Expressivity of extensions of dynamic first-order logic

Balder ten Cate[1], Jan van Eijck[2] and Juan Heguiabehere[3]

Abstract

Dynamic predicate logic (DPL), presented in [5] as a formalism for representing anaphoric linking in natural language, can be viewed as a fragment of a well known formalism for reasoning about imperative programming [6]. An interesting difference from other forms of dynamic logic is that the distinction between formulas and programs gets dropped: DPL formulas can be viewed as programs. In this paper we show that DPL is in fact the basis of a hierarchy of formulas-as-programs languages.

1 Formulas-as-Programs in DPL and its Extensions

In this paper we investigate the landscape of extensions of DPL with the six operations $\cup$, $\breve{}$, σ, $\breve{\sigma}$, $\cap$, $\exists$. All 64 combinations are classified with respect to their expressive power. Extensions of DPL with $\cap$ (relation intersection) and $\exists$ (local variable declaration) are studied in [8], while in [1], an extension of DPL with $\cup$ (relation union) and σ (simultaneous substitution) is axiomatized, and ω-completeness is proved for the extension of DPL with $\cup$, σ and $*$ (Kleene star). In a dynamic setting, left-to-right substitutions σ have right-to-left counterparts $\breve{\sigma}$ (converse substitutions). For pre- and postcondition reasoning with extensions of DPL, converse substitution and relation converse $\breve{}$ are attractive.

Some of these operators have been studied from a linguistic perspective. In particular, [4] discusses potential linguistic applications of the $\cup$ and $\cap$ operator. The relevance of $\cap$ operator is mentioned there for the analysis of *plurals*, and it is argued that $\cup$ is needed for a proper analysis of sequences such as: *"A professor or an assistant professor will attend the meeting of the university board. She will report to the faculty."*

Unlike [5], we allow function symbols, so DPL terms are given by $t ::= x \mid c \mid f(t_1, \ldots, t_n)$. DPL formulas are given by $\phi ::= \exists x \mid Rt_1 \ldots t_n \mid t_1 \approx t_2 \mid \neg\phi \mid \phi_1; \phi_2$. Terms are interpreted as usual. We use $t^{\mathcal{M},g}$ for the interpretation of term t in model $\mathcal{M}$ under valuation g.

A substitution is a finite set of bindings $x := t$, with the usual conditions that no binding is trivial (of the form $x := x$) and that every x in the set has at most one binding (substitutions are functional). Examples are $\{x := f(x)\}$ ("set new x equal to f-value of old x"), $\{x := y, y := x\}$ ("swap values of x and y"). If a substitution contains just a single binding we omit the curly brackets and write just the *assignment statement* $x := t$. A converse substitution is a finite set of converse bindings $(x := t)\breve{}$, with the same conditions as those for substitutions. An example is $(x := f(x))\breve{}$ ("set old x equal to f-value of new x", or "look at all inputs g that differ from the output h only in x, and that satisfy $f(g(x)) = h(x)$").

A variable state in a model $\mathcal{M} = \langle D, I\rangle$ is a member of D^{Var}, where Var is the set of variables of the language. Letting g, h, k range over variable states, the interpretation of DPL and the extensions that we study, in a model $\mathcal{M} = \langle D, I\rangle$, is given by the following definition:

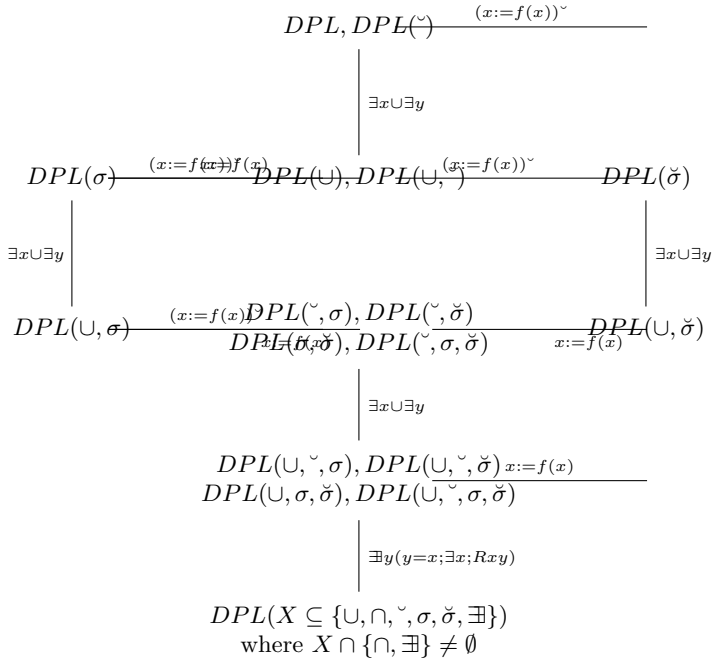
$$
\begin{array}{rll}
g[\![\exists x]\!]^{\mathcal{M}} h & \text{iff} & h = g^x_d \text{ for some } d \in D \\
g[\![Rt_1 \ldots t_n]\!]^{\mathcal{M}} h & \text{iff} & h = g \text{ and } \langle t_1^{\mathcal{M},g}, \ldots, t_n^{\mathcal{M},g}\rangle \in R^{\mathcal{M}} \\
g[\![t_1 \approx t_2]\!]^{\mathcal{M}} h & \text{iff} & h = g \text{ and } t_1^{\mathcal{M},g} = t_2^{\mathcal{M},g} \\
g[\![\neg\phi]\!]^{\mathcal{M}} h & \text{iff} & h = g \text{ and there is no } k \text{ such that } g[\![\phi]\!]^{\mathcal{M}} k \\
g[\![\phi;\psi]\!]^{\mathcal{M}} h & \text{iff} & g[\![\phi]\!]^{\mathcal{M}} k \text{ and } k[\![\psi]\!]^{\mathcal{M}} h \text{ for some } k \in D^{\mathrm{Var}} \\
g[\![\sigma]\!]^{\mathcal{M}} h & \text{iff} & h = g^{x_1 \cdots x_n}_{d_1 \cdots d_n} \text{ where } \{x_1, \ldots, x_n\} = \mathrm{dom}(\sigma) \text{ and } d_i = \sigma(x_i)^{\mathcal{M},g} \\
g[\![\breve{\sigma}]\!]^{\mathcal{M}} h & \text{iff} & g = h^{x_1 \cdots x_n}_{d_1 \cdots d_n} \text{ where } \{x_1, \ldots, x_n\} = \mathrm{dom}(\sigma) \text{ and } d_i = \sigma(x_i)^{\mathcal{M},h} \\
g[\![\exists x(\phi)]\!]^{\mathcal{M}} h & \text{iff} & (g^x_d)[\![\phi]\!]^{\mathcal{M}} k \text{ and } h = k^x_{g(x)} \text{ for some } d \in D, k \in D^{\mathrm{Var}} \\
g[\![\phi \cup \psi]\!]^{\mathcal{M}} h & \text{iff} & g[\![\phi]\!]^{\mathcal{M}} h \text{ or } g[\![\psi]\!]^{\mathcal{M}} h \\
g[\![\phi \cap \psi]\!]^{\mathcal{M}} h & \text{iff} & g[\![\phi]\!]^{\mathcal{M}} h \text{ and } g[\![\psi]\!]^{\mathcal{M}} h \\
g[\![\phi\breve{}]\!]^{\mathcal{M}} h & \text{iff} & h[\![\phi]\!]^{\mathcal{M}} g.
\end{array}
$$

[1]ILLC, Amsterdam
[2]CWI and ILLC, Amsterdam, Uil-OTS, Utrecht
[3]ILLC, Amsterdam

With DPL we will denote the basic language. Extensions will be indicated with DPL(X), where X is a set of operators. For instance, DPL($\sigma, \breve{}$) denotes the extension of DPL with simultaneous substitutions and converse.

2 The Lattices of DPL and DPL(*) Extensions

The following figure represents the lattice of all possible combinations of DPL with operators from $\{\cup, \cap, \breve{}, \sigma, \breve{\sigma}, \exists\!\!\exists\}$ (union, intersection, converse, simultaneous substitution, converse substitution, hiding). It indicates which operators can be defined in terms of which; the labels on the arrows indicate counterexamples to equal expressivity, i.e., formulas from the lower language that don't have a counterpart in the upper language.



Note that all 64 combinations of the six operators are present in the diagram. The diagram makes immediately clear which extensions of DPL are closed under converse: precisely those which are in the same node of the lattice as the corresponding version of DPL *with* converse operator. Adding Kleene star gives an isomorphic lattice for DPL(*) and its extensions: none of the distinctions collapse because the same counter-examples to equal expressivity still work. The justification of the lattices is in Section 5.

3 Expressivity and Programming Constructs

Most of the operators we consider for extending DPL have obvious uses in programming [2, 6]. Kleene star allows the implementation of *while* loops.

$$\exists s; \exists i; s \approx 0; i \approx -1; (i \not\approx n; i := i + 1; s := s + b(i))^*; i \approx n \tag{1}$$

Program (1) computes the sum of the $n + 1$ elements of an array b and places the result in the variable s. If the value for n is not known beforehand (at 'compile time'), it is not possible to write an equivalent program without the use of *.

$$(x \approx 1;\ \exists y; y \approx t_1) \cup (x \approx 0; \exists z; z \approx t_2) \tag{2}$$

Nondeterminism ($\cup$) allows conditional execution: program (2) changes the value of either y or z, depending on the value of x. This cannot be done without $\cup$ (or stronger operators like $\cap$ or $\exists\!\!\exists$).

The $\exists\!\!\exists$ operator allows for the declaration of local variables. Simultaneous substitution permits performing certain computations without the use of auxiliary variables, and finally converse and converse simultaneous substitution are useful for pre- and postcondition reasoning, as they allow us to define the inverses of programs under certain conditions (Chapter 21 in [3]).

4 Left-to-Right and Right-to-Left Substitution

Because the semantics of DPL formulas is completely symmetric, performing a substitution in a DPL formula can be done in two directions: left-to-right and right-to-left [8] (see also [7], where substitutions for DPL with a stack semantics are studied). Left-to-right substitutions affect the left-free variable occurrences, right-to-left substitutions the right-free (or 'actively bound') variable occurrences.

The substitution lemma for first order logic has the form $\mathcal{M} \models_g \sigma(\phi)$ iff $\mathcal{M} \models_{g_\sigma} \phi$, given an obvious safety condition on σ, to the effect that no variables in the range of σ should get accidentally captured by quantifiers in ϕ.

One would like to prove a left-to-right substitution lemma for DPL to the effect that $g[\![\sigma(\phi)]\!]h$ iff $g_\sigma[\![\phi]\!]h$ (suppressing the parameter $\mathcal{M}$ for readability). Viewing the substitution itself as a state change, we can decompose this into $g[\![\sigma]\!]g'[\![\phi]\!]h$. This uses $g[\![\sigma]\!]k$ iff $k = g_\sigma$. Since, in general, σ is not expressible in DPL, we need to extend the language with substitutions [1].

The right-to-left substitution lemma for DPL that one would like to prove says that $g[\![\breve{\sigma}(\phi)]\!]h$ iff $g[\![\phi]\!]h_\sigma$. Viewing the right-to-left substitution itself as a state change, we can decompose this into $g[\![\phi]\!]h'[\![\breve{\sigma}]\!]h$. This uses $k[\![\breve{\sigma}]\!]h$ iff $k = h_\sigma$. Again, since in general $\breve{\sigma}$ is not expressible in DPL, we have a motivation to extend the language with converse substitutions.

Use $\circ$ for relational composition of substitution expressions, defined in the standard way. Note that $\sigma;\rho$ is equivalent to $\sigma(\rho)$, which is in turn equivalent to $\rho \circ \sigma$. E.g., $x := x+1; y := x$ is equivalent to $\{x := x+1, y := x+1\}$.

Every DPL($\cup, \sigma$) formula can be written with ; associating to the right, as a list of predicates, quantifications, negations, choices and substitutions, with a substitution ρ at the end (possibly the empty substitution). Left-to-right substitution in DPL($\cup, \sigma$) is defined by:

$$\begin{array}{rcl}
\sigma(\rho) & := & \rho \circ \sigma \\
\sigma(\rho;\phi) & := & \rho \circ \sigma;\phi \\
\sigma(\exists v;\phi) & := & \exists v;\sigma'\phi \text{ where } \sigma' = \sigma\backslash\{v := t \mid t \in T\} \\
\sigma(P\bar{t};\phi) & := & P\sigma\bar{t};\sigma\phi \\
\sigma(t_1 \approx t_2;\phi) & := & \sigma t_1 \approx \sigma t_2;\sigma\phi \\
\sigma(\neg(\phi_1);\phi_2) & := & \neg(\sigma\phi_1);\sigma\phi_2 \\
\sigma((\phi_1 \cup \phi_2);\phi_3) & := & \sigma(\phi_1;\phi_3) \cup \sigma(\phi_2;\phi_3)
\end{array}$$

A term t is left-to-right free for v in ϕ if all variables in t are input-constrained in all positions of the left-free occurrences of v in ϕ. A substitution σ is safe for ϕ if all bindings $v := t$ of σ are such that t is left-to-right free for v in ϕ. This allows us to prove:

Lemma 1 (Left-to-Right Substitution) *If σ is safe for ϕ then $g[\![\sigma(\phi)]\!]h$ iff $g_\sigma[\![\phi]\!]h$.*

Right-to-left substitution is defined in a symmetric fashion, now reading the formulas in a left-associative manner, with a converse substitution at the front, and overloading the notation by also using $\circ$ for the relational composition of converse substitutions (defined as one would expect, to get $\breve{\sigma} \circ \breve{\rho} = (\rho \circ \sigma)^\smile$):

$$\begin{array}{rcl}
\breve{\sigma}(\breve{\rho}) & := & \breve{\sigma} \circ \breve{\rho} \\
\breve{\sigma}(\phi;\breve{\rho}) & := & \phi;\breve{\sigma} \circ \breve{\rho} \\
\breve{\sigma}(\phi;\exists v) & := & \breve{\sigma}'\phi;\exists v \text{ where } \breve{\sigma}' = \breve{\sigma}\backslash\{(v := t)^\smile \mid t \in T\} \\
\breve{\sigma}(\phi;P\bar{t}) & := & \breve{\sigma}\phi;P\sigma\bar{t} \\
\breve{\sigma}(\phi;t_1 \approx t_2) & := & \breve{\sigma}\phi;\sigma t_1 \approx \sigma t_2 \\
\breve{\sigma}(\phi_1;\neg(\phi_2)) & := & \breve{\sigma}\phi_1;\neg(\breve{\sigma}\phi_2); \\
\breve{\sigma}(\phi_1;(\phi_2 \cup \phi_3)) & := & \breve{\sigma}(\phi_1;\phi_2) \cup \breve{\sigma}(\phi_1;\phi_3)
\end{array}$$

A term t is right-to-left free for v in ϕ if all variables in t are output-constrained in all positions of the right-free (actively bound) occurrences of v in ϕ. A converse substitution $\breve{\sigma}$ is safe for ϕ if all converse bindings $(v := t)^\smile$ of $\breve{\sigma}$ are such that t is right-to-left free for v in ϕ. This allows us to prove:

Lemma 2 (Right-to-Left Substitution) *If $\breve{\sigma}$ is safe for ϕ then $g[\![\breve{\sigma}(\phi)]\!]h$ iff $g[\![\phi]\!]h_\sigma$.*

5 The Justification of the DPL Extension Lattices

This section contains the proofs of the theorems that justify the lattices of DPL and DPL(*) extensions. Many of the proofs are generalizations of, or adaptations of, proofs given in [8].

Theorem 1 *$DPL(\exists)$ is equally expressive as $DPL(\cup, \cap, ^\smile, \sigma, \breve{\sigma}, \exists)$.*
Proof: *Let a formula ϕ be given, and let V be the set of variables occurring in ϕ. Furthermore, let V' and V'' be sets of variables, such that V, V' and V'' are mutually disjoint and of equal cardinality. Let $V = \{x_1, \ldots, x_n\}, V' = \{x'_1, \ldots, x'_n\}$, and $V'' = \{x''_1, \ldots x''_n\}$. The following function C translates a formula from $DPL(\cup, \cap, ^\smile, \sigma, \breve{\sigma}, \exists)$ into a test from DPL.*

$$
\begin{array}{lcl}
C(\exists y) & = & \bigwedge_{x \in V \setminus \{y\}} x' \approx x \\
C(Rt_1 \ldots t_n) & = & \bigwedge_{x \in V} x' \approx x; Rt_1 \ldots t_n \\
C(t_1 \approx t_2) & = & \bigwedge_{x \in V} x' \approx x; t_1 \approx t_2 \\
C(\neg\phi) & = & \bigwedge_{x \in V} x' \approx x; \neg(\exists x'_1; \ldots; \exists x'_n; C(\phi)) \\
C(\phi; \psi) & = & \neg\neg(\exists x''_1; \ldots; \exists x''_n; C(\phi)^{[x'_1/x''_1, \ldots, x'_n/x''_n]}; C(\psi)^{[x_1/x''_1, \ldots, x_n/x''_n]}) \\
C(\phi \cap \psi) & = & C(\phi); C(\psi) \\
C(\phi \cup \psi) & = & \neg(\neg C(\phi); \neg C(\psi)) \\
C(\phi^\smile) & = & C(\phi)^{[x_1/x'_1, \ldots, x_n/x'_n, x'_1/x_1, \ldots, x'_n/x_n]} \\
C(\sigma) & = & \bigwedge_{x \in dom(\sigma)} x' \approx \sigma(x); \bigwedge_{x \in V \setminus dom(\sigma)} x' \approx x \\
C(\breve{\sigma}) & = & \bigwedge_{x \in dom(\sigma)} x \approx \sigma(x)^{[x_1/x'_1, \ldots, x_n/x'_n]}; \bigwedge_{x \in V \setminus dom(\sigma)} x' \approx x \\
C(\exists x.\phi) & = & \neg\neg(\exists x; \exists x'; C(\phi)); x' \approx x
\end{array}
$$

Here, $\bigwedge$ is used as a shorthand for a long composition, which is non-ambiguous because the order of the particular sentences involved doesn't matter. By induction, it can be shown that every ϕ containing only variables in V, is equivalent to $\exists x'_1 \ldots x'_n(C(\phi); x_1 := x'_1; \ldots x_n := x'_n)$. □

Theorem 2 *$DPL(^*, \exists)$ is equally expressive as $DPL(^*, \cup, \cap, ^\smile, \sigma, \breve{\sigma}, \exists)$*
Proof: *As the proof of Theorem 1, now adding the following clause to the definition of C.*

$$C(\phi^*) \quad = \quad \neg\neg(\exists x''_1; \ldots; \exists x''_n; (C(\phi)^{[x'_i/x''_i]}; \textstyle\bigwedge_{x \in V} x := x'')^*; \bigwedge_{x \in V} x \approx x')$$

□

It follows immediately that every formula $\phi \in DPL(\cup, \cap, ^\smile, \sigma, \breve{\sigma}, \exists)$ is equivalent to a first order logic formula, in the sense that ϕ can be executed in $\mathcal{M}$ with input assignment g iff the first order translation of ϕ is true in $\mathcal{M}$ under g.

Theorem 3 (Visser) *$DPL(\exists)$ can be embedded into $DPL(\cap)$.*
Proof: *Let ϕ be of the form $\exists x(\psi)$, and let $\{y_1, \ldots, y_n\} = B(\phi) \setminus \{x\}$, where $B(\phi)$ are the blocking variables of ϕ, i.e., the variables y such that ϕ contains an $\exists y$ not in the scope of a negation [8]. Then ϕ is equivalent to $(\exists x; \psi; \exists x) \cap (\exists y_1; \ldots; \exists y_n)$* □

In a similar way, the following can be proved:

Theorem 4 *$DPL(^*, \exists)$ can be embedded into $DPL(^*, \cap)$.*

It is also easy to show that * gets us beyond first order expressivity:

Theorem 5 *The formula*

$$\neg(\exists y; y \approx 0; (\exists z; z \approx f(x); \exists y; z \approx f(y))^*; x \approx y)$$

cannot be expressed in $DPL(\cup, \cap, ^\smile, \sigma, \breve{\sigma}, \exists)$.
Proof: *On the natural numbers (interpreting f as the successor relation), this formula defines the odd numbers. Oddness on the natural numbers cannot be captured in a first order formula with only successor.* □

Definition 1 *A substitution $\{x_1 := t_1, \ldots, x_n := t_n\}$ is* full *if every x_i occurs in some t_j and every t_i contains some x_j.*

Examples of full substitutions are $x := f(x)$ and $\{x := y, y := x\}$, while the substitution $x := y$ is not full. It is easy to see that full substitutions are closed under composition. Note that a substitution without function symbols is full iff it is a renaming. Also, note that any formula of DPL(σ) or any of its extensions can be transformed into a formula in the same language containing only full substitutions, by replacing bindings of the form $x := t$, where t does not contain variables, by $\exists x; x \approx t$.

Lemma 3 *Every formula $\phi \in DPL(\sigma)$ is equivalent to a formula of one of the following forms (for some ψ, x, χ, σ, where σ is full):*

1. $\neg\neg\chi; \sigma$.
2. $\psi; \exists x; \neg\neg\chi; \sigma$.

Proof: *First rewrite ϕ into a formula that contains only full substitutions. After that, the only non-trivial case in the translation instruction is the case of $\tau; \psi$, where τ is full and ψ is of the first form, i.e., where ψ is equivalent to $\neg\neg\chi; \sigma$, for some χ, σ, with σ full. In this case, $\tau; \psi$ is equivalent to $\neg\neg(\tau; \chi); \sigma \circ \tau$, where $\sigma \circ \tau$ is full because σ and τ are.* □

Theorem 6 *$(\exists x \cup \exists y)$ cannot be expressed in $DPL(\sigma)$.*
Proof: *Suppose $\phi \in DPL(\sigma)$ is equivalent to $(\exists x \cup \exists y)$. Take a model with as domain the natural numbers, and let R be the interpretation of ϕ. By Lemma 3, it follows that ϕ is equivalent to $\psi; \exists z; \neg\neg\chi; \sigma$, for some formulas ψ, χ, some variable z and some full substitution σ (otherwise, ϕ would be deterministic). Two cases can be distinguished.*

1. *z does not occur in σ. Without loss of generality, assume that $z \neq x$. Take any pair of assignments g, h such that $g \neq h$ and $g \sim_x h$. Then gRh. Take any $k \neq h$ such that $k \sim_z h$. Then gRk, but g and k differ with respect to two variables (x and z), which is in contradiction with the fact that ϕ is equivalent to $(\exists x \cup \exists y)$.*
2. *z occurs in σ. By the fact that there are no function symbols involved, and by the fact that σ is full, there must be exactly one binding in σ of the form $u := z$. We can apply the same argument as before, now using u instead of z, and again we arrive at a contradiction.*

□

Since every substitution is equivalent to a DPL formula containing only full substitutions, and since every full substitution without function symbols is a renaming, and therefore has a converse that is also a renaming, we get:

Lemma 4 *Every converse substitution containing no function symbols is equivalent to a formula in $DPL(\sigma)$.*

This immediately gives:

Corollary 1 *$(\exists x \cup \exists y)$ cannot be expressed in $DPL(\sigma, ^\smile)$.*

By a straightforward induction we get:

Lemma 5 *Every formula in $DPL(\sigma, \cup)$ is equivalent to a formula of the form $\phi_1 \cup \ldots \cup \phi_n$ ($n \geq 1$) where each $\phi_i \in DPL(\sigma)$.*

Theorem 7 *$(x := f(x))^\smile$ cannot be expressed in $DPL(\sigma, \cup)$.*
Proof: *Suppose $\phi \in DPL(\sigma, \cup)$ is equivalent to $(x := f(x))^\smile$. By Lemma 5, we can assume that ϕ is of the form $\phi_1 \cup \ldots \cup \phi_n$, where each $\phi_i \in DPL(\sigma)$. Consider the model with as domain $\{0, \ldots, n\}$, and where f is interpreted as the "successor modulo $n+1$" function.*

Let us say that a relation R fixes a variable x if for $\forall gh \in cod(R)$: $g \sim_x h$ implies that $h = g$. Analyzing each ϕ_i, we can distinguish the following two cases.

- *ϕ_i is equivalent to $\neg\neg\chi; \sigma$, with σ full. Then $[\![\phi_i]\!]$ fixes x.*
- *ϕ_i is equivalent to $\psi; \exists y; \neg\neg\chi; \sigma$, again with σ full. If y occurs in σ, then let z_i be the (unique) variable such that σ contains a binding of the form $z_i := f^k(y)$. If σ does not contain y then let $z_i = y$. Then it must be the case that $[\![\phi_i]\!]$ fixes z_i, for otherwise $[\![\phi_i]\!]$ is not injective.*

Thus, we have that every ϕ_i fixes some variable z_i. Let $\{z_1, \ldots, z_m\}$ be all variables that are fixed by some ϕ_i (where $m \leq n$).

Consider all possible ways of assigning objects from the domain to the variables $z_1, \ldots z_m$ (assigning 0 to all other variables). This gives us $(n+1)^m$ assignments, each of which is in the co-domain of ϕ. Now, of this space of assignments, each ϕ_i can cover only a small part: at most $(n+1)^{m-1}$ (since one variable is fixed). So, together, $\phi_1, \ldots, \phi_n$ can cover at most $n * (n+1)^{m-1} = (n+1)^m - (n+1)^{m-1} < (n+1)^m$ assignments, which means that some assignments are not in the co-domain of ϕ. This is in contradiction with the fact that ϕ is equivalent to $(x := f(x))^\smile$. □

By symmetry, we get the following

Corollary 2 *$x := f(x)$ cannot be expressed in $DPL(\breve{\sigma}, \cup)$.*

Theorem 8 *$\exists y(y \approx x; \exists x; Rxy)$ cannot be expressed in $DPL(\cup, \sigma, ^\smile)$.*
Proof: *The same proof as for Theorem 7 can be used. Assume a signature without function symbols. Let the domain of the model be the set $\{0, \ldots, n\}$. Let R be interpreted as "successor modulo $n+1$". Then R is interpreted in the same way as f was in the proof of Theorem 7. Notice that, under this interpretation, $\exists y(y \approx x; \exists x; Rxy)$ means the same as $(x := f(x))^\smile$ did in the proof of Theorem 7. It follows that $\exists y(y = x; \exists x; Rxy)$ cannot be expressed in $DPL(\cup, \sigma)$. Since the signature contains no function symbols, it follows by Lemma 4 that this formula cannot be expressed in $DPL(\cup, \sigma, ^\smile)$ either.* □

6 Conclusion and Future Work

Our study of extensions of DPL started from an observation about the process of substitution in a dynamic setting. The substitution lemmas shed some further light on how variables are handled in DPL, and thus indirectly on what the processes of anaphoric reference and anaphora resolution in natural language analysis and variable declaration and variable use in programming have in common.

The formulas-as-programs languages that extend DPL hold promises for correctness reasoning and program analysis. Tableau style reasoning for DPL$(\sigma, \cup, ^*)$ could be used for generating counterexamples from incorrect programs. The formulas-as-programs perspective also simplifies the process of program inversion. Gries's example of the inversion of $\{x \approx 3\}\ x := 1$ to $\{x \approx 1\}\ x := 3$ involved a magical transformation of a command into an assertion and vice versa. In the formulas-as-programs perspective this becomes the mapping of $x \approx 3; \exists x; x \approx 1$ to $x \approx 1; \exists x; x \approx 3$, which is just a matter of two applications of the inversion law $(R \circ S)^\smile = S^\smile \circ R^\smile$. Inversion of substitutions can always be done by means of $^\smile$, but sometimes $\breve{\sigma}$ can again be written as a substitution: converses of renamings are themselves renamings, $(x := x+1)^\smile$ is equivalent to $x := x - 1$, and so on. Further analysis of these matters is future work.

References

[1] J. van Eijck, J. Heguiabehere, and B. Ó Nualláin. Tableau reasoning and programming with dynamic first order logic. *Logic Journal of the IGPL*, 9(3), May 2001.

[2] R. Goldblatt. *Logics of Time and Computation, Second Edition, Revised and Expanded*, volume 7 of *CSLI Lecture Notes*. CSLI, Stanford, 1992 (first edition 1987). Distributed by University of Chicago Press.

[3] D. Gries. *The Science of Programming*. Springer, Berlin, 1981.

[4] J. Groenendijk and M. Stokhof. Dynamic Montague Grammar. In L. Kalman and L. Polos, editors, *Papers from the Second Symposium on Logic and Language*, pages 3–48. Akademiai Kiadoo, Budapest, 1990.

[5] J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[6] D. Harel. *First-Order Dynamic Logic*. Number 68 in Lecture Notes in Computer Science. Springer, 1979.

[7] C. Vermeulen. A calculus of substitutions for DPL. *Studia Logica*, 68:357–387, 2001.

[8] A. Visser. Contexts in dynamic predicate logic. *Journal of Logic, Language and Information*, 7(1):21–52, 1998.