Pure Pointer Programs with Iteration

Martin Hofmann and Ulrich Schöpp
Ludwig-Maximilians-Universität München
Oettingenstraße 67, D-80538 Munich, Germany

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 using 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.

Categories and Subject Descriptors: F.1.3 [**Complexity Measures and Classes**]: Machine-independent complexity

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Pointer program, iteration in unspecified order, logarithmic space, pebble automaton, deterministic transitive closure logic

1. INTRODUCTION

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

While such 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 are *Jumping Automata on Graphs* (JAGs) [Cook and Rackoff 1980] and *Deterministic Transitive Closure (*DTC*) logic* [Ebbinghaus and Flum 1995]. 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 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 s-t-connectivity in undirected graphs [Reingold 2005], 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.

This observation suggests that what one can compute with programs that use only a constant number of pointers to graph nodes is strictly less than LOGSPACE.

Indeed, Cook and Rackoff [1980] show that s-t-connectivity is not computable with JAGs. On the other hand, JAGs cannot compute some other problems either

that clearly can be implemented using by storing just a constant number of graph nodes, such as whether a graph has isolated nodes. Thus, important though the result of Cook and Rackoff is, we cannot consider it evidence that computation with "a constant number of pointers to graph nodes" is strictly weaker than LOGSPACE.

Likewise, DTC-logic without a total order on the graph nodes is fairly weak and brittle [Grädel and McColm 1995; Etessami and Immerman 1995], 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 and DTC-logic becomes as strong as LOGSPACE. But with a total order one has more than just a constant number of pointers: the total order can be used to simulate logarithmically sized work tapes [Ebbinghaus and Flum 1995].

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 "a 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 that 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.

In general, JAGs cannot visit all nodes of their input graphs and therefore are incapable of evaluating DTC formulae 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: graph pointer variables and boolean variables. Pointer variables are used to reference graph nodes in the same way that pebbles are used in the automata-theoretic formulation of JAGs. Boolean variables are used to represent the finite state of the JAG and the usual boolean operations are available for boolean expressions. For pointer variables, on the other hand, one only has an equality test and, for each constant number i, a successor operation to move a pointer variable along the i-th edge from the graph node it points to.

Our proposal, which we call PURPLE for "PURe Pointer LanguagE", consists of adding to this programming language theoretic version of JAGs a `forall`-loop (`forall` x `do` P) whose meaning is to set the pointer 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: ($b := \text{true}$; `forall` x `do` $b := \neg b$). It is known that this property is not expressible in locally-ordered DTC logic [Etessami and Immerman 1995], which establishes strictness of the inclusion.

Besides the introduction of PURPLE, the main technical contribution of this paper is a proof that PURPLE is strictly weaker than LOGSPACE. This proof goes by showing that the property "the number of graph nodes is a power of two" is not computable in PURPLE. It therefore also shows that the `forall`-loop in PURPLE cannot be used to somehow simulate counting, as can be done in totally ordered DTC-logic [Ebbinghaus and Flum 1995]. This further validates the choice of the `forall`-loop as a primitive for expressing pure pointer algorithms that avoids the artificial strength of a total ordering.

In order to justify the naturality of PURPLE we can invoke, besides the fact that formulae of locally-ordered DTC-logic may be evaluated easily, 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. Note though, that we do not provide a decidable criterion as to when a PURPLE-program is legal, i.e. when the output is indeed independent of the traversal order.

2. POINTER STRUCTURES

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

Definition 2.1. 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 together with n unary functions $l_1, \ldots, l_n \colon U \to U$.

We will not distinguish notationally between L-models and their underlying sets, writing U for both of them.

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 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

The pure pointer language PURPLE is parameterised by a finite set of *operation labels* $L = \{l_1, \ldots, l_n\}$.

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.

3.1 Terms

There are two types of terms, one for boolean values and one for pointers into the universe U. Fix countably infinite sets $Vars_U$ 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.

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

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 of this model to x. The only direct observation about pointers is the equality test $t = t'$.

3.2 Programs

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

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

We abbreviate ($\texttt{if } b \texttt{ then } P \texttt{ else skip}$) 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. By leaving the iteration order unspecified we allow for the case that, within a single run of the program, two executions of the same `forall`-loop may use a different ordering.

We write $Vars_U(P)$ for the pointer variables in program P and $Vars_B(P)$ for the boolean variables in it.

On certain classes of L-models the power of PURPLE coincides with that of 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. Namely, in this case we can evaluate queries in deterministic transitive closure logic with a total ordering (see Section 4.2), which is known to capture LOGSPACE [Ebbinghaus and Flum 1995].

In general, however, PURPLE fails to capture all of LOGSPACE, as we show in Section 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.

3.3 Graph Representations

Locally ordered graphs of some fixed out-degree d are most easily represented as L-models with L being $\{succ_1, \ldots, succ_d\}$. Any locally ordered graph G of out-degree d becomes such an L-model if we take the underlying set to be the node set of G and interpret $succ_i$ such that $x.succ_i$ is the graph node reached from node x by following edge number i. We write $U_d(G)$ for this L-model.

However, for graphs of unbounded degree, this representation does not work well. Each particular program can only address graph edges of some maximum number, so if the degree is not bounded by some constant then even simple properties cannot be programmed, such as whether or not all edges are self-loops.

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 $U(G)$ of these labels. The universe of this model is $\{(v,i) \mid v \in V,\ 0 \le i \le deg(v)\}$, where V denotes the node set of G. A pair $(v,i) \in U(G)$ 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 distinct darts, one for each direction in which the edge can be traversed. We include objects of the form $(v,0)$ 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

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

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 [Cook and McKenzie 1987; Reingold 2005; Johannsen 2004]. It is also prevalent in other contexts, see [Gonthier 2005] and the discussion there.

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

Moreover, it is possible to test whether two variables x and y denote darts on the same node, i.e. if they are of the form (v,i) and (v,j) respectively. Specifically, a program $P_=(x, y, equals)$ for this equality test is given by:

$$x' := x;\ y' := y;$$
$$\texttt{while } \neg(x' = x'.prec) \texttt{ do } x' := x'.prec;$$
$$\texttt{while } \neg(y' = y'.prec) \texttt{ do } y' := y'.prec;$$
$$equals := (x' = y')$$

Herein, x' and y' should be chosen afresh for each occurrence of $P_=(x, y, equals)$ within a larger program.

While we have now introduced two representations for graphs of constant 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, see Proposition 3.5.

We remark that the representation of graphs using darts is closely related to the familiar one using adjacency lists if we consider these lists implemented in the usual way using pointers; we learned this from [Gonthier 2005].

3.4 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 if x and y represent graph nodes, i.e. satisfy $x = x.prec$ and $y = y.prec$, then after the evaluation of $E(x, y, b)$ the boolean variable b is true if and only if there is an edge from x to y.

$$b := \text{false};\ x' := x;$$
$$\texttt{while}\ \neg(x' = x'.next)\ \texttt{do}\ (x' := x'.next;\ b := b \vee (x'.succ = y))$$

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

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

In the `forall`-loops we have tests for $(x = x.prec)$ and $(y = y.prec)$, so that the loop bodies are executed once for each graph node rather than for each dart. In this way, the inner `forall`-loop iterates over all nodes having 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, which is a well-known LOGSPACE-complete problem [Cook and McKenzie 1987] can be decided in PURPLE. We first recall the LOGSPACE-algorithm given in [Cook and McKenzie 1987] 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 it being strictly weaker than LOGSPACE, since not all reductions to the complete problem can be expressed in PURPLE.

Let G be an undirected locally ordered graph with node set V and let $U(G)$ be the model defined in Section 3.3 above. Let σ be the permutation on $U(G)$ such that $\sigma(v, 0) = (v, 0)$ holds and such that $\sigma(v, i) = (w, j)$ implies both $succ_i(v) = w$ and $succ_j(w) = v$. 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 (v, i) represents one half of an edge in G, then $\sigma(v, i)$ is a distinct dart that represents the other half of the same edge. Let π be the permutation on $U(G)$ satisfying $\pi(v, 0) = (v, 0)$, $\pi(v, i) = (v, i + 1)$ for $1 \leq i < deg(v)$ and $\pi(v, deg(v)) = (v, \min(1, deg(v)))$.

Consider now the composite permutation $\pi \circ \sigma$ on $U(G)$. It implements a way of exploring the graph G in depth-first-search order. That is, the walks in depth-first-search order are 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 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 [Cook and McKenzie 1987].

LEMMA 3.1. *The undirected graph G is acyclic if and only if the following property holds for all $x_0 \in \{(v,i) \mid v \in V,\ 1 \leq i \leq deg(v)\}$: 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.*

We start by writing programs $P_\pi(x)$ and $P_\sigma(x)$ for the permutations π and σ. We write out $P_\pi(x)$ concretely and omit the definition of $P_\sigma(x)$, noting that it can be defined similarly to $E(x,y,b)$ above. The program $P_\pi(x)$ may be defined as:

```
if ¬(x = x.prec) then (
  if x = x.next then (while ¬(x = x.prec) do x := x.prec);
  x := x.next)
```

It should be clear that using $P_\pi(x)$ and $P_\sigma(x)$ it is possible to define a program $P_{\pi\circ\sigma}(x)$ for the composite permutation.

With these programs, the property of the above lemma can be decided in PURPLE using a single `forall`-loop with an inner `while`-loop:

```
acyclic := true
forall x do
  if ¬(x = x.prec) then (
    keq0 := true; kleq1 := true;
    x0 := x; returned := false;
    while ¬returned do
      (kleq1 := keq0; keq0 := false; x' := x; P_{π∘σ}(x); P_=(x0, x, returned));
    P_σ(x'); acyclic := acyclic ∧ ¬kleq1 ∧ (x' = x0))
```

3.5 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 a finite L-model U and is formulated in terms of a small-step transition relation $\longrightarrow_U$. To define this transition relation, we define a set of *annotated programs* that

Boolean Terms

$$[\![\text{true}]\!]_{q,\rho} = \text{true}$$
$$[\![\text{false}]\!]_{q,\rho} = \text{false}$$
$$[\![b]\!]_{q,\rho} = q(b)$$
$$[\![\neg t]\!]_{q,\rho} = \neg [\![t]\!]_{q,\rho}$$
$$[\![t_1 \wedge t_2]\!]_{q,\rho} = [\![t_1]\!]_{q,\rho} \wedge [\![t_2]\!]_{q,\rho}$$
$$[\![t_1 \vee t_2]\!]_{q,\rho} = [\![t_1]\!]_{q,\rho} \vee [\![t_2]\!]_{q,\rho}$$
$$[\![t_1 = t_2]\!]_{q,\rho} = ([\![t_1]\!]_{q,\rho} = [\![t_2]\!]_{q,\rho})$$

Graph Pointer Terms

$$[\![x]\!]_{q,\rho} = \rho(x)$$
$$[\![t.l_i]\!]_{q,\rho} = l_i\,([\![t]\!]_{q,\rho})$$

Fig. 1. Interpretation of Terms

have annotations for keeping track of which variables have already been visited in the computation of the `forall`-loops. The set of annotated programs consists of PURPLE-programs in which the `forall`-loops are not restricted to an iteration over the whole universe U, but where (`for` $x \in W$ `do` P) is allowed for any subset W of U. We identify (`forall` x `do` P) with (`for` $x \in U$ `do` P), which allows us to consider each unannotated PURPLE program an annotated program as well.

The transition relation $\longrightarrow_U$ is a binary relation on configurations. A *configuration* is a triple (P, q, ρ), where P is an annotated program, $q\colon Vars_B(P) \to 2$ is an assignment of boolean variables to booleans and $\rho\colon Vars_U(P) \to U$ is an assignment of pointer variables to elements of the universe. For assignments q and ρ that are defined on more than the variables in P, we will usually write just (P, q, ρ) for $(P, q|_{Vars_B(P)}, \rho|_{Vars_U(P)})$, making the restriction implicit.

The inference rules defining $\longrightarrow_U$ appear in Figure 2. There we write $[\![t]\!]_{q,\rho}$ for the evident interpretation of terms with respect to the variable assignments q and ρ, as defined in Figure 1. 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 configuration (P, q, ρ) on an L-model U is *strongly terminating* if there is no infinite reduction sequence of $\longrightarrow_U$ starting from it. We say that a program P is *strongly terminating* if any configuration of the form (P, q, ρ) on any L-model is strongly terminating.

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 3.2. A set X of finite L-models is *recognised* by a program P, if P is strongly terminating and, for all L-models U and all ρ, ρ', q and q' satisfying $(P, q, \rho) \longrightarrow_U^* (\texttt{skip}, q', \rho')$, one has $q'(\mathit{result}) = \text{true}$ if and only if $U \in X$.

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.

Assignment

$$(x := t^U,\, q,\, \rho) \longrightarrow_U (\texttt{skip},\, q,\, \rho[x \mapsto [\![t^U]\!]_{q,\rho}])$$
$$(b := t^B,\, q,\, \rho) \longrightarrow_U (\texttt{skip},\, q[b \mapsto [\![t^B]\!]_{q,\rho}],\, \rho)$$

Composition

$$(\texttt{skip}; P,\, q,\, \rho) \longrightarrow_U (P,\, q,\, \rho) \qquad \frac{(P,\, q,\, \rho) \longrightarrow_U (P',\, q',\, \rho')}{(P; Q,\, q,\, \rho) \longrightarrow_U (P'; Q,\, q',\, \rho')}$$

Conditional

$$(\texttt{if } t \texttt{ then } P \texttt{ else } Q,\, q,\, \rho) \longrightarrow_U (P,\, q,\, \rho) \text{ if } [\![t]\!]_{q,\rho} = \text{true}$$
$$(\texttt{if } t \texttt{ then } P \texttt{ else } Q,\, q,\, \rho) \longrightarrow_U (Q,\, q,\, \rho) \text{ if } [\![t]\!]_{q,\rho} = \text{false}$$

`while`-loop

$$(\texttt{while } t \texttt{ do } P,\, q,\, \rho) \longrightarrow_U (\texttt{skip},\, q,\, \rho) \text{ if } [\![t]\!]_{q,\rho} = \text{false}$$
$$(\texttt{while } t \texttt{ do } P,\, q,\, \rho) \longrightarrow_U (P;\ \texttt{while } t \texttt{ do } P,\, q,\, \rho) \text{ if } [\![t]\!]_{q,\rho} = \text{true}$$

`for`-loop

$$(\texttt{for } x \in \emptyset \texttt{ do } P,\, q,\, \rho) \longrightarrow_U (\texttt{skip},\, q,\, \rho)$$
$$(\texttt{for } x \in W \texttt{ do } P,\, q,\, \rho) \longrightarrow_U (P;\ \texttt{for } x \in W \setminus \{v\} \texttt{ do } P,\, q,\, \rho[x \mapsto v]) \text{ for any } v \in W$$

Fig. 2. Operational Semantics

3.6 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 annotated one, then σP is identical to P.

LEMMA 3.3. *Let $\sigma\colon U \to V$ be an isomorphism of L-models. Then, whenever $(P,\, q,\, \rho) \longrightarrow_U (P',\, q',\, \rho')$ holds, then so does $(\sigma P,\, q,\, \sigma \circ \rho) \longrightarrow_V (\sigma P',\, q',\, \sigma \circ \rho')$.*

PROOF. The statement is proved by a routine induction on the derivation of $(P,\, q,\, \rho) \longrightarrow_U (P',\, q',\, \rho')$. □

Another property of PURPLE that is useful for studying its expressivity is that `while`-loops can be eliminated.

LEMMA 3.4. *For any program P with labels in L, there exists a `while`-free program P' that recognises the same sets of finite L-models.*

PROOF. If P is not strongly terminating then it does not recognise any set of L-models, and we can take P' to be, e.g. $((\texttt{forall } x \texttt{ do skip}); \mathit{result} := (x = y))$.

To prove the assertion for strongly terminating programs P, we prove the following stronger property by induction on programs. For any program P there exists a `while`-free (and therefore strongly terminating) program P' such that whenever $(P,\, q,\, \rho)$ is a strongly terminating configuration on some L-model U then for all q' and ρ' the following equivalence holds:

$$(P,\, q,\, \rho) \longrightarrow_U^* (\texttt{skip},\, q',\, \rho') \quad \Longleftrightarrow \quad (P',\, q,\, \rho) \longrightarrow_U^* (\texttt{skip},\, q',\, \rho')$$

The assertion then follows immediately.

We prove the assertion by induction on the program P. The cases where P is `skip` or $x := t$ are trivial, as we can simply take P' to be P. The cases where P is $P_1; P_2$ or `if` t `then` P_1 `else` P_2 or `forall` x `do` P_1 follow easily from the induction hypothesis. Note that all the configurations reachable from a strongly terminating configuration are themselves strongly terminating.

We spell out the case where P is `while` t `do` P_1. In this case, let b be the number of boolean variables in P_1 and let p be the number of pointer variables in P_1. Let P_1' be the program obtained from P_1 using the induction hypothesis.

Write Q for the program (`if` t `then` P_1' `else skip`) and Q^{2^b} for the composition $Q; \ldots; Q$ of 2^b copies of Q. Let $z_1, \ldots, z_p$ be fresh variables and define

$$P' := \texttt{forall } z_1 \texttt{ do forall } z_2 \texttt{ do } \ldots \texttt{ forall } z_p \texttt{ do } Q^{2^b}.$$

This program behaves like the original `while`-loop with the additional constraint that the execution on any L-model U is aborted after $|U|^p \cdot 2^b$ executions of the loop body.

Let now (P, q, ρ) be a strongly terminating configuration on some L-model U. Note that in this case, the loop body P_1 can be executed at most $|U|^p \cdot 2^b$ times in any computation sequence starting from (P, q, ρ). This is the case, since there are at most $|U|^p \cdot 2^b$ configurations of the form (P, q', ρ'), so that if there were a run that executes the loop body P_1 more times, then a configuration of the form (P, q', ρ') would appear twice in this run. But then we could also construct an infinite run, in contradiction to the assumption that (P, q, ρ) is strongly terminating.

With the induction hypothesis, we thus get that P' adequately simulates P, i.e. that the desired assertion holds. □

As promised in Section 3, we next show that the two ways of representing locally ordered graphs of constant degree introduced there are essentially the same.

PROPOSITION 3.5. *For each natural number d and each set X of finite locally ordered graphs of degree d, the following are equivalent.*

(1) *The set $\{U_d(G) \mid G \in X\}$ is recognised by some* PURPLE*-program with labels in $\{succ_1, \ldots, succ_d\}$.*

(2) *The set $\{U(G) \mid G \in X\}$ is recognised by some* PURPLE*-program with labels in $\{succ, next, prec\}$.*

PROOF. To show the implication (1) $\implies$ (2), we first note that it suffices to show that for each strongly terminating program P with labels $succ_1, \ldots, succ_d$ there exists a strongly terminating program P' with labels $succ, next, prec$ that simulates P in the following sense: Whenever $(P, q, \rho) \longrightarrow^*_{U_d(G)} (\texttt{skip}, q', \rho')$ holds then so does $(P', q, \imath \circ \rho) \longrightarrow^*_{U(G)} (\texttt{skip}, q', \imath \circ \rho')$, where we write $\imath \colon U_d(G) \to U(G)$ for the injection defined by $\imath(v) = (v, 0)$.

To see that it suffices to prove this property, note first that we can restrict our attention to strongly terminating programs since only those can recognise any set of models. Let P be a strongly terminating program with labels $succ_1, \ldots, succ_d$ and let P' be obtained from P by the above property. Write $N(x)$ for the normalisation program (`while` $\neg(x = x.prec)$ `do` $x := x.prec$) that normalises the value of x from

(v, i) to $(v, 0)$. By pre-composing P' with such normalisation programs, we can assume that P' is started with a variable assignments of the form $\imath \circ \rho$. If $x_1, \ldots, x_n$ are the pointer variables in P' then the above property therefore implies that, for any set X of finite locally ordered graphs of degree d, the program $N(x_1); \ldots; N(x_n); P'$ accepts $\{U(G) \mid G \in X\}$ if and only if P accepts $\{U_d(G) \mid G \in X\}$.

To prove the above property, we note first that by Lemma 3.4 we can assume P to be `while`-free. Without loss of generality, we can moreover assume that P contains only terms of depth at most 1, i.e. terms such as x, $x.succ_i$ and $x = y$ for variables x and y, but not $x.succ_i.succ_j$ or $x.succ_i = y$, etc., and that the boolean term in each `if`-statement is a variable.

Then, the construction of P' from P goes by an easy induction on P. For example, if P is $x' := x.succ_2$ then we can take P' to be $x' := x.next.next.succ$ and if P is `forall` x `do` P_1 then we can take P' to be (`forall` x `do if` $x = x.prec$ `then` P_1'), where P_1' is the translation of P_1.

For the implication (2) $\implies$ (1) we note that for graphs of constant degree darts can be encoded in the language. To translate a program P' with operation labels *succ*, *next* and *prec* to a program P with labels $succ_1, \ldots, succ_d$, we pick $\lceil \log(d+1) \rceil$ fresh boolean variables for each pointer variable x in P'. Using these variables we can encode pairs (x, i_x) with $0 \leq i_x \leq d$ and it is straightforward to encode the operations *succ*, *next* and *prec* on the thus encoded darts. Using a similar setup as above, we can then translate all programs. For instance the loop (`forall` x `do` P_1') can be translated to (`forall` x `do` $(i_x := 0; P_1; \ldots; i_x := d; P_1)$), where P_1 is the translation of P_1' and where we have used suggestive notation for working with i_x, which is a number encoded by boolean variables.

Note that the simulated `forall`-loop iterates over the darts on each single node in a certain fixed order. This is enough to simulate the original program, however, as it must accept or reject its input for all possible ways of iterating over all darts. □

PROPOSITION 3.6. *Let X be a class of finite L-models recognised by a program P. Then X is decidable in* LOGSPACE.

PROOF. We assume a reasonable coding of L-models as words to be written on the input tape of a LOGSPACE-machine. Pointers should be encoded as bitstrings of logarithmic size and the L-operations should be computable in LOGSPACE. That is, for each $l \in L$ we assume a LOGSPACE-subroutine that, whenever the input tape contains a particular L-model U and a given worktape contains the encoding of some $x \in U$, then after running the subroutine a given worktape contains the encoding of $l(x)$. Moreover, we assume a LOGSPACE-subroutine capable of deciding whether the contents of a worktape form the encoding of an element of U.

For example, such an encoding of L-models can be obtained from a standard encoding of graphs by viewing an L-model U as the edge-labelled graph with node set U and edge set $\{x \xrightarrow{l} y \mid l \in L,\ x, y \in U,\ y = l(x)\}$.

It is then possible to define a LOGSPACE-program which accepts precisely the inputs from X. The program has a worktape for each pointer variable of P; it uses its finite control to represent the values of the boolean variables and the program counter of P. All PURPLE-constructs except possibly the `forall`-loop are easily represented within LOGSPACE. For the `forall`-loop we note that under the assumption that P recognises X it will yield the same result, accept or reject, ir-

respective of the order in which the `forall`-loops are worked off. Therefore, the simulating LOGSPACE-program is free to choose a particular ordering, for example the one induced by the encoding of the elements of U as bitstrings. One thus enumerates all bitstrings of the required length and checks for each one whether it is an encoding or not by using the assumed subroutine. The ones that are encodings are then presented to the simulation of the body of the loop. □

We emphasise that it is presumably not possible to decide in LOGSPACE whether or not $(P, q, \rho) \longrightarrow_U^* (\texttt{skip}, q', \rho')$ holds for an arbitrary P and L-model U (and q, ρ, q', ρ'). This is because the final ρ' and q' reached may in general depend on the traversal order and it will not be possible for a LOGSPACE-program to guess one such. Indeed, this latter contrived question is hard for nondeterministic LOGSPACE (reachability in directed graphs is easily encoded), but of no interest here.

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 and Rackoff [1980] 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 with operation labels $succ_1, succ_2, \ldots$ and vice versa. The pointer variables in these PURPLE programs are references to graph nodes. As a result, JAGs can only compute local properties of the input graph. If, for instance, all the graph pointer variables reference nodes in some connected component of the input graph then they will remain in this connected component throughout the whole computation.

Cook and Rackoff [1980] show that it is possible to prove upper bounds on the expressivity of JAGs. Indeed, they show that both on directed and on undirected graphs reachability cannot be solved by JAGs. Together with the local character of JAG computations, their results imply 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.

One criticism of Jumping Automata on Graphs as a computation model is that JAGs are artificially weak on directed graphs. Since, with the operation labels $succ_1, succ_2, \ldots$, edges can only be traversed in the forward direction, there is no way for a JAG to reach a node that only has 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 [Etessami and Immerman 1995]. In PURPLE we can use the `forall`-loop to iterate over all the nodes that have an edge to a given node, as we have shown in Section 3 above. We see this as further evidence for the robustness of PURPLE.

4.2 Deterministic Transitive Closure Logic

In the context of descriptive complexity theory deterministic transitive closure logic (DTC-logic) was introduced as a logical characterisation of LOGSPACE on ordered structures [Immerman 1999]. This logic is parameterised by a relational signature σ. Its syntax extends that of first-order logic with equality over the signature σ by a construct $(\mathrm{DTC}_{\vec{x},\vec{y}}\,\varphi)(\vec{s},\vec{t})$ for deterministic transitive closure. The deterministic transitive closure of a binary relation R is the transitive closure of the relation $R_d = \{(x,y) \mid xRy \wedge (\forall z.\, xRz \Rightarrow z = y)\}$ and the formula $(\mathrm{DTC}_{\vec{x},\vec{y}}\,\varphi)(\vec{s},\vec{t})$ expresses that the pair $(\vec{s},\vec{t})$ is in the deterministic transitive closure R_d of the binary relation on tuples that is defined by $\vec{x}R\vec{y} \iff \varphi(\vec{x},\vec{y})$.

Note that R_d is a partial function in the sense that $R_d(x,y) \wedge R_d(x,y')$ implies $y = y'$ and that if R itself is a partial function then $R = R_d$.

The formula $(\mathrm{DTC}_{\vec{x},\vec{y}}\,\varphi)(\vec{s},\vec{t})$ is well-formed when $\vec{x}$ and $\vec{y}$ are vectors of variables, $\vec{s}$ and $\vec{t}$ are vectors of terms, all vectors have the same length and no variable appears more than once in $\vec{x},\vec{y}$. Any occurrence in φ of a variable in $\vec{x},\vec{y}$ becomes bound in the DTC-formula.

We use the following notation for the semantics of DTC-logic. We write $\mathcal{A} \models_\nu \varphi$ to indicate that the formula φ is satisfied in the σ-structure $\mathcal{A}$ by the variable assignment ν. For the first-order connectives, this satisfaction relation is defined in the usual way. We just recall here that $\mathcal{A} \models_\nu (\mathrm{DTC}_{\vec{x},\vec{y}}\,\varphi)(\vec{s},\vec{t})$ holds if and only if the pair of tuples $([\![\vec{s}]\!]_\nu, [\![\vec{t}]\!]_\nu)$ is in the deterministic transitive closure of $\{(\vec{u},\vec{v}) \mid \mathcal{A} \models_{\nu[\vec{u}/\vec{x},\,\vec{v}/\vec{y}]} \varphi\}$, where we write $[\![t]\!]_\nu$ for the standard interpretation of the term t with respect to the variable assignment ν and extend this notation to tuples of terms in the evident way.

Deterministic transitive closure logic captures LOGSPACE on finite structures with a total ordering, i.e. where σ contains a binary relation $<$ that is interpreted as a total ordering, see e.g. [Ebbinghaus and Flum 1995]. Informally, this is because with a total ordering one can do enough arithmetic in the logic to encode work-tapes of LOGSPACE Turing Machines and thus simulate the computation of such machines using the DTC-operator.

On unordered structures, however, DTC-logic is strictly weaker than LOGSPACE. A typical example of a property that cannot be expressed without an ordering is that the universe has an even number of elements. If graphs are represented by an edge relation $E(-,-)$ and without any ordering, then DTC-logic on graphs is very weak. Grädel and McColm [1995] have shown that there exist families of graphs on which DTC without any ordering is no more expressive than first-order logic.

DTC logic without a total ordering 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 for any node v the binary relation $F(v,-,-)$ is a total ordering on $\{w \mid E(v,w)\}$. With such a graph representation, DTC can encode JAGs and it is strictly more expressive, since it allows first-order quantification [Etessami and Immerman 1995]. 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 of Tree Walking Automata [Neven and Schwentick 2000].

We next observe that PURPLE subsumes DTC-logic on locally ordered graphs, in the sense that PURPLE can evaluate queries in that logic.

PROPOSITION 4.1. *For each closed* DTC *formula φ on 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 $U(G)$.*

PROOF. We show the following stronger property by induction on φ. For each DTC formula φ and each assignment ν of the free variables $FV(\varphi)$ of φ to graph nodes, there exists a strongly terminating program P_φ such that for all ρ, ρ', q and q' satisfying $\forall x \in FV(\varphi).\, \exists i.\, \rho(x) = (\nu(x), i)$ and $(P_\varphi,\, q,\, \rho) \longrightarrow^*_{U(G)} (\texttt{skip},\, q',\, \rho')$ we have that $q'(\mathit{result}) = \text{true}$ holds if and only if $G \models_\nu \varphi$ does.

—φ is $\top$. Let P_φ be $(\mathit{result} := \text{true})$.

—φ is $\bot$. Let P_φ be $(\mathit{result} := \text{false})$.

—φ is $E(x, y)$. We can take P_φ to be the following program, in which we write $N(x)$ as an abbreviation for the loop $(\texttt{while}\ \neg(x = x.\mathit{prec})\ \texttt{do}\ x := x.\mathit{prec})$ that normalises x to represent a node in $U(G)$.

$$\begin{array}{l} \mathit{result} := \text{false};\ N(x);\ N(y); \\ \texttt{while}\ \neg(x = x.\mathit{next})\ \texttt{do}\ (x := x.\mathit{next};\ \mathit{result} := \mathit{result} \vee (y = x.\mathit{succ})) \end{array}$$

—φ is $F(x, y, z)$. For P_φ we write a program that iterates through the neighbours of x in ascending order and checks whether y appears before z. In it we use the notation $N(x)$ introduced in the previous case.

$$\begin{array}{l} \mathit{result} := \text{false};\ \mathit{seen\_z} := \text{false}; \\ N(x);\ N(y);\ N(z); \\ \texttt{while}\ \neg(x = x.\mathit{next})\ \texttt{do}\ (\\ \quad x := x.\mathit{next}; \\ \quad \mathit{seen\_z} := \mathit{seen\_z} \vee (z = x.\mathit{succ}); \\ \quad \mathit{result} := \mathit{result} \vee ((y = x.\mathit{succ}) \wedge \neg \mathit{seen\_z})) \end{array}$$

—φ is $\varphi_1 \wedge \varphi_2$. Let $\vec{x}$ be the free variables of φ. Let $\vec{z}$ be a vector of fresh variables of the same length as $\vec{x}$ that we use to save the initial values of $\vec{x}$. With these variables P_φ can be defined as the program

$$\vec{z} := \vec{x};\ P_{\varphi_1};\ \mathit{result}_1 := \mathit{result};\ \vec{x} := \vec{z};\ P_{\varphi_2};\ \mathit{result} := \mathit{result} \wedge \mathit{result}_1,$$

in which we write $\vec{z} := \vec{x}$ as an abbreviation for $z_1 := x_1; \ldots; z_n := x_n$.

—φ is $\varphi_1 \vee \varphi_2$. This case is similar to that for $\wedge$.

—φ is $\forall x.\, \psi$. Let $\vec{x}$ be the free variables of φ and let $\vec{z}$ be a vector of fresh variables of the same length. Then we can use the `forall`-loop to evaluate the universal quantifier and define P_φ to be:

$$\begin{array}{l} \vec{z} := \vec{x};\ \mathit{all} := \text{true}; \\ \texttt{forall}\ x\ \texttt{do}\ (P_\psi;\ \mathit{all} := \mathit{all} \wedge \mathit{result};\ \vec{x} := \vec{z}); \\ \mathit{result} := \mathit{all} \end{array}$$

—φ is $\exists x.\, \psi$. This case can be handled similarly to the case for universal quantification.

—φ is $(\mathrm{DTC}_{\vec{x},\vec{y}}\, \psi)(\vec{s},\vec{t})$. In this case we use a number of nested `forall`-loops to cycle through all tuples of elements in order to find the next tuple. We can assume that φ is the graph of a partial function. If it is not we can replace it in the original formula by $\varphi \wedge (\forall \vec{z}.\, \varphi[\vec{z}/\vec{y}] \Rightarrow \vec{y}{=}\vec{z})$. We can then take for P_φ the program below. In this program we write `forall` $\vec{x}$ `do` P as a shorthand for `forall` x_1 `do` ... `forall` x_n `do` P. We use similar notation for assignments to vectors of variables and use a generalisation to vectors of the equality test $P_=$ from Section 3.4. The vectors $\vec{a}$, $\vec{u}$ and $\vec{z}$ are all vectors of fresh variables of the same length as $\vec{x}$.

$\vec{x} := \vec{s}$;
result := false;
`forall` $\vec{u}$ `do` (/* Execute body $\geq |G|^n$ times */
  $\vec{z} := \vec{x}$;
  `forall` $\vec{y}$ `do` /* Find next step */
    ($\vec{x} := \vec{z}$; P_ψ; `if` *result* `then` $\vec{a} := \vec{y}$ `else skip`);
  $P_=(\vec{t},\vec{a},\mathit{found})$;
  $\mathit{result} := \mathit{result} \vee \mathit{found}$;
  $\vec{x} := \vec{a}$)

□

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 [Etessami and Immerman 1995].

When studying DTC-logic as a model for pointer programs, one may hope to obtain insights using the mathematical tools developed in finite model theory. For locally ordered DTC-logic, however, many tools are not available yet. In particular, at present there are no simple Ehrenfeucht-Fraïssé games for this logic; and such games are the main tool for proving inexpressivity results in finite model theory. Most of the existing results about locally ordered DTC-logic have been proved either directly or by reduction to a proof that uses automata-theoretic techniques. Etessami and Immerman [1995], for example, prove that undirected reachability cannot be expressed by a formula of the form $(\mathrm{DTC}_{\vec{x},\vec{y}}\, \varphi)(\vec{s},\vec{t})$, where φ is a first-order formula (without a total ordering not every formula can be expressed in this way). They do so by using the locality of first-order logic to reduce the problem to the corresponding result of Cook and Rackoff for JAGs.

The relative success of automata-theoretic methods is part of the motivation for studying the programming language PURPLE. Indeed, we have recently been able to show that PURPLE cannot express s-t-reachability in undirected graphs [Schöpp and Hofmann 2008], which by the above proposition implies the same for locally ordered DTC-logic. We do not know how to obtain this result directly using just logical methods.

Furthermore, when viewed as a model of pointer algorithms, DTC-logic is somewhat unnaturally restricted. To implement universal quantification, say on a Turing Machine with logarithmic space, one needs to have a form of iteration over all pos-

sible 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. The point is that 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. [Etessami and Immerman 1995; Immerman 1999]. 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 and Grädel, personal communication]. However, we believe that this problem is not expressible in PURPLE and in view of the Proposition 4.1 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 [Immerman 1999]. An order-independent query is a DTC formula that has access to a total order on the universe, but that must not depend on the particular choice of the order. 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 a variant 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 this version of PURPLE one can use the order to capture all of LOGSPACE, as is the case for order-independent queries.

In the next section we show that PURPLE cannot express arithmetic properties, such as that the size of the input structure is a power of two, which can be expressed in DTC-logic with numbers as well as with order-independent DTC-queries.

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 on $V_n = \{1, \ldots, n\}$, i.e. the graph with node set V_n that does not have any edges between its nodes. Since this graph has constant degree 0, the model with universe V_n induced by it can be assumed not to have any operations.

Fix a finite set M of pointer variables. We show that no program with pointer 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 the 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 the 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 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 ($z := x$; `forall` x `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 and 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 Definition 5.4 and Lemma 5.5 below.

Definition 5.1. 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 $x[\rho]y \iff \rho(x) = \rho(y)$. We call $[\rho]$ the *incidence relation* of ρ.

Definition 5.2. 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 5.3. Let $\rho, \rho'\colon M \to V_n$ 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 $\mathit{Vars}_B \to 2$ of boolean states.

Definition 5.4. 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, whenever $n \geq k$ and $q \in Q$ and $\rho\colon M \to V_n$ and $B(q, [\rho], n \bmod l) = (q', E', f)$ then $(P, q, \rho) \longrightarrow^*_{G_n} (\mathtt{skip}, q', \rho')$ for some ρ' compatible with (E', ρ, f).

Notice that in contrast to the definition of recognition we only require that *for some* evaluation of (P, q, ρ) 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 5.5. *Suppose (B, k, l) represents the behaviour of P on discrete graphs. Whenever $n \geq k$ and $q \in Q$ and $\rho\colon M \to V_n$ and $B(q, [\rho], n \bmod l) = (q', E', f)$ then $(P, q, \rho) \longrightarrow^*_{G_n} (\texttt{skip}, q', \rho')$ for all ρ' compatible with (E', ρ, f).*

PROOF. Choose τ compatible with (E', ρ, f) such that moreover $(P, q, \rho) \longrightarrow^*_{G_n} (\texttt{skip}, q', \tau)$ holds. Such τ must exist by the definition of "represents." We have $\tau(x) = \rho(f(x)) = \rho'(x)$ whenever $f(x) \neq \mathit{fresh}$ and $\tau(x), \rho'(x) \notin \mathrm{im}(\rho)$ whenever $f(x) = \mathit{fresh}$. Hence we can find an automorphism $\sigma : G_n \to G_n$ satisfying $\sigma \circ \rho = \rho$ and $\sigma \circ \tau = \rho'$. The claim then follows from Lemma 3.3. □

THEOREM 5.6. *There exist numbers k and l (depending on the number of variables in M) such that each `while`-free program P with pointer variables in M is represented by $([\![P]\!], k, l)$ for some function $[\![P]\!]$.*

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 $[\![P_1]\!](q, E, t) = (q_1, E_1, f_1)$ as well as $[\![P_2]\!](q_1, E_1, t) = (q_2, E_2, f_2)$. 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 $[\![P]\!](q, E, t) = (q_2, E_2, f)$. Fix some n with $n \bmod l = t$ and some $\rho\colon M \to V_n$ 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 5.5 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* $\texttt{if } s^B \texttt{ then } P_1 \texttt{ else } P_2$. On discrete graphs, the meaning of the boolean term s^B depends only on the values of the boolean variables and the incidence relation of the pointer variables. Write $[\![s^B]\!]_{q,E}$ for the value of $[\![s^B]\!]_{q,\rho}$ for any ρ with $[\rho] = E \in \Sigma$. With this notation, we can define $[\![P]\!]$ as follows.

$$[\![P]\!](q, E, t) = \begin{cases} [\![P_1]\!](q, E, t) \text{ when } [\![s]\!]_{q,E} = \text{true}, \\ [\![P_2]\!](q, E, t) \text{ when } [\![s]\!]_{q,E} = \text{false}. \end{cases}$$

— *Case* $\texttt{forall } x \texttt{ do } P_1$. 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 $\rho\colon M \to V_n$ with $[\rho] = E$

and assume without loss of generality that $\mathrm{im}(\rho) \subseteq \{1, \ldots, |M|\}$. We then iterate through the graph nodes $\{1, \ldots, n\}$ in ascending order, choosing fresh nodes from $\{|M|+1, \ldots, 3|M|\}$. Since there are only $|M|$ variables we have enough space in this interval so as to satisfy any request for fresh nodes possibly arising during the evaluation of P_1. Formally, we choose sequences $(\rho_i)_{0 \le i \le n}$ and $(q_i)_{0 \le i \le n}$ in such a way that
(1) $\rho_0 = \rho$, $q_0 = q$;
(2) $(P_1, q_i, \rho_i[x \mapsto i{+}1]) \longrightarrow^*_{G_n} (\mathtt{skip}, q_{i+1}, \rho_{i+1})$;
(3) for all $y \in M$, $\rho_{i+1}(y) \notin \mathrm{im}(\rho_i[x \mapsto i{+}1])$ implies $\rho_{i+1}(y) \in \{|M|{+}1, \ldots, 3|M|\}$.
That such sequences exist follows from the induction hypothesis and Lemma 5.5.
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 $m > n \ge k$ with $m \bmod l = t = n \bmod l$ and $\chi \in M \to V_m$ with $[\chi] = E$.
In view of Lemma 3.3 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 $(\chi_i)_{0 \le i \le n}$ and $(r_i)_{0 \le i \le n}$ such that
(1) $\chi_0 = \chi$, $r_0 = q$;
(2) $(P_1, r_i, \chi_i[x \mapsto i{+}1]) \longrightarrow^*_{G_m} (\mathtt{skip}, r_{i+1}, \chi_{i+1})$;
(3) for all $y \in M$, $\chi_{i+1}(y) \notin \mathrm{im}(\chi_i[x \mapsto i{+}1])$ implies $\chi_{i+1}(y) \in \{|M|{+}1, \ldots, 3|M|\}$;
(4) $\chi_i(y) = \rho_i(y)$ and $q_i = r_i$ for all $y \in M$ and $i \le n$.
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 all $i > 3|M|$ and 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 r_i and t, of course). Indeed, for $i > 3|M|$ we have

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

In view of the choice of N there must exist indices $3|M| < t < t' \le 3|M| + N = k$ 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 $d \ge 0$ with $t' + d \le m$. 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 $t' \le k \le n \le i < i' \le m$ 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,

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

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

□

COROLLARY 5.7. *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 Definition 3.2. By Lemma 3.4 we can assume that P is `while`-free. Then Theorem 5.6 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 $q'(\mathit{result}) = \text{true}$. Now let χ be a valuation of the variables in G_{n+l} satisfying $[\rho] = [\chi]$. We then get $(P,\, q,\, \chi) \longrightarrow^*_{G_{n+l}} (\texttt{skip},\, q',\, \_)$, 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 Definition 3.2. □

We remark that Theorem 5.6 implies that if X is a property of discrete graphs that is recognised by some PURPLE program then the set $\{a^n \mid G_n \in X\}$ is a regular set over the unary alphabet $\{a\}$. Moreover, it is easy to see that every regular set over this alphabet arises in this way.

A reader of an earlier version of this paper suggested that this alternative viewpoint yields an almost trivial proof of Theorem 5.6: 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 recall 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 5.6 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 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 s-t-reachability in undirected graphs uses counting registers of logarithmic size. Indeed, it is not possible to solve undirected reachability in PURPLE and therefore not in DTC-logic [Schöpp and Hofmann 2008].

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 Definition 3.2 captures this and, as argued at the end of Section 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 5.5 suggest some rather unexpected connection to the semantic study of name generation and α-conversion [Gabbay and Pitts 2002; Pitts and Stark 1993]. It remains to be seen whether this is merely coincidence or whether techniques and results can be fruitfully transferred in either direction.

ACKNOWLEDGMENT

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

REFERENCES

Cook, S. and McKenzie, P. 1987. Problems complete for deterministic logarithmic space. *Journal of Algorithms 8,* 3, 385–394.

Cook, S. and Rackoff, C. 1980. Space lower bounds for maze threadability on restricted machines. *SIAM Journal of Computing 9,* 3, 636–652.

Ebbinghaus, H. and Flum, J. 1995. *Finite Model Theory.* Springer-Verlag.

Etessami, K. and Immerman, N. 1995. Reachability and the power of local ordering. *Theoretical Computer Science 148,* 2, 261–279.

Gabbay, M. and Pitts, A. 2002. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing 13*, 341–363.

Gonthier, G. 2005. A computer-checked proof of the four-colour theorem. Available at `http://research.microsoft.com/~gonthier`.

Grädel, E. and McColm, G. 1995. On the power of deterministic transitive closures. *Information and Computation 119,* 1, 129–135.

Immerman, N. 1999. *Descriptive Complexity.* Springer-Verlag.

Johannsen, J. 2004. Satisfiability problems complete for deterministic logarithmic space. In *STACS*. 317–325.

Neven, F. and Schwentick, T. 2000. On the power of tree-walking automata. In *ICALP*. 547–560.

Pitts, A. and Stark, I. 1993. Observable properties of higher order functions that dynamically create local names, or: What's new? In *MFCS93*. LNCS 711. Springer, 122–141.

Reingold, O. 2005. Undirected st-connectivity in log-space. In *STOC05*. 376–385.

Schöpp, U. and Hofmann, M. 2008. Pointer programs and undirected reachability. *Electronic Colloquium on Computational Complexity (ECCC) 15,* 090.

Received September 2008; accepted February 2009