MIT/LCS/TM-157

ON THE EXPRESSIVE POWER OF DYNAMIC LOGIC

Albert R. Meyer
Karl Winklmann

February 1980

On The Expressive Power Of Dynamic Logic

Albert R. Meyer and Karl Winklmann
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

February 25, 1980

Abstract. We show that "looping" of while-programs can be expressed in Regular First Order Dynamic Logic, disproving a conjecture made by Harel and Pratt.

In addition we show that the expressive power of quantifier-free Dynamic Logic increases when nondeterminism is introduced in the programs that are part of formulae of Dynamic Logic. Allowing assignments of random values to variables also increases expressive power.

This research was sponsored in part by National Science Foundation Grants MCS77-19754 and MCS77-19754A01. A preliminary version of this paper has appeared in *Proceedings of the Eleventh Annual ACM Conference on Theory of Computing,* Atlanta, May 1979.

1. Introduction

Dynamic Logic has been introduced by V. R. Pratt [Pratt 1976] as a formalism for reasoning about programs. Pratt has observed that Dynamic Logic provides simple expressions for an assortment of familiar properties of programs: equivalence, termination, partial correctness, and determinacy, for example. However, there remains a general question of the extent to which other interesting properties of programs can be expressed in the formalism.

One property whose expressibility in First Order Dynamic Logic (DL) has been an open question, first suggested to us by M. J. Fischer, is the property of "looping." A nondeterministic program is said to "loop" if its execution tree has an infinite path. The concept of looping is a basic one in certain definitions of total correctness of programs [Harel 1979, Dijkstra 1976, Greif-Meyer 1979, Hoare 1978], and the desire for a formal system suitable for reasoning about looping motivated the introduction of an augmented version of DL, called DL^+, in which an assertion that a program loops is added as an explicit primitive [Harel-Pratt 1978, Harel 1979].

In Section 3 we show that DL as originally formulated was in fact powerful enough to express looping. Thus the introduction of DL^+ was not necessary to obtain the desired expressive power. However, it is obvious, as we observe in Section 2.3, that there is no single formula scheme of DL which uniformly expresses looping of an arbitrary program. So the introduction of DL^+ can still be justified on the ground that it provides such a uniform expression of looping.

In Section 4 we study the expressive power of nondeterminism in the context of DL. Since programs appear as syntactic objects within formulae of DL, one can compare the expressive power of two versions of DL which differ only in that one version uses nondeterministic programs while the other does not. This comparison of *expressive* power is substantially different from the standard problem of comparing the *computational* power of deterministic and nondeterministic programs, not the least difference being that we can obtain definite results. We prove that for formulas without quantifiers, allowing nondeterministic programs increases expressive power. In Section 5 we consider DL with programs containing "random assignment" statements and show that expressive power is again increased.

2. Syntax and Semantics of DL

Briefly, DL is First Order Predicate Calculus augmented by a construct $\langle\alpha\rangle P$ with the meaning that there is an execution of program α after which P holds. The class of programs we consider differs slightly from the class used in [Harel 1979,Pratt 1976]. Specifically, our programs use **if-then** and **while-do** constructs rather than set union and Kleene star. However, none of our proofs depends on these differences in terminology. The definitions given in the remainder of this Section are intended to make this paper self-contained.

2.1 Syntax of DL

The *symbols* used in the language of DL contain the usual assortment of symbols from First Order Predicate Calculus: predicate symbols like p, q, r, p_1, ... ; function symbols like f, g, h, f_1, ..., each with an associated nonnegative integer arity; and the special symbols $\neg$, $\vee$, $\exists$,), (, and = . Zeroary function symbols are called variables, and zeroary predicate symbols are called Boolean variables. Typical variables are x, y, z, x_1, ... , and typical Boolean variables are p, q, r, ... In addition DL uses the pair of symbols < > (pronounced

"diamond"), and a few special symbols which are used in programs: :=, **while**, **do**, **if**, **then**, **true**, **false**, and **choice**. Symbols which are strictly speaking not in the language but serve to abbreviate formulae are $\forall$, $\equiv$, $\Rightarrow$ and [ ] ("box"). [ ] is an abbreviation for $\neg< >\neg$, just as $\forall$ is an abbreviation for $\neg\exists\neg$.

Terms and *atomic formulae* are formed exactly as in First Order Predicate Calculus with Equality. *Formulae* are also formed exactly as in First Order Predicate Calculus except that $<\alpha>$ may be used in place of $\exists x$, where α is any program in a simple programming language to be defined below. Thus formulae are characterized inductively as follows:

any atomic formula is a formula;

for any two formulae P and Q, variable x, and program α, the following are formulae:

$\neg P$, $(P \vee Q)$, $\exists x P$, and $<\alpha>P$.

Formulae of DL^+ are characterized in the same way except that for each program α there is a designated zeroary predicate symbol $Loop_\alpha$ which never appears in any program. (This is not the definition given for DL^+ in [Harel-Pratt 1978], but it will be easy to verify that our definition is merely a notational variant of theirs, and so both versions of DL^+ have the same expressive power (cf. [Harel 1979]).)

Our programs are familiar **if-then-while** schemes with the addition of a **choice**-statement where **choice**(α,β) means nondeterministically choose to do either α or β. The following BNF-description is a convenient way to define the syntax of these programs.

<program> ::= <statement>; <program> | <statement>
<statement> ::= <assignment statement> | <if-statement> |
<while-statement> | <choice-statement>
<assignment statement> ::= <variable> := <term> |
<Boolean variable> := <test>
<if-statement> ::= **if** <test> **then** <program> **fi**
<while-statement> ::= **while** <test> **do** <program> **od**
<choice-statement> ::= **choice**(<program>,<program>)
<test> ::= <open formula of predicate calculus with equality> |
true | **false** .

We refer to this class of programs as while-programs. The formal definition of their semantics will be given in Section 2.2.

It will be convenient to have a tree-representation of such programs. For each program α we define a (rooted) tree T_α. Informally, this tree T_α is just the flowchart of the program α unwound into an infinite tree in the obvious way (see e.g. [Greibach 1975]). Nodes which correspond to a nondeterministic choice (in the execution of a choice-statement) are labeled **choice**. More precisely, T_α is defined by induction on α as follows: If α is an assignment statement, then T_α is the tree in Figure 1, where the dashed arrow indicates the root of T_α.

For a choice-statement, $T_{\mathbf{choice}(\alpha,\beta)}$ is the tree in Figure 2.

For an if-statement, $T_{\mathbf{if}\ B\ \mathbf{then}\ \alpha}$ is the tree in Figure 3.

For any two trees S and T, let $S \circ T$ denote S with each leaf (i.e., Halt-node) replaced by a copy of T. Then for a while-statement, $T_{\mathbf{while}\ B\ \mathbf{do}\ \alpha\ \mathbf{od}}$ is the (unique) infinite tree which satisfies the equation α is the (unique) infinite tree which satisfies the equation in Figure 4. If α is a list of statements $s_1;s_2;\ldots;s_k$, then T_α is the tree $T_{s_1} \circ T_{s_2} \circ \ldots \circ T_{s_k}$.

2.2 Semantics of DL

2.2.1 Informal presentation

Note that what we have called while-programs are actually program *schemes*, because they contain uninterpreted function and predicate symbols. By giving an interpretation I to all those symbols (i.e. by defining a *state*), the schemes are made into programs and it is obvious how to execute them. During execution of such a program, the interpretations (values) of some symbols may change. Thus, the execution of the statement x:=f(y) will in general change the value of the symbol x.

While-programs as defined above will always leave functions and predicates of positive arity unchanged. However, later in the paper we consider "array assignments", like f(x):=y, which do change the interpretation of function symbols of positive arity. Thus there is no reason to distinguish function symbols from variables, and we merge the standard concepts of a *structure*, which provides an interpretation for all function and predicate symbols, and a *valuation*, which assigns values to all variables, into the concept of a *state*, which gives interpretations of all symbols.

For any state I and program α, let $\alpha(I)$ be the set of states in which α, when started in state I, can terminate. Let $m(\alpha) = \{(I,J):J\in\alpha(I)\}$. This relation $m(\alpha)$ captures the "input-output behavior" of the program α. Now to say that after executing α starting in state I, it is possible to halt and have P hold true, which we express in symbols as $I\models\langle\alpha\rangle P$, is the same as saying that there is a state $J\in\alpha(I)$ for which P is true. (The assertion $J\in\alpha(I)$ is of course equivalent to the assertion $(I,J)\in m(\alpha)$. It will be convenient to have both $m(\alpha)$ and $\alpha(I)$ defined, although one of them would clearly be sufficient.) The semantics of formulae of forms other than $\langle\alpha\rangle P$ are defined as in First Order Predicate Calculus.

Looping of nondeterministic programs is a notion which is independent of their input-output behavior, viz., it cannot be defined in terms of the relation

m(α). For a program, α, define the predicate $Loop_\alpha$ to be true in a state I, that is $I \models Loop_\alpha$, if the execution tree describing the possible computations of α started in state I, has an infinite path.

2.2.2. Formal Definitions

A *state* I is a mapping of all predicate symbols, function symbols, and (Boolean) variables to predicates, functions, and (truth) values on some domain D. The mapping of predicates etc. to the symbols observes the arity of the symbols: every k-ary predicate symbol p is assigned a predicate p_I on D^k, every k-ary function symbol f is assigned a function f_I from D^k to D; in particular, for k=0, every Boolean variable p is assigned a truth value p_I, and every variable x is assigned a value $x_I \in D$. The symbol = is always interpreted as equality. Given the values p_I, ... ,f_I, ... , and x_I, ... of all symbols p, ... ,f, ... , and x, ... , the values t_I of terms t and the truth values of all program-free formulae under I are defined in the standard way. As usual we write $I \models P$ if a formula P is true in state I, and $\models P$ if P is true in all states.

The *execution tree* $T_\alpha(I)$ of a programα in a state I is obtained from T_α (defined in Section 2.1 above). It consists of the subtree of T_α whose nodes receive labels according to the following inductively defined procedure:

1. The root is labeled with the state I.

2. For any node which contains an assignment statement, if the node is labeled J then its son is labeled by the state K which results from executing the assignment statement in state J. Namely, K agrees with J on the interpretation of all symbols except for the variable to the left of the symbol ":=" in the assignment statement. The value in K of the variable to the left of ":=" is the value in J of the term to the right.

3. For any node containing choice, if the node is labeled J then both of its sons are also labeled J (i.e., nondeterministic choices do not change the state).

4. For any node containing a test B, if the node is labeled with a state satisfying B, then the son pointed to by the arrow labeled true receives the same state label, and the false son receives no state label. Symmetrically, if the state label does not satisfy B, then the false son receives the same label and the true son remains unlabeled.

Let $\alpha(I) = \{J : T_\alpha(I)$ has a leaf labeled $J\}$, and define $I \models \langle\alpha\rangle P$ for an arbitrary formula P iff there is a state $J \in \alpha(I)$ and $J \models P$. The definition of $I \models Q$ for formulae Q not of the form $\langle\alpha\rangle P$ is the standard inductive definition from First Order Predicate Calculus. This completes the definition of the semantics of DL.

For notational convenience we define $m(\alpha)$ for a program α to be

$$\{(I,J) : J \in \alpha(I)\},$$

i.e. $\alpha(I)$ is the set of states in which program α can terminate when started in state I, and $m(\alpha)$ is the input-output relation of α.

Finally, we define the predicate $Loop_\alpha$ to hold in a state I iff $T_\alpha(I)$ has an infinite path.

It is easy (and will be useful) to give an equivalent definition of $Loop_\alpha$ by induction on the structure of α (cf. [Harel 1979, pp.92-93]). For all states I, programs α, β, γ, and tests B, we define:

If α is an assignment-statement then $I \models \neg Loop_\alpha$;

if α is choice(β,γ) then $I \models Loop_\alpha$ iff $I \models (Loop_\beta \vee Loop_\gamma)$;

if α is $\beta;\gamma$ then $I \models Loop_\alpha$ iff $I \models (Loop_\beta \vee \langle\beta\rangle Loop_\gamma)$;

if α is if B then β fi then $I \models Loop_\alpha$ iff $I \models (B \wedge Loop_\beta)$;

if α is while B do β od then $I \models Loop_\alpha$ iff $I \models (Local\text{-}Loop_\alpha \vee Global\text{-}Loop_\alpha)$

where

$I \models Local\text{-}Loop_\alpha$ is defined by

"There is a finite sequence of states $J_0, J_1, \ldots, J_k$ such that $I=J_0$, $J_{i+1} \in \beta(J_i)$ for all $0 \leq i < k$, $J_i \models B$ for all $0 \leq i \leq k$, and $J_k \models Loop_\beta$."

and $I \models Global\text{-}Loop_\alpha$ is defined by

"There is an infinite sequence of (not necessarily distinct) states $J_0, J_1, \ldots$ such that $I=J_0$, and $J_{i+1} \in \beta(J_i)$ and $J_i \models B$ for all $i \geq 0$."

2.3 The general problem of expressing looping in DL

Before we go into the technical details in the next section we wish to point out that in our formulation of the problem of expressing $Loop_\alpha$ in DL we have made two apparently arbitrary choices, namely, our choice of while-programs, α, and our definition of the associated trees T_α.

The definition of the semantics of $\langle\alpha\rangle P$ makes sense for any syntactic object α which names a binary relation on states, although we have chosen to restrict ourselves to the relations $m(\alpha)$ named by while-programs. However, because the semantics of the statement $\langle\alpha\rangle P$ depends only on the *relation* associated with program α, as opposed to the *syntax* of α, there is no loss in generality in our choice of while-programs compared to the class of regular programs considered in previous studies [Pratt 1976, Harel 1979]. In particular, the relations definable by while-programs are precisely the "regular" relations definable by regular expressions over the alphabet of statements and tests as in [Pratt 1976, Harel 1979], or equivalently the relations definable by nondeterministic flowchart schemes over these statements and tests.

We have chosen our while-program syntax in order to emphasize the converse fact that looping is a property of programs, or more precisely a property of the trees associated with programs, as opposed to just the relations named by the programs. Thus, if the relations associated with α and α' are the same, then by definition of the semantics of DL, α may be replaced by α' in any formula of DL and the meaning of the formula remains unchanged. This implies that there cannot be a single DL-formula F with a "program variable," say A, such that for any program α, the formula F with A replaced by α would express $Loop_\alpha$. To see this consider the two programs α and α' where α is : p:=false and α' is: p:=true; **while** p **do choice**(p:=true, p:=false) **od**.

These two programs α and α' have the same semantics, i.e. $m(\alpha)=m(\alpha')$, and are therefore "indistinguishable" to any formula F of the kind described. At the same time they satisfy $\models\neg Loop_\alpha$ and $\models Loop_{\alpha'}$.

We hope that the simple definition of T_α from α will persuade the reader that the question of whether T_α has an infinite path accurately reflects the intuitive idea that α "loops," that is, that one of the nondeterministic computations by α may not terminate.

There may be some loss of generality in our particular definition of T_α. Not all computation trees obtainable by similar, intuitively simple inductive definitions aimed at defining looping for various classes of programs are in fact obtainable as T_α for while-programs α.[1] However, our proofs that looping is expressible in DL do not involve such a loss in generality, and in fact apply to the more general class of *regular trees* whose edges are labeled with assignments and tests. These can be defined as state transition trees of finite automata each of which has as alphabet some finite set of assignments and tests.[2] It is easily seen that T_α is regular for any while-program α, and we believe that any sensible formulation of looping for while-programs, flowchart schemes, etc. will naturally yield only regular trees.

Thus our result in its general form can be summarized informally as saying that looping, i.e. having infinite paths, of regular trees is definable in

regular DL. This naturally raises the question of what happens when we enrich the class of programs and associated trees, e.g., we can ask whether looping of context-free (or recursively enumerable) trees is expressible in context-free (or recursively enumerable) DL (cf. [Harel 1979]). for that matter, we can ask whether looping of r.e. trees is expressible in regular DL. The context-free case and the r..e. tree in regular DL question remain open, although we conjecture that the techniques developed below will extend to yield a positive answer in the context-free case. In the case that r.e. programs are allowed, i.e., in r.e. DL, the construction of formulas that express looping presents no difficulty: for each r.e. program α there is another r.e. program β which, informally speaking, carries out a depth-first search of the execution tree of α. Thus $Loop_\alpha$ is equivalent to $[\beta]$ false.[3] No such simple observation seems to settle the problem of expressing looping in regular DL which we consider next.

3. Expressing $Loop_\alpha$ in DL and in DL with array assignments

In this Section we first show that for any while-program α, the predicate $Loop_\alpha$ can be expressed by a formula of DL (Theorem 1). Then we show that this remains true if we allow "array assignments" in while-programs (Theorem 2). Note that Theorem 2 is by no means a corollary to Theorem 1. Although allowing array assignments adds to the power of the language of DL, it is also true that expressing $Loop_\alpha$ for programs α with array assignments is a more general problem than doing it for programs without array assignments. Indeed our proof of Theorem 2 is quite different from our proof of Theorem 1.

3.1 Expressing $Loop_\alpha$ in DL

The crucial part of the proof of Theorem 1 is a characterization lemma about infinite, finitely branched "transition trees" (Lemma 1), which we prove first. Transition trees are the kind of trees one gets by "unwinding" finite or infinite "state transition diagrams," which are typically in the form of arbitrary directed graphs. More formally, a *transition tree T* is a tree with labeled nodes

which has the property that any two nodes with equal labels have isomorphic subtrees, and sibling nodes have distinct labels. (Transition trees may have infinitely many labels. It happens that regular trees mentioned in Section 2.3 coincide with transition trees with finitely many labels.) T is *finitely branched* if each node has only finitely many sons. We shall use *depth(n)* to denote the depth of a node n in T, with the depth of the root being 0; we shall use *label(n)* to denote the label of a node n, $Labels_T$ to denote the set of all labels occurring in T, and $Nodes_T$ to denote the set of all nodes in T. For any $l \in Labels_T$ let

$$min\text{-}depth(l) = \min \{depth(n) : n \in Nodes_T \wedge label(n)=l\}$$

and similarly for *max-depth*. If there is no bound on the depths of occurrences of l in T then $max\text{-}depth(l)=\infty$. We write $max\text{-}depth(l)<\infty$ if *max-depth(l)* is bounded.

For any transition tree T we define

$$A_T = \{l \in Labels_T : \exists l' \in Labels_T\ (max\text{-}depth(l) < min\text{-}depth(l'))\}.$$

Note that if $l \in A_T$, then l must not occur at arbitrarily deep levels in T, i.e. *max-depth(l)* must be defined.

Lemma 1: A finitely branched transition tree T is infinite if and only if

(i) two nodes along some path in T have the same label,

or (ii) A_T is infinite.

Proof:

1. It is easy to see that (i) and (ii) each imply infinity of T: If there is a repetition of a label along a path in T, then by the definition of transition trees, T contains a subtree which properly contains a copy of itself and hence is infinite; and if A_T is infinite then T is infinite because A_T is a subset of the labels occurring in T.

2. To show that T being infinite implies (i) or (ii) we assume

(1) T is infinite, and

(2) (i) does not hold, i.e. no label repeats along any path in T,

and conclude that A_T must be infinite.

Since T is finitely branched, Konig's Lemma is applicable and (1) is therefore equivalent to

(3) T contains an infinite path,

which together with (2) implies

(4) $Labels_T$ is infinite.

Since T is finitely branched only finitely many labels can occur up to any given depth. This together with (4) implies that "new" labels keep showing up at arbitrarily deep levels, i.e.

(5) $\{min\text{-}depth(l) : l \in Labels_T\}$ is unbounded,

which together with the definition of A_T implies

(6) $A_T \supseteq \{l \in Labels_T : max\text{-}depth(l) < \infty\}$.

(In fact, $A_T = \{l \in Labels_T : max\text{-}depth(l) < \infty\}$, but we do not need this fact.) We finish the proof by showing that

(7) $\forall d \in \mathbb{N}\ \exists l \in Labels_T\ (d = max\text{-}depth(l))$.

To prove (7), let $d \in \mathbb{N}$ and $L_d = \{label(n) : n \in Nodes_T \wedge depth(n) = d\}$. Define a relation $<$ on $Labels_T$ by

> $l < l'$ iff there is a path from the root in T along which l occurs after (i.e. deeper than) l'.

By the definition of transition trees, the relation $<$ is transitive, and by assumption (2) it is loop-free, hence a partial ordering. So there is a (not necessarily unique) maximal element in the finite set L_d, i.e. there is a label $l \in L_d$ which does not occur on any path below any of the labels in L_d. Since L_d contains the labels of all the nodes of depth d, label l does not occur below depth d. Hence $max\text{-}depth(l) \leq d$, and because $l \in L_d$, we have $max\text{-}depth(l)=d$. □

Lemma 2: Let T be a finitely branched transition tree. Then A_T is infinite iff

(8) $\quad A_T \neq \varnothing \wedge \forall l \in A_T \, \exists l' \in A_T \, (max\text{-}depth(l) < min\text{-}depth(l'))$.

Proof: Clearly, (8) implies that A_T is infinite. Conversely, assume A_T is infinite, and let l be any element of A_T. Then $max\text{-}depth(l) < \infty$, since otherwise $l \notin A_T$. Since T is finitely branched, there are only finitely many labels $l'' \in Labels_T$ with $max\text{-}depth(l) \geq min\text{-}depth(l'')$. Since $A_T \subseteq Labels_T$ and A_T is infinite, there must be a label (in fact, infinitely many labels) $l' \in A_T$ with $max\text{-}depth(l) < min\text{-}depth(l')$. □

Theorem 1: For each while-program α there is a DL-formula equivalent to the predicate $Loop_\alpha$.

Proof: We define the desired formula L_α inductively following the inductive definition of the semantics of $Loop_\alpha$ given at the end of Section 2.2. The only problem lies in translating the definitions of $Local\text{-}Loop_\alpha$ and $Global\text{-}Loop_\alpha$ into DL-formulae when α is **while** B **do** β . For $Local\text{-}Loop_\alpha$ this is actually easy to do assuming, by induction, that we can express $Loop_\beta$. Namely, as the reader can verify, $Local\text{-}Loop_\alpha$ is equivalent to

$<$ **choice(p:=true, p:=false); while** (p$\wedge$B) **do** β; **choice(p:=true, p:=false)** $>$ (B$\wedge L_\beta$) ,

where p is a Boolean variable not occurring in α.

Global-Loop$_\alpha$ poses a harder problem. In fact we shall only show how to express *Global-Loop*$_\alpha$ under the assumption that *Local-Loop*$_\alpha$ does not hold, i.e., we shall prove

Lemma 3: There is a DL-formula GL_α such that

$$\models(\textit{Local-Loop}_\alpha \vee (GL_\alpha \equiv \textit{Global-Loop}_\alpha)) .$$

Note that for such a formula GL_α, (*Local-Loop*$_\alpha \vee$ *Global-Loop*$_\alpha$) is equivalent to (*Local-Loop*$_\alpha \vee GL_\alpha$) . Assuming Lemma 3, we can thus define the desired formula L_α following the inductive definition of the semantics of *Loop*$_\alpha$ given at the end of Section 2. This finishes the proof of Theorem 1 except for a proof of Lemma 3. □

Proof of Lemma 3: For any program α of the form **while** B **do** β **od** and any state I, consider the smallest tree $S_\alpha(I)$ whose root is labeled by I and which has an edge from a node labeled J to a node labeled K iff $J \models B$ and $K \in \beta(J)$. (Intuitively, $S_\alpha(I)$ is obtained from $T_\alpha(I)$ by ignoring all the steps "inside" any pass through β, recording entire passes through β as single nodes.) Note that *Global-Loop*$_\alpha$ is equivalent to $S_\alpha(I)$ having an infinite path, with the states J_0, J_1, ... mentioned in the definition of *Global-Loop*$_\alpha$ being the labels along that infinite path.

Claim: $S_\alpha(I)$ is a transition tree, and if $I \models \neg$*Local-Loop*$_\alpha$ then $S_\alpha(I)$ is finitely branched.

Proof of Claim: By the definition of $S_\alpha(I)$, the number of sons of a node and their labels are determined solely by the label of the node. This makes $S_\alpha(I)$ a transition tree.

If $I \models \neg Local\text{-}Loop_\alpha$, assume that $S_\alpha(I)$ has a node with infinitely many sons. Let J be the label of such a node. Then the program α can enter that state after some number of passes through β and on the next pass through β can reach infinitely many different states. Hence, since $T_\beta(J)$ is finitely branched, it must have an infinite path, i.e. $J \models Loop_\beta$, which implies by definition that $I \models Local\text{-}Loop_\alpha$. This finishes the proof of the Claim.

This Claim makes Lemmas 1 and 2 applicable and we can finish this proof by defining GL_α so that $I \models GL_\alpha$ iff

(A) Two nodes along a path in $S_\alpha(I)$ have the same label
or
(B1) $(A_{S_\alpha(I)} \neq \emptyset$
and
(B2) $\forall K \in A_{S_\alpha(I)}\ \exists L \in A_{S_\alpha(I)}\ (max\text{-}depth(K) < min\text{-}depth(L))$),

for all states I where $S_\alpha(I)$ is finitely branched. This is just a "programming exercise" in the language of DL, which we carry out by gradually rewriting (A), (B1), (B2) into DL-formulae. For notational convenience, let $v_1, \ldots, v_s$ be all the Boolean variables of α, and let $v_{s+1}, \ldots, v_{s+m}$ be all the other variables of α. For any state I, all the states in $S_\alpha(I)$ differ at most in the values of $v_1, \ldots, v_{s+m}$ and we can therefore identify each state in $S_\alpha(I)$ by the s+m-tuple of these values. We call the s+m-tuple $(v_1, \ldots, v_{s+m})$ a "state variable" and denote it by V. We shall use other state variables, denoted by J, K, K′, V′ etc., in various places. These are s+m-tuples of variables, all of which are pairwise different and, except for $v_1, \ldots, v_{s+m}$, are different from all symbols in α. Tests like V=J and assignments like L:=K are abbreviations for

$$v_1 \equiv j_1 \wedge \ldots \wedge v_s \equiv j_s \wedge v_{s+1} = j_{s+1} \wedge \ldots \wedge v_{s+m} = j_{s+m}$$

and

$$l_1 := k_1;\ l_2 := k_2;\ \ldots;\ l_{s+m} := k_{s+m}$$

respectively. For any state variable J, let α_J, β_J and B_J stand for α, β and B with each v_i replaced by j_i, $1 \le i \le s+m$. Again, let p be a Boolean variable not occurring in α and let BUZZ be the program **while true do x:=x od**.

As a first refinement of the above formula we get

(A′) (< **choice(p:=true, p:=false);**
while (p ∧ B) **do** β; **choice(p:=true, p:=false) od;**
if ¬B **then** BUZZ **fi;**
V′:=V;
β;
choice(p:=true, p:=false);
while (p ∧ B) **do** β; **choice(p:=true, p:=false) od>** V=V′)

∨

(B1′) ($\exists K(K \in A_{S_\alpha(I)})$

∧

(B2′) $\forall K\ \exists L((K \in A_{S_\alpha(I)}) \Rightarrow (L \in A_{S_\alpha(I)} \wedge \mathit{max\text{-}depth}(K) < \mathit{min\text{-}depth}(L))))$.

We now replace every subformula of the form "$C \in A_{S_\alpha(I)}$" by

$$\exists N \in \mathit{Labels}_{S_\alpha(I)}\ (\mathit{max\text{-}depth}(C) < \mathit{min\text{-}depth}(N))$$

and then replace every subformula of the form "$\mathit{max\text{-}depth}(C) < \mathit{min\text{-}depth}(D)$" by

$$\neg\ \langle\gamma\rangle(C=C' \wedge D=D')$$

where γ is a program which searches the tree $S_\alpha(I)$ for an occurrence of D with an occurrence of C at an equal or greater depth. The program γ performs this search by first running two copies, $\alpha_{C'}$ and $\alpha_{D'}$ of α for an equal number of passes through their while-statements thereby setting C′ and D′ to labels occurring at the same depth. Then γ runs $\alpha_{C'}$ for zero or more extra passes through its while-statement, thereby setting C′ to a label occurring at least as deeply as D′. Specifically, the program γ can be taken to be

C′:=V;
D′:=V;
choice (p:=true; p:=false);
while ($p \wedge B_{C'} \wedge B_{D'}$) do ($\beta_{C'}$; $\beta_{D'}$; choice (p:=true, p:=false) od;
choice (p:=true; p:=false);
while ($p \wedge B_{C'}$) do $\beta_{C'}$ od.

Finally we replace all subformulae of the form "$\exists N \in Labels_{S_\alpha}(I)$" by $\langle\delta\rangle$ where δ is the program

N:=V;
choice(p:=true, p:=false);
while ($p \wedge B_N$) do β_N; choice(p:=true, p:=false) od.

All these substitutions preserve meaning, a claim whose verification we leave to the reader. The resulting formula is the desired formula GL_α. This finishes the proof of Lemma 3. □

We conclude immediately

Corollary 1: For every DL$^+$-formula there is an equivalent DL-formula.

3.2 Expressing $Loop_\alpha$ in DL with array assignments

Array assignment statements are of the form <term> := <term>. Execution of an array assignment statement, say of $f(t_1, \ldots, t_k) := t$, changes the interpretation of f at the single point specified by the values of $t_1, \ldots, t_k$. The definitions of T_α and $T_\alpha(I)$ from Section 2 can readily be modified to account for this. Specifically, if a node of $T_\alpha(I)$ contains the statement $f(t_1, \ldots, t_k) := t$ and is labeled by a state J then its son is labeled by the state K which agrees with J on all interpretations except that $f_K((t_1)_J, \ldots, (t_k)_J) = t_J$. We denote this version of DL by DL_{arrays}.

Theorem 2: For each while-program α (with array assignments) there is a DL_{arrays}-formula L_α such that L_α is equivalent to $Loop_\alpha$.

Proof: As in the proof of Theorem 1 the only difficulty lies in expressing the predicate $Global\text{-}Loop_\alpha$ for α of the form **while** B **do** β **od**. This, however, is the end of the analogy between the two proofs. The reason why the techniques from the proof of Theorem 1 do not work here is the fact that the states which can be reached by an unbounded number of repetitions of β cannot any longer be described by the values of a bounded number of variables. Repeated array assignments may change the values of function symbols of positive arity at some unbounded number of points. Hence a new approach is needed, which we now outline.

First we define for each program α and state I the set $Dom_I(\alpha) = \{ t_I : t$ is a term formed from symbols that occur in $\alpha \}$. Clearly, execution of **while** B **do** β **od** starting in state I cannot "produce" any values not in $Dom_I(\alpha)$ and hence cannot change any k-ary function at points outside $(Dom_I(\alpha))^k$. Finiteness of $Dom_I(\alpha)$ can be expressed in DL_{arrays} by a formula Fin_α (Lemma 4). This allows us to split the task of expressing $Global\text{-}Loop_\alpha$ into two parts, according to whether $Dom_I(\alpha)$ is finite or infinite.

When $Dom_I(\alpha)$ is finite then $Global\text{-}Loop_\alpha$ can be expressed by a formula L_α^{fin} of DL_{arrays} which, informally speaking, says that "α can be executed in such a way that it enters the same state just before two different passes through β" (Lemma 5).

When $Dom_I(\alpha)$ is infinite we can express $Global\text{-}Loop_\alpha$ by a formula L_α^{inf} of DL_{arrays} which, informally speaking, says "For all $n \geq 0$, if there are n different elements in $Dom_I(\alpha)$ then β can be executed n times and B is true before each of these repetitions of β" (Lemma 6).

With the formulae from the three lemmas, $Global\text{-}Loop_\alpha$ can then be expressed as

$$(Fin_\alpha \wedge L_\alpha^{\mathrm{fin}}) \vee (\neg Fin_\alpha \wedge L_\alpha^{\mathrm{inf}}) .$$

We finish this proof of Theorem 2 by formulating Lemmas 4,5, and 6, and indicating their proofs.

For the proofs of Lemmas 4-6, let f be a unary function symbol, v,z,z_0,z_1, ... be variables and let p and q be Boolean variables, none of which occurs in α. Let x be a variable which does occur in α. For any state I, define $R_f(I) = \{(f^n(z_0))_I : n \geq 0\}$, where f^n is the n-fold composition of f with itself. In what follows R_f will serve as "storage device," holding finite portions of $Dom_I(\alpha)$ in a "circular chain" $z_0 \neq f(z_0) \neq f^2(z_0) \neq \ldots \neq f^m(z_0) = z_0$.

Lemma 4: For each while-program α (with array assignments) there is a DL_{arrays}-formula Fin_α such that for all states I, $I \models Fin_\alpha$ iff $Dom_I(\alpha)$ is finite.

Proof: We construct a program $FREE_\alpha$ which nondeterministically builds up R_f to comprise an arbitrary finite subset of $Dom_I(\alpha)$. $FREE_\alpha$ does this by entering into R_f the values of variables from α and also entering the values, at arguments that are already in R_f, of function symbols from α. To write out $FREE_\alpha$ in a concise way, let SET_z for any variable z be an abbreviation for

```
z:=z0; q:=true;
while q do z:=f(z); choice(q:=true, q:=false) od.
```

Thus, SET_z is a program which sets z to an arbitrary value from R_f.

For any k-ary function symbol h in α, $k \geq 0$, let $APPLY_h$ be the program

```
SETz1; ... ; SETzk;
v:=h(z1, ... ,zk); z:=f(z0);
while (z≠z0 ∧ z≠v) do z:=f(z) od;
if z≠v then f(v):=f(z0); f(z0):=v; p:=true fi.
```

Assuming that R_f is in the form of a circular chain as described above, this program $APPLY_h$ picks k random values from R_f, applies h to those values,

checks if the value thus obtained is not yet in R_f, and, if it is not, enters the value into R_f, preserving the form of a circular list, and sets a flag p to true whenever a new value is entered into R_f. For zeroary function symbols h, i.e. for simple variables, the first line of the above program APPLY_h is empty and the second line reads "v:=h".

Let APPLY_α be a program which nondeterministically chooses a function symbol from α to "apply." If $h_1, \ldots, h_r$ are all the function symbols in α, including variables, then APPLY_α is the program

$$\textbf{choice}(\mathrm{APPLY}_{h_1}, \textbf{choice}(\mathrm{APPLY}_{h_2}, \ldots, \textbf{choice}(\mathrm{APPLY}_{h_r}, \mathrm{APPLY}_{h_r}) \ldots)),$$

and FREE_α is the program

z_0:=x; $f(z_0)$:=z_0;
p:=true;
while p **do** p:=false; APPLY_α **od**.

Any execution of FREE_α in a state I has the effect of making R_f into a finite subset of $Dom_I(\alpha)$, and, conversely, for each finite subset of $Dom_I(\alpha)$ containing x_I there is some execution of FREE_α which makes R_f into that finite subset of $Dom_I(\alpha)$. Hence we can express finiteness of $Dom_I(\alpha)$ by the following formula Fin_α:

$$\langle \mathrm{FREE}_\alpha \rangle \ \mathrm{CLOSED}_\alpha$$

where CLOSED_α stands for the formula

$$[\text{p:=false; } \mathrm{APPLY}_\alpha]\ \neg p,$$

which asserts that R_f is closed under application of function symbols (including zeroary function symbols, i.e. variables) from α. This ends the proof of Lemma 4. □

Lemma 5: For each while-program α (with array assignments) of the form while B do β there is a DL_{arrays}-formula L_α^{fin} such that for all states I, if $Dom_I(\alpha)$ is finite then $I \models (Global\text{-}Loop_\alpha \equiv L_\alpha^{fin})$.

Proof of Lemma 5: Define the tree $S_\alpha(I)$ as in the proof of Lemma 3. As mentioned in the proof of Lemma 3, $I \models Global\text{-}Loop_\alpha$ is equivalent to $S_\alpha(I)$ having an infinite path, with the states $J_0, J_1, \ldots$ mentioned in the definition of $Global\text{-}Loop_\alpha$ being the labels along that infinite path. If $Dom_I(\alpha)$ is finite then there are only finitely many different states (labels of nodes) in $S_\alpha(I)$. This is so because execution of β, no matter how often repeated, cannot change the values of any k-ary function symbols, $k \geq 0$, on arguments outside $(Dom_I(\alpha))^k$ and it cannot set those values to anything outside $Dom_I(\alpha)$. Hence, under the assumption that $Dom_I(\alpha)$ is finite, $S_\alpha(I)$ has an infinite path iff there is a repetition of a state along some path. The existence of a state which is repeated along some path in $S_\alpha(I)$ can be expressed by a formula L_α^{fin} of DL_{arrays}, again under the assumption that $Dom_I(\alpha)$ is finite. We finish this proof of Lemma 5 by showing how to construct such a formula L_α^{fin}.

As in the proof of Lemma 3, let V be a vector of all the variables of α, Boolean and others, and let V′ be a vector of new symbols matching the types of symbols in V. Let $H=(h_1, \ldots, h_r)$ be a vector of all the function symbols in α of positive arity and let $H'=(h_1', \ldots, h_k')$ be a matching vector of new symbols. Let V′:=V, V′=V, H′:=H, H′=H stand for componentwise assignments and equality. (Obviously, DL_{arrays} does not provide for the assignments and tests between whole functions we are using here, and we may not use such wholesale assignments and tests in the final formula L_α^{fin}.) The assertion that a state repeats along a path in $S_\alpha(I)$ can then be expressed as

```
< choice(p:=true, p:=false);
while (p ∧ B) do β; choice(p:=true, p:=false) od;
if then BUZZ fi;
V′:=V; H′:=H;
β;
choice(p:=true, p:=false);
while (p ∧ B) do β; choice(p:=true, p:=false) od>
(V=V′ ∧ H=H′) .
```

This formula is entirely analogous to the formula (A′) in the proof of Lemma 3, except that now, due to the presence of array assignments, functions as well as simple variables are subject to change and consequently have to be remembered (H′:=H) and later compared (H′=H).

As already pointed out, DL_{arrays} does not provide any means for assigning and comparing whole functions. But note that the program α obviously cannot change the values of functions on arguments outside $Dom_I(\alpha)$. Hence it would suffice to have both the assignments and the comparisons between functions restricted to arguments from $Dom_I(\alpha)$. In addition, remember that we are assuming that $Dom_I(\alpha)$ is finite. If we assume that $R_f(I) = Dom_I(\alpha)$, then we can express equality of two k-ary function symbols h and h′ on arguments from $Dom_I(\alpha)$ by the following formula EQUAL(h,h′):

$$[SET_{z_1}; \ldots ; SET_{z_k}] \ (h(z_1, \ldots ,z_k)=h'(z_1, \ldots ,z_k)) .$$

Let $EQUAL_\alpha(H,H')$ to be the conjunction of the formulae $EQUAL(h_i,h_i')$, $1 \le i \le r$.

Assuming similarly that $R_f(I)=Dom_I(\alpha)$ and $Dom_I(\alpha)$ is finite, it is straightforward to write a program, ASSIGN(h,h′), which uses k nested loops to run k variables through all possible k-tuples from $(Dom_I(\alpha))^k$ and assigns the value of h to h′ on all those arguments. Let $ASSIGN_\alpha(H',H)$ be the program $ASSIGN(h_1',h_1); \ldots ;ASSIGN(h_r',h_r)$.

The desired formula L_α^{fin} can now be written as

$\langle \text{FREE}_\alpha \rangle$ (CLOSED$_\alpha$ ∧
<choice(p:=true, p:=false);
while (p ∧ B) do β; choice(p:=true, p:=false)od;
if ¬B then BUZZ fi;
V′:=V; ASSIGN$_\alpha$(H′,H);
β;
choice(p:=true, p:=false);
while (p ∧ B) do β; choice(p:=true, p:=false)od>
V=V′ ∧ EQUAL$_\alpha$(H,H′))).

Note that the initial portion "$\langle \text{FREE}_\alpha \rangle$ (CLOSED$_\alpha$ ∧ ..." in the above formula allows us to assume that $R_f(I)=Dom_I(\alpha)$ and $Dom_{(\alpha)}$ is finite when ASSIGN$_\alpha$ is executed and EQUAL$_\alpha$ is evaluated. □

Lemma 6: For each while-program α (with array assignments) of the form while B do β there is a DL_{arrays}-formula L_α^{inf} such that for all states I, if $Dom_I(\alpha)$ is infinite then

$$I \models (\textit{Global-Loop}_\alpha \equiv L_\alpha^{\text{inf}}).$$

Proof of Lemma 6: Let L_α^{inf} be the formula

$$[\text{FREE}_\alpha] \langle z_1:=f(z_0);\ \text{while } (z_0 \neq z_1) \wedge B \text{ do } (\beta;\ z_1:=f(z_1)\ \text{od} \rangle\ z_1=z_0$$

with FREE$_\alpha$ as in the proof of Lemma 4.

To understand why the above formula L_α^{inf} has the desired property, note that it can be paraphrased as follows: No matter what finite subset of $Dom_I(\alpha)$ one puts into the circular chain R_f ("[FREE$_\alpha$]"), it is always possible to execute at least as many passes through β as there are elements in $R_f(I)$ (to be precise, minus one). The variable z_1 is used to count the passes through β by moving along the chain of R_f. This finishes the proof of Lemma 6 and Theorem 2. □

The proofs of Theorems 1 and 2 do not depend on what tests are allowed in while-programs. Hence they carry over without change to the version of regular DL defined in [Pratt 1976, Harel 1979] where tests can be any formulae of First Order Predicate Calculus, as well as to the version of DL called "rich-test" DL in [Harel 1979] where the syntax of tests is defined inductively to allow any DL-formulae as tests.

4. Deterministic versus nondeterministic DL

In this Section we show that in the absence of quantifiers the expressive power of DL decreases if we restrict ourselves to deterministic while-programs. A while-program is *deterministic* if it does not contain any choice-statements. A DL-formula is *deterministic* if all the programs it contains are deterministic.

Theorem 3: There is a quantifier-free DL-formula for which there is no equivalent quantifier-free deterministic DL-formula.

Proof: A *partial state* specifies a domain D and an arity-respecting assignment of *partial* functions and predicates on D to function and predicate symbols. We now let I,J, ... denote partial states as well as states and let f_I denote the function or predicate assigned by I to the symbol f.

A partial function or predicate f_1 is an *extension* of f_2 iff f_1 restricted to the domain of f_2 is equal to f_2. The extension is *finite* iff domain (f_1) - domain (f_2) is finite. We say that a partial state J is a (finite) extension of a partial state I iff f_J is a (finite) extension of f_I for all symbols f.

If I is a partial state and F is a DL-formula, we say that I *satisfies* F, in symbols $I \models F$, if $J \models F$ for all states J which extend I. Finally, we say that I *determines* iff $I \models F$ or $I \models \neg F$.

The main part of the proof rests on the following lemma.

Lemma 7: Let F be a quantifier-free deterministic DL-formula. For any partial state there is a finite extension which determines F.

Assuming Lemma 7 for a moment, we can complete the proof of Theorem 3 by considering the following formula F:

<p:=true; while p do x:=f(x); choice(p:=true,p:=false) od>
[p:=true; while p do x:=f(x); choice(p:=true,p:=false) od] q(x) .

Let I be a partial state whose domain is the integers and such that f_I is the successor function, x_I=0, and the domain of q_I is empty. For states extending I, the formula F is equivalent to the assertion $(\exists n>0)(\forall m>n)q(m)$, whose truth is obviously not determined by any partial interpretation of q with finite domain. Hence no finite extension of I determines F, and therefore F cannot be equivalent to any quantifier free deterministic DL-formula.

Proof of Lemma 7: The proof is by induction on the structure of F. The cases in which F is atomic, a conjunction, or a negated formula follow easily. Suppose F is of the form $\langle\alpha\rangle$G, and let I be a partial state. There are two possibilities.

First, suppose that there is no extension of I on which α halts, i.e., $\alpha(J) = \varnothing$ for all states J which extend I. Then I already determines the truth value of $\langle\alpha\rangle$G, viz., false.

So we may assume that there is some state I_0 extending I such that α halts started in state I_0. More precisely, because α is deterministic we may assume that $T_\alpha(I_0)$ consists of a single path from the root to a halt node. Now, because the path is finite and the tests are quantifier-free, the outcomes of the tests on this path depend on the values in I_0 of only a finite number of terms. Thus there is a finite extension I_1 of I which ensures that α will behave as though it were started in I_0. That is, let α_0 be the program consisting of the

successive assignment statements on the path in $T_\alpha(I_0)$. Then for any state J extending I_1, we have $\alpha(J) = \alpha_0(J)$.

It follows that there is a partial state, which for obvious reasons we call $\alpha_0(I_1)$, such that if a state J extends I, then $\alpha_0(J) = \{J'\}$ for some J' extending $\alpha_0(I_1)$, and conversely if J' is a state extending $\alpha_0(I_1)$, then $\{J'\} = \alpha_0(J)$ for some state J extending I_1. Moreover, f_{I_1} differs from $f_{\alpha_0(I_1)}$ at only finitely many arguments, for all symbols f.

Now by induction, there is a finite extension I_2 of $\alpha_0(I_1)$ which determines G. Let I_3 be the partial state such that f_{I_3} restricted to the domain of f_{I_1} is equal to f_{I_1}, and f_{I_3} restricted to domain (f_{I_3}) - domain (f_{I_1}) is equal to f_{I_2} for all symbols f. Then I_3 is a finite extension of I_1 which determines $\langle\alpha\rangle G$, as the reader may verify. □

It is worth pointing out that Lemma 7 does not depend on our restriction to while-programs. It depends only on two facts about the programs under consideration: first, the truth value of any test can be determined by a finite extension of any given partial state, and, second, in any terminating computation of a program only finitely many evaluations of tests are performed. Thus Lemma 7 and Theorem 3 would remain true if we considered programs α described by trees T_α which are not effectively generable; it would also remain true if we allowed tests to be quantifier-free deterministic DL-formulas themselves; and it would remain true if we allowed array assignments or, for that matter, even assignments of whole functions to function symbols by a single statement. Naturally, it would break down if we introduced assignments of random values to variables (cf. Section 5) since such assignments can obviously simulate nondeterminism.

The proof of Theorem 3 works equally well for Propositional Dynamic Logic ("PDL", cf. [Pratt 1976, Harel 1978]). We only need to choose F to be the formula $\langle a^* \rangle [a^*] q$ and interpret the symbols a and q to mean "x:=f(x)" and "q(x)".

Theorem 4: There is a PDL-formula for which there is no equivalent deterministic PDL-formula.

We conjecture that even in the presence of quantifiers, nondeterminism adds to the expressive power of deterministic DL, although we expect the proof to be more complicated in this case.

5. Random assignments versus quantifiers

We let x:=? denote assignment of an arbitrary "random" value to the variable x. Formally, $m(x:=?) = \{(I,J) : s_I = s_J \text{ for all symbols } s \neq x\}$. It is easy to see that assignments of random values from the domain make quantifiers superfluous in DL: instead of $\exists x$ we can use $\langle x:=? \rangle$, cf. [Harel 1979]. Similarly, random assignments can replace choice-statements. Random assignments can however be used in more powerful ways:

Theorem 5: The predicate "(x,y) is in the reflexive transitive closure of the binary relation R" cannot be expressed in DL without random assignments. It can be expressed in DL with random assignments.

The choice of this kind of predicate is due to V. R. Pratt. A simplification of an earlier proof of the second author is due to M. Paterson.

Proof: With random assignments the predicate "(x,y) is in the reflexive transitive closure of the binary relation R" can be expressed as

$$\langle z:=?;\ \textbf{while}\ x \neq y \wedge R(x,z)\ \textbf{do}\ x:=z;\ z:=?\ \textbf{od} \rangle\ x=y.$$

To show that the same predicate cannot be expressed without random assignments, we first prove two lemmas.

Define two programs α and β to be *equivalent* if $m(\alpha) = m(\beta)$. Call any while-program which does not contain any function symbols of positive arity a

shuffle program. (Such programs can only "shuffle" the values of variables but cannot "create" any new values during execution.) A while-program is *while-free* if it does not contain any while-statement.

Lemma 8: Any shuffle program α is equivalent to some program of the form

if C then BUZZ fi; γ

where γ is a while-free shuffle program.

Proof: Let $x_1, \ldots, x_n$ be the variables of α. In the absence of function symbols, the program α can only "shuffle" the values of $x_1, \ldots, x_n$ around but it cannot "create" any new values. Formally this means that for all states I and J, where J is the label of some node in $T_\alpha(I)$, we have $\{(x_1)_I, \ldots, (x_n)_I\} \supseteq \{(x_1)_J, \ldots, (x_n)_J\}$.[4] This can be proven easily by induction on the trees $T_\alpha(I)$. For any shuffling s of values among $x_1, \ldots, x_n$ (which could be formalized as a mapping from $\{1, \ldots, n\}$ to itself), whether $T_\alpha(I)$ contains path from the root to a halt nodes whose execution results in the shuffling s is determined by the values of all predicate symbols from α on arguments from $\{(x_1)_I, \ldots, (x_n)_I\}$, simply because the outcome of all the tests in T_α is determined by these values. Hence there is a test C_s which is true in state I iff $T_\alpha(I)$ contains a path which accomplishes the shuffling s. Using these formulae C_s, it is straightforward to write the desired while-free program γ. Similarly, there is a test C, namely the conjunction over all s of $\neg C_s$, which is true in state I iff $\alpha(I) = \varnothing$ □

Now observe that for a program α of the form described in the Lemma 8, the formula $\langle\alpha\rangle P$ is equivalent to $\neg C \wedge \langle\gamma\rangle P$. By techniques from [Pratt 1976], if γ is a while-free program then there is a first order formula which is equivalent to $\langle\gamma\rangle P$. This implies the following

Lemma 9: Let R be a DL-formula which does not contain any function symbols of positive arity. Then there is a program-free DL-formula, i.e. a first order formula, which is equivalent to R.

To finish the proof of Theorem 5, assume that some DL-formula Q expresses the predicate x,y) is in the reflexive transitive closure of R." Consider states which have the integers as domain, interpret the predicate R(u,v) as the successor predicate u+1=v, interpret every predicate symbol except R and = as being true on all arguments, and interpret all function symbols of positive arity as identity functions on the first argument. Note that in such states, the predicate "(x,y) is the the reflexive transitive closure of R" is equivalent to the predicate $x \leq y$. Consider the formula Q′ obtained from Q by (recursively) replacing each term of the form $f(t_1, \ldots, t_k)$, $k>0$, with its first argument, t_1, and by replacing with true each atomic formula containing a predicate symbol other than R. In states of the kind we are considering, this formula Q′ is equivalent to Q. By Lemma 9, there is an equivalent first order formula Q′′. If we replace in this formula Q′′ each term of the form R(u,v) by u=v+1, we obtain a formula of the First Order Theory of Successor which expresses the predicate $x \leq y$. But it is well-known that this predicate cannot be expressed in the First Order Theory of Successor. □

Theorem 5 holds for a much wider class of programs than just the while-programs we are considering. First, we can generalize the structure of the trees T_α by allowing any trees whose edges carry labels from a finite set of assignment statements and tests. This generalization covers "context-free" and "r.e." DL (cf. [Harel 1979]) as well as versions of DL where the trees T_α are not effectively generable. Second, we can expand the set T of tests allowed in the programs. For example, Lemma 9 and consequently Theorem 5 hold for "rich-test" DL, where tests are inductively defined to be DL-formulae themselves, cf. [Harel 1979].

6. Other Results and Open Problems

The results reported above are part of a study of the comparative expressive power of different versions of DL. Related results which will be reported in subsequent papers reveal, for example, that DL with array assignments and DL

with random assignments are incomparable in expressive power, and consequently, both are strictly less expressive than DL with both array and random assignments which is equivalent to one formulation of Weak Second-Order Predicate Calculus. If we permit a very generous notion of program, namely r.e. programs with infinitely many "rich" tests (cf. [Harel 1979]), then the corresponding version of DL is equivalent to the constructive segment of the infinitary language $L_{\omega_1,\omega}$. This latter result was obtained by the first author in collaboration with Rohit Parikh.

The major open problem concerning the topics discussed in this paper is the expressive power of nondeterminism in the presence of quantifiers (cf. Section 4). As mentioned, we conjecture that nondeterminism adds to the expressive power of DL even in the presence of quantifiers.

Another open question similar to the problem of expressing looping is whether the "intersection" operator, of Algorithmic Logic [Rasiowa 1977, Salwicki 1970] can be expressed in DL. The meaning of the Algorithmic Logic formula $\cap\alpha P$ was originally defined only for deterministic programs α, and there are at least two different sensible ways to extend its meaning to nondeterministic programs. We can define $\cap\alpha P$ to be equivalent to $\forall n \geq 0\ \langle\alpha^n\rangle P$, where α^n stands for the n-fold repetition of α. It is still an open question whether or not $\forall n \geq 0\ \langle\alpha^n\rangle P$ can be expressed in DL, with or without array assignments. A different and perhaps more natural extension of the meaning of $\cap\alpha P$ to nondeterministic programs α is achieved by defining $\cap\alpha P$ to be equivalent to *Global-Loop*$_\beta$ with β = **while P do** α **od**. (See Section 2.2 for a definition of *Global-Loop*.) Again we do not know if this predicate is expressible in DL, with or without array assignments.

Acknowledgments. We wish to thank Michael J. Fischer for raising the question about the expressibility of looping in DL, Michael Paterson for simplifying an earlier proof of Theorem 5, and Karel Culik, Joe Halpern, David Harel, Rohit Parikh, and Vaughan R. Pratt for helpful comments.

7. References

E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976, 217 pp.

S. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification.* Lecture Notes in Computer Science, vol. 36, Springer-Verlag: Berlin-Heidelberg-New York 1975.

I. Greif and A. R. Meyer. Specifying the semantics of while-programs. Tech. Memo. 130, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1979.

D. Harel. *First Order Dynamic Logic.* Lecture Notes in Computer Science, vol.68, Springer-Verlag: Berlin-Heidelberg-New York 1979, 133 pp. (Revised version of : D. Harel. Logics of Programs: Axiomatics and Descriptive Power. PhD Thesis, Dept. of Electrical Engineering and and Computer Science, Massachusetts Institute of Technology, May 1978, 152 pp.)

D. Harel and V. R. Pratt. Nondeterminism in Logics of Programs. *5th Annual Symposium on Principles of Programming Languages,* January 1978, 203-213.

C. A. R. Hoare. Some properties of predicate transformers, *Journal of the ACM, 25,* 3 (1978), 461-480.

R. Hossley and C. Rackoff. The emptiness problem for automata on infinite trees. *13th Annual Symposium on Switching and Automata Theory,* October 1972.

V. R. Pratt. Semantical Considerations on Floyd-Hoare Logic. *17th Annual Symposium on Principles of Programming Languages,* January 1976, 109-121.

M. Rabin. Automata on infinite trees and the synthesis problem. Technical Report No. 37, Hebrew University, Jerusalem, January 1970.

H. Rasiowa. Algorithmic Logic. Report No. 281, Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland, 1977, 206 pp.

A. Salwicki. Formalized algorithmic languages. *Bull. de l'Acad. Polonaise des Sciences, Serie des Science math., astr., et phys. - vol. XVIII,* 5, 1970, 227-232.

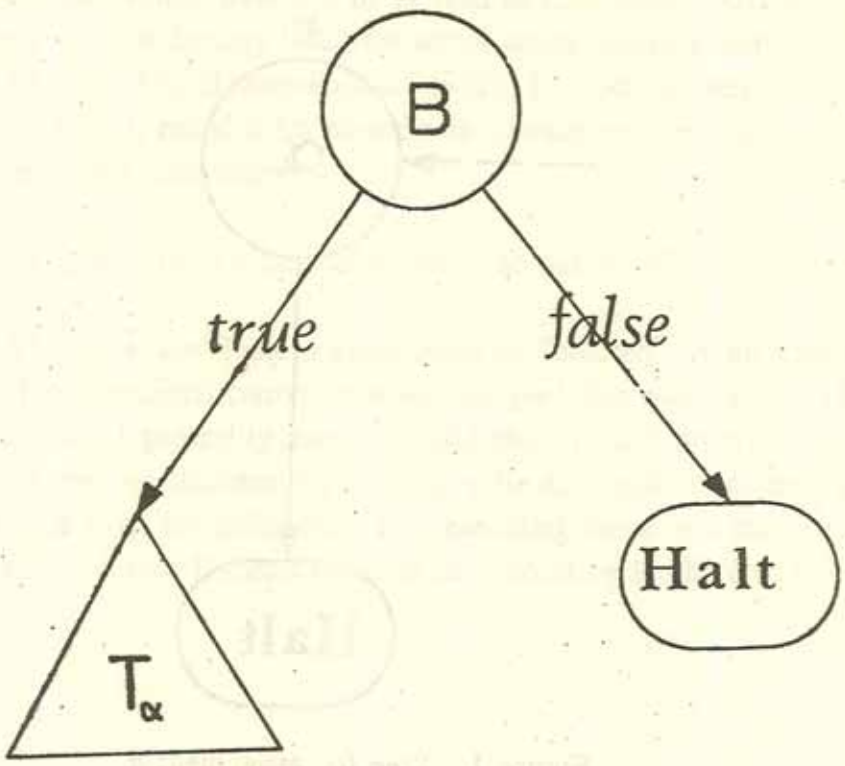

Figure 3. Tree for if B then α fi.

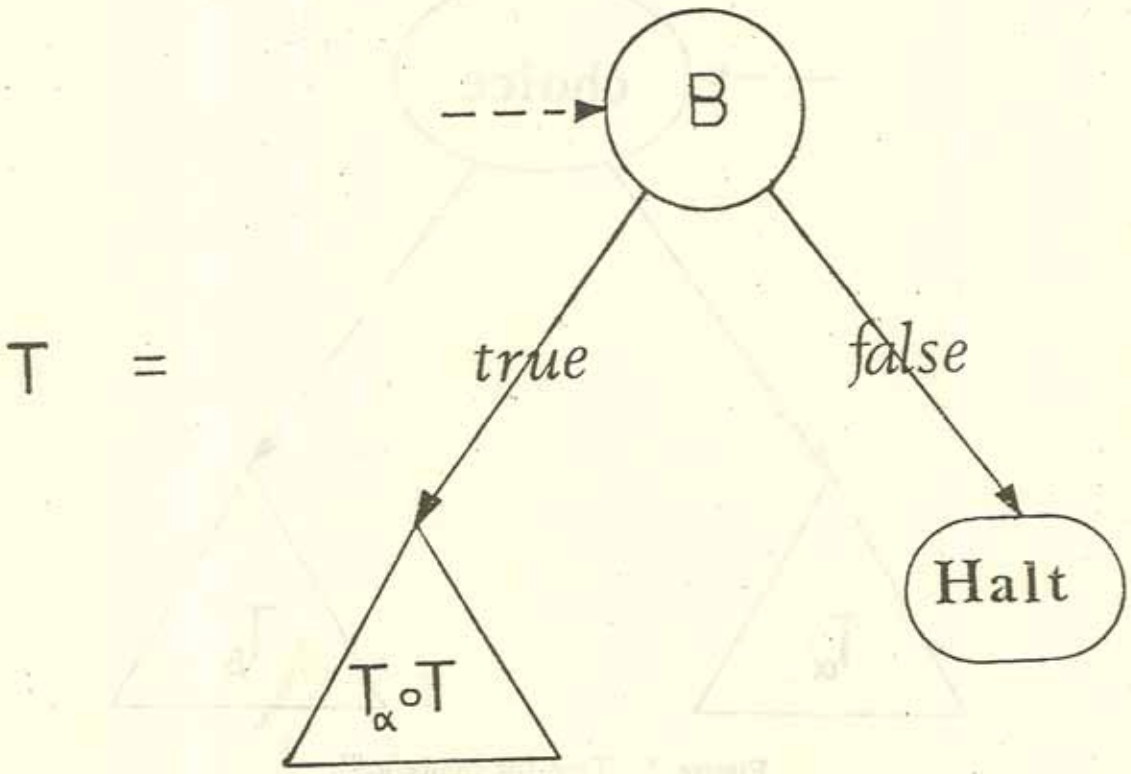

Figure 4. Equation defining the tree for while B do α od.