq 3|M| + N = k$ such that $r_t = r_{t'}$ and $I_t = I_{t'}$ and $\xi_t = \xi_{t'}$. But then we also have $r_{t+d} = r_{t'+d}$ and $I_{t+d} = I_{t'+d}$ and $\xi_{t+d} = \xi_{t'+d}$ for all $d \geq 0$ with $t' + d \leq m$. Now, since $t' - t$ divides l we find that $r_i = r_{i'}$ and $I_i = I_{i'}$ and $\xi_i = \xi_{i'}$ as soon as $t' \leq k \leq n \leq i < i' \leq m$ and $i \cong i'$ modulo l. Hence in particular, $r_m = r_n = q_n$ and $I_m = I_n = E_n$ and $\xi_m = \xi_n$. Thus,

$$\langle P, \chi, q\rangle \longrightarrow^*_{G_m} \langle \mathtt{skip}, q_n, \chi_m\rangle$$

with χ_m compatible with (E_n, ρ, f_n) as required. □

Corollary 13. *Checking whether the input is a discrete graph with n nodes with n a power of two is possible in deterministic logarithmic space but not in* PURPLE.

Proof. To program this in LOGSPACE count the number of nodes on a work tape in binary and see whether its final content has the form 10^*. Suppose there was a PURPLE-program P recognising this class of graphs in the sense of Def. 3. By Lemma 5 we can assume that P is `while`-free. Then Theorem 12 furnishes $(\llbracket P\rrbracket, k, l)$ representing P on discrete graphs. Let *result* be the boolean variable in P containing the return value. Let n be a power of two such that $n > k$ holds and $n + l$ is not a power of two. Let ρ, q be arbitrary initial values. Now, since P purportedly recognises G_n we must have $\llbracket P\rrbracket(q, [\rho], n \bmod l) = (q', \_, \_)$ with $q'(\mathit{result}) = \mathrm{true}$. Now let χ be a valuation of the variables in G_{n+l} satisfying $[\rho] = [\chi]$. We then get $\langle P, \chi, q\rangle \longrightarrow^*_{G_{n+l}} \langle \mathtt{skip}, q', \_\rangle$, which contradicts the assumption on P, since on all runs of P on G_{n+l} the value of the boolean variable *result* would have to be false. Recall the explanation after Def. 3. □

A reader of an earlier version of this paper suggested an alternative, purportedly simpler route to this result. From Theorem 12 one can conclude that if X is a property of discrete graphs then the set $\{a^n \mid G_n \in X\}$ is a regular set over the unary alphabet $\{a\}$; in fact, every regular set over this alphabet arises in this way. Given that all iterations through n indistinguishable discrete nodes look essentially the same and that the internal control of a PURPLE-program is a finite state machine it should not be able to do anything more than a DFA when run on a unary word.

We cannot see, how to turn this admittedly convincing intuition into a rigorous proof and would like to point out that a PURPLE-program can test for equality of nodes, thus it can store certain nodes from an earlier iteration and then find out when they appear in a subsequent one. Indeed, if the order of traversal were always the same then we could use this feature to define a total ordering on the nodes and thus program all of LOGSPACE including the question whether the cardinality of the universe is a power of two. This would be true even if the traversal order was not fixed a priori but the same for all iterations in a given run of a program. Thus, any proof of Theorem 12 must exploit the fact that an input is recognised or rejected only if this is the case for all possible traversal sequences. Doing so rigorously is what takes up most of the work in our proof.

6 Conclusion

We believe that PURPLE captures a natural class of pure pointer programs within LOGSPACE. By showing that PURPLE is unable to express arithmetic properties, we have demonstrated that it is not merely a reformulation of LOGSPACE but defines a standalone class whose properties are worth of study in view of its motivation from practical programming with pointers.

On the one hand, PURPLE strictly subsumes JAGs and DTC logic and can therefore express many pure pointer algorithms in LOGSPACE. On the other hand, Reingold's algorithm for st-reachability in undirected graphs uses counting registers of logarithmic size. We believe that it is not possible to solve undirected reachability in PURPLE and therefore not in DTC-logic. The details will appear elsewhere.

We also consider it an important contribution of our work to have formalised the notion that the order of iteration through a data structure may not be relied upon. Such provisos often appear in the documentation of library functions like Java's iterators. Our notion of recognition in Def. 3 captures this and, as argued at the end of Sec. 5, it measurably affects the computing strength (otherwise all of LOGSPACE could be computed).

The appearance of "freshness" and the accompanying $\forall\exists$-coincidence expressed in Lemma 11 suggest some rather unexpected connection to the semantic study of name generation and α-conversion [6,12]. It remains to be seen whether this is merely coincidence or whether techniques and results can be fruitfully transferred in either direction.

References

1. Cook, S.A., McKenzie, P.: Problems complete for deterministic logarithmic space. Journal of Algorithms 8(3), 385–394 (1987)
2. Cook, S.A., Rackoff, C.: Space lower bounds for maze threadability on restricted machines. SIAM Journal of Computing 9(3), 636–652 (1980)
3. Ebbinghaus, H.D., Flum, J.: Finite Model Theory. Springer, Heidelberg (1995)
4. Edmonds, J., Poon, C.K., Achlioptas, D.: Tight lower bounds for st-connectivity on the NNJAG model. SIAM Journal of Computing 28(6), 2257–2284 (1999)
5. Etessami, K., Immerman, N.: Reachability and the power of local ordering. Theoretical Computer Science 148(2), 261–279 (1995)
6. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing 13, 341–363 (2002)
7. Gonthier, G.: A computer-checked proof of the four-colour theorem, `http://research.microsoft.com/~gonthier`
8. Grädel, E., McColm, G.L.: On the power of deterministic transitive closures. Information and Computation 119(1), 129–135 (1995)
9. Immerman, N.: Descriptive Complexity. Springer, Heidelberg (1999)
10. Johannsen, J.: Satisfiability problems complete for deterministic logarithmic space. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 317–325. Springer, Heidelberg (2004)
11. Neven, F., Schwentick, T.: On the power of tree-walking automata. In: Welzl, E., Montanari, U., Rolim, J. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 547–560. Springer, Heidelberg (2000)
12. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or: What's new? In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 122–141. Springer, Heidelberg (1993)
13. Reingold, O.: Undirected st-connectivity in log-space. In: STOC, pp. 376–385 (2005)