Programming with Dynamic Predicate Logic

Jan van Eijck

CWI, Amsterdam, ILLC, Amsterdam, Uil-OTS, Utrecht

EMAIL: jve@cwi.nl

Abstract

We propose to bring together two research traditions, computation with first order logic from computer science, and dynamic interpretation of first order logic from natural language semantics. We define a new executable process interpretation for first order logic, and show that it is a faithful approximation of the dynamic interpretation procedure for first order formulas. We then demonstrate the dynamic logic programming alternative to destructive assignment and show how to obtain a versatile logic programming language by adding constructs for bounded choice and for bounded and unbounded iteration. Finally, we show that the operational semantics for first order logic given in Apt and Bezem [1] is an approximation of our executable semantics. It follows that the operational semantics is faithful to the dynamic interpretation of first order logic. Our results relate a recent turn in executable computational interpretation of FOL formulas to a research tradition from natural language semantics, and suggest a new paradigm of dynamic logic programming that combines imperative power with dynamic declarative semantics.

1991 CR Subject Classification: F.3.1, F.3.2, F.4.1, I.2.4, I.2.7.
Keywords and Phrases: dynamic semantics, declarative programming, first order logic, logic programming.

1 The Dynamic Interpretation of FOL

A dynamic interpretation of an extension of first order predicate logic was proposed for purposes of natural language semantics in Barwise [5]. A streamlined first order version called DPL is given in Groenendijk and Stokhof [11], and a German variant on the theme can be found in Staudacher [12]. The key idea of these proposals is to represent introduction of new items of discourse by means of random assignment to a variable, and let pronouns pick up anaphoric references to previously mentioned items of discourse by reading off the value of the appropriate variable. The introduction of a referent takes place by means of existential quantification, suitably reinterpreted as follows. The meaning of $\exists x$ becomes an action of random reset of the value of x. In other words, $\exists x$ changes an input variable state by resetting its x value to a random new value. Recent work by Apt c.s. ([2, 1]) suggests a new variation on this theme.

A first order signature is a triple $(\mathbf{P}, \mathbf{F}, \mathbf{Ar})$, where $\mathbf{P}, \mathbf{F}$ are disjoint finite sets of symbols (the relation and function symbols, respectively), and $\mathbf{Ar}$ is a function from $\mathbf{P} \cup \mathbf{F}$ to the natural numbers (the arity function). Let a first order signature and a set of variables V be given. Let v range over V, f and P over the function and relation symbols, with arities n as specified by the signature. Then the terms and formulas of our dynamic version of first order logic over this signature are given by:

Definition 1 (Terms and formulas of L)

$$
\begin{array}{lcl}
t & ::= & v \mid ft_1 \cdots t_n \\
\phi & ::= & \perp \mid Pt_1 \cdots t_n \mid t_1 \doteq t_2 \mid \exists v \mid \neg\phi \mid (\phi_1 ; \phi_2) \mid (\phi_1 \cup \phi_2)
\end{array}
$$

This language is an extension of the DPL language studied by Groenendijk and Stokhof. Their language does not have a union (or disjunction) operator. For our purposes the extension with $\cup$ will be essential.

If you want to insist on appearances for a first order language, just replace $\exists v; \phi$ by $\exists v\phi$, replace other occurrences of ; by $\wedge$ and replace $\cup$ by $\vee$. The promotion of the quantifier prefix $\exists v$ to the status of a formula in its own right reflects the fact that quantification is thought of as an *action* that changes a variable state. The choice of ; for conjunction and $\cup$ for disjunction reflect the facts that dynamically, conjunction behaves like sequential composition and therefore is associative but not commutative, while disjunction behaves like set theoretic union and is both associative and commutative.

We will first give the (by now) standard dynamic semantics of this language and then turn to a computational process interpretation that was inspired by the computational interpretation of logical formulas proposed by Apt and Bezem [1].

The standard interpretation of a predicate dynamic logic formula in a language over variable set V in a first order model $\mathcal{M} = (M, I)$ is a binary relation on the set M^V, the set of valuations or variable states for that model, or equivalently, a function in $M^V \to \mathcal{P}(M^V)$. We will adopt the functional perspective. Let $A \subseteq M^V$ and let a be a member of M^V. Use $a \sim_v b$ for: a and b differ at most in their v value. The interpretation function for dynamic predicate logic is now given by:

Definition 2 (Dynamic interpretation of L in a model $\mathcal{M} = (M, I)$)

$$
\begin{array}{lcl}
v^a & := & a(v) \\
(ft_1 \cdots t_n)^a & := & I(f)t_1^a \cdots t_n^a \\
[\![\perp]\!](a) & := & \emptyset \\
[\![Pt_1 \cdots t_n]\!](a) & := & \left\{ \begin{array}{ll} \{a\} & \textit{if } (t_1^a, \ldots, t_n^a) \in I(P) \\ \emptyset & \textit{otherwise.} \end{array} \right. \\
[\![t_1 \doteq t_2]\!](a) & := & \left\{ \begin{array}{ll} \{a\} & \textit{if } t_1^a = t_2^a \\ \emptyset & \textit{otherwise.} \end{array} \right. \\
[\![\exists v]\!](a) & := & \{a' \in M^V \mid a' \sim_v a\} \\
[\![\neg\phi]\!](a) & := & \left\{ \begin{array}{ll} \{a\} & \textit{if } [\![\phi]\!](a) = \emptyset \\ \emptyset & \textit{otherwise.} \end{array} \right. \\
[\![\phi_1 ; \phi_2]\!](a) & := & \bigcup\{[\![\phi_2]\!](a') \mid a' \in [\![\phi_1]\!](a)\} \\
[\![\phi_1 \cup \phi_2]\!](a) & := & [\![\phi_1]\!](a) \cup [\![\phi_2]\!](a)
\end{array}
$$

The clauses make clear that ; is indeed interpreted as composition, while $\cup$ is interpreted as union of possible outputs.

Note that the interpretation function is in general not finite-valued, and that this is due only to the semantics of the quantifier. For if M is infinite, the set $\{b \in M^V \mid b \sim_v a\}$ will be infinite, so the interpretation of $\exists v$ will give rise to infinite branching in the transition system representing the dynamic interpretation. No other construct of the language gives rise to infinite branching in the semantics.

To appreciate the dynamic flavour of this process interpretation of first order logic, we define the variables that determine the input behaviour and those that may be affected during the interpretation process. The variables in the first class we call *dynamically free*, those in the second class *dynamically active*. These classes are defined by simultaneous recursion, as follows (we use $var(t)$ for the variables of a term, and $var(A)$ for the variables of an atom):

Definition 3 (Dynamically free variables, dynamically active variables)

$$\begin{array}{lll@{\qquad}lll}
df(\bot) & := & \emptyset & da(\bot) & := & \emptyset \\
df(\exists v) & := & \emptyset & da(\exists v) & := & \{v\} \\
df(Pt_1 \cdots t_n) & := & var(Pt_1 \cdots t_n) & da(Pt_1 \cdots t_n) & := & \emptyset \\
df(t_1 \doteq t_2) & := & var(t_1 \doteq t_2) & da(t_1 \doteq t_2) & := & \emptyset \\
df(\neg\phi) & := & df(\phi) & da(\phi) & := & \emptyset \\
df(\phi_1;\phi_2) & := & df(\phi_1) \cup (df(\phi_2) - da(\phi_1)) & da(\phi_1;\phi_2) & := & da(\phi_1) \cup da(\phi_2) \\
df(\phi_1 \cup \phi_2) & := & df(\phi_1) \cup df(\phi_2) & da(\phi_1 \cup \phi_2) & := & da(\phi_1) \cup da(\phi_2).
\end{array}$$

The dynamically free variables of ϕ are the members of $df(\phi)$, the dynamically active variables of ϕ the members of $da(\phi)$.

Note how this definition differs from the usual definition of freedom for first order logic. For example, $\exists x; (Px; Rxy); Qx$ does not have any dynamically free occurrences of x.

Recall the finiteness lemma in first order logic, which states that the truth value of ϕ in $\mathcal{M}$ under a depends only on the values that a assigns to the free variables of ϕ. Under the process interpretation of first order logic, this lemma splits into two parts, one for input behaviour and one for output behaviour. The two lemmas confirm that the above definition of dynamically free and dynamically active variable occurrences does what it is supposed to do. The first lemma confirms that the dynamically free variables of a formula are the variables that determine whether successful computations are possible for a given input.

Lemma 4 (Input Finiteness Lemma) *For all $a, b \in M^V$: If a and b agree on the variables in $df(\phi)$, then $[\![\phi]\!](a) = \emptyset$ iff $[\![\phi]\!](b) = \emptyset$.*

Proof. Induction on the structure of ϕ. □

The corresponding lemma about output behaviour is phrased in terms of the dynamically active variables of a formula.

Lemma 5 (Output Finiteness Lemma) *For all formulas ϕ, all valuations $a, b \in M^V$: If $b \in [\![\phi]\!](a)$ then b differs from a at most on the variables in $da(\phi)$.*

Proof. Induction on the structure of ϕ. □

Another feature worth mentioning is that DPL formulas can be reversed. Here is how.

Definition 6

$$\begin{array}{rcl}
\bot^r & := & \bot \\
(Pt_1 \cdots t_n)^r & := & Pt_1 \cdots t_n \\
(t_1 \doteq t_2)^r & := & t_1 \doteq t_2 \\
(\exists v)^r & := & \exists v \\
(\neg\phi)^r & := & \neg\phi \\
(\phi \cup \psi)^r & := & \phi^r \cup \psi^r \\
(\phi;\psi)^r & := & \psi^r;\phi^r.
\end{array}$$

Theorem 7 *For all formulas ϕ, all valuations $a, b \in M^V$: $b \in [\![\phi]\!](a)$ iff $a \in [\![\phi^r]\!](b)$.*

Proof. Induction on the structure of ϕ. $\Box$

The interest for us of this feature is that it will allow us, in principle at least, to run dynamic predicate logic programs backwards.

The relation between dynamic predicate logic and standard predicate logic is given by the following two translation functions characterizing the input and output states (call these forward and backward translations, respectively). We use $\Box$ as an abbreviation for $\neg\bot$, and T for a formula of the form $\bot, Pt_1 \cdots t_n, t_1 \doteq t_2, \neg\phi$ (T for test formula).

Definition 8

$$
\begin{array}{lcllcl}
\bot^{\triangleright} & := & \bot & \bot^{\triangleleft} & := & \bot \\
(Pt_1 \cdots t_n)^{\triangleright} & := & Pt_1 \cdots t_n & (Pt_1 \cdots t_n)^{\triangleleft} & := & Pt_1 \cdots t_n \\
(t_1 \doteq t_2)^{\triangleright} & := & t_1 \doteq t_2 & (t_1 \doteq t_2)^{\triangleleft} & := & t_1 \doteq t_2 \\
(\exists v)^{\triangleright} & := & \exists v \Box & (\exists v)^{\triangleleft} & := & \exists v \Box \\
(\neg\phi)^{\triangleright} & := & \neg\phi^{\triangleright} & (\neg\phi)^{\triangleleft} & := & \neg\phi^{\triangleright} \\
(\phi_1 \cup \phi_2)^{\triangleright} & := & \phi_1^{\triangleright} \vee \psi_2^{\triangleright} & (\phi_1 \cup \phi_2)^{\triangleleft} & := & \phi_1^{\triangleleft} \vee \psi_2^{\triangleleft} \\
((\phi_1;\phi_2);\phi_3)^{\triangleright} & := & (\phi_1;(\phi_2;\phi_3))^{\triangleright} & (\phi_1;(\phi_2;\phi_3))^{\triangleleft} & := & ((\phi_1;\phi_2);\phi_3)^{\triangleleft} \\
(T;\phi)^{\triangleright} & := & (T^{\triangleright} \wedge \phi^{\triangleright}) & (\phi;T)^{\triangleleft} & := & (\phi^{\triangleleft} \wedge T^{\triangleleft}) \\
(\exists v;\phi)^{\triangleright} & := & \exists v \phi^{\triangleright} & (\phi;\exists v)^{\triangleleft} & := & \exists v \phi^{\triangleleft} \\
(\phi_1 \cup \phi_2;\psi)^{\triangleright} & := & (\phi_1;\psi)^{\triangleright} \vee (\phi_1;\psi)^{\triangleright} & (\psi;\phi_1 \cup \phi_2)^{\triangleleft} & := & (\psi;\phi_1)^{\triangleleft} \vee (\psi;\phi_2)^{\triangleleft}
\end{array}
$$

Theorem 9 *For all models $\mathcal{M} = (M, I)$, all formulas ϕ, all valuations a, b:*

1. *If $b \in [\![\phi]\!](a)$ then $\mathcal{M}, a \models \phi^{\triangleright}$ and $\mathcal{M}, b \models \phi^{\triangleleft}$.*
2. *$\mathcal{M} \models_a \phi^{\triangleright}$ iff $[\![\phi]\!](a) \neq \emptyset$.*
3. *$\mathcal{M} \models_b \phi^{\triangleleft}$ iff $\{a \mid b \in [\![\phi]\!](a)\} \neq \emptyset$.*

Proof. Claim 1. is proved by induction on the structure of ϕ. Claims 2 and 3 follow from 1. $\Box$

Another, equivalent way to arrive at a formula for the output states, is by taking $\phi^{\triangleleft} := (\phi^r)^{\triangleright}$.

Since a dynamically interpreted formula denotes a relation, we can study weakest preconditions and strongest postconditions of dynamic formulas. The weakest precondition of a dynamic formula ϕ for a (static) condition ψ is the (static) condition that holds at precisely those input valuations a with the property that all b with $b \in [\![\phi]\!](a)$ satisfy ψ. The strongest postcondition of a dynamic formula ϕ for a (static) condition ψ is the (static) condition that holds at precisely those output valuations b with the property that all a with $b \in [\![\phi]\!](a)$ satisfy ψ. The strongest postcondition and the weakest precondition with respect to condition ψ of a dynamic formula ϕ are given by the following definitions.

Definition 10

$$
\begin{array}{lcllcl}
\mathrm{STP}_{\bot}(\psi) & := & \Box & \mathrm{WPR}_{\bot}(\psi) & := & \Box \\
\mathrm{STP}_{\mathrm{Pt_1 \cdots t_n}}(\psi) & := & Pt_1 \cdots t_n \wedge \psi & \mathrm{WPR}_{\mathrm{Pt_1 \cdots t_n}}(\psi) & := & Pt_1 \cdots t_n \rightarrow \psi \\
\mathrm{STP}_{\mathrm{t_1 \doteq t_2}}(\psi) & := & t_1 \doteq t_2 \wedge \psi & \mathrm{WPR}_{\mathrm{t_1 \doteq t_2}}(\psi) & := & t_1 \doteq t_2 \rightarrow \psi \\
\mathrm{STP}_{\exists \mathrm{v}}(\psi) & := & \exists v \psi & \mathrm{WPR}_{\exists \mathrm{v}}(\psi) & := & \forall v \psi \\
\mathrm{STP}_{\neg\phi}(\psi) & := & \mathrm{WPR}_{\phi}\bot \wedge \psi & \mathrm{WPR}_{\neg\phi}(\psi) & := & \mathrm{WPR}_{\phi}\bot \rightarrow \psi \\
\mathrm{STP}_{\phi_1 \cup \phi_2}(\psi) & := & \mathrm{STP}_{\phi_1}(\psi) \vee \mathrm{STP}_{\phi_2}(\psi) & \mathrm{WPR}_{\phi_1 \cup \phi_2}(\psi) & := & \mathrm{WPR}_{\phi_1}(\psi) \wedge \mathrm{WPR}_{\phi_2}(\psi) \\
\mathrm{STP}_{\phi_1;\phi_2}(\psi) & := & \mathrm{STP}_{\phi_2}(\mathrm{STP}_{\phi_1}(\psi)) & \mathrm{WPR}_{\phi_1;\phi_2}(\psi) & := & \mathrm{WPR}_{\phi_1}(\mathrm{WPR}_{\phi_2}(\psi))
\end{array}
$$

Theorem 11 confirms that these definitions are correct. Theorem 12 relates weakest preconditions to the forward and backward translations of dynamic formulas.

Theorem 11 *For all formulas ϕ of L, all FOL ψ over the same signature:*

1. $\mathcal{M} \models_a \mathrm{WPR}_\phi(\psi)$ *iff for all* $b \in [\![\phi]\!](a)$ *it holds that* $\mathcal{M} \models_b \psi$.
2. $\mathcal{M} \models_b \mathrm{STP}_\phi(\psi)$ *iff for all* a *with* $b \in [\![\phi]\!](a)$ *it holds that* $\mathcal{M} \models_a \psi$.

Proof. Induction on the structure of ϕ. □

Theorem 12 *For all formulas ϕ of L:*

1. $\phi^{\triangleright} \equiv \neg\mathrm{WPR}_\phi(\bot)$.
2. $\phi^{\triangleleft} \equiv \neg\mathrm{WPR}_{\phi^r}(\bot)$.

Proof. Claim 1: $\mathcal{M} \models_a \phi^{\triangleright}$ iff $[\![\phi]\!](a) \neq \emptyset$ iff $\mathcal{M} \not\models_a \mathrm{WPR}_\phi(\bot)$ iff $\mathcal{M} \models_a \neg\mathrm{WPR}_\phi(\bot)$.

Claim 2: $\mathcal{M} \models_b \phi^{\triangleleft}$ iff $\{a \mid b \in [\![\phi]\!](a)\} \neq \emptyset$ iff $\exists a : b \in [\![\phi]\!](a)$ iff $[\![\phi^r]\!](b) \neq \emptyset$ iff $\mathcal{M} \not\models_b \mathrm{WPR}_{\phi^r}(\bot)$ iff $\mathcal{M} \models_b \neg\mathrm{WPR}_{\phi^r}(\bot)$. □

Pre- and postcondition reasoning for dynamic predicate logic was first explored in Van Eijck and De Vries [9]. A streamlined version of the calculus can be found in Van Eijck [8], but the essence is in Definition 10.

2 An Executable Process Interpretation for First Order Logic

An executable version of dynamic predicate logic assumes that we work with partial rather than total valuations.[1] Let $\mathcal{A} := \bigcup_{X \subseteq V} M^X$, let ϵ be the empty partial valuation (the only member of $M^\emptyset$), and let $\bullet$ be an object not in $\mathcal{A}$.

If $a \in M^X$, for $X \subseteq V$, then a term t is a-closed if all variables in t are in X, an atom $Pt_1 \cdots t_n$ is a-closed if all t_i are a-closed, and an identity $t_1 \doteq t_2$ is a-closed if both of t_1, t_2 are. If $a \in M^X$ we call X the domain of a. We will use $dom(a)$ to refer to the domain of a. Note that we do not impose the condition that domains of valuations are finite.

In the spirit of Apt and Bezem [1], identities of the form $v \doteq t$ and $t \doteq v$ may function as assignment statements, namely in those cases where v is not closed for the input valuation, but t is. The value $\bullet$ (for 'unknown') is generated in case $t_1 \doteq t_2$ can neither be interpreted as a test for identity (because $t_1 \doteq t_2$ is not closed for the input valuation) nor as an assignment statement (because it is not of the appropriate form or does not satisfy the appropriate conditions for that).

In standard dynamic semantics, the existential quantifier is interpreted as random assignment, but from a computational point of view that interpretation is awkward. If the model has an infinite domain (as any model that includes the natural numbers has) random assignment blows up the output valuation space to infinity. If the domain is the natural numbers then, assuming for an instant that the variable set consists of just this one variable, we get that the output of $[\![\exists x]\!]$ for any input state becomes:

$$\{\{x/0\}, \{x/1\}, \{x/2\}, \{x/3\}, \ldots\}.$$

This is impractical if one wants to keep track of the output valuation space for purposes of computation, so we need an alternative for the quantifier rule that keeps the computation feasible.

[1] Visser [13] calls a partial valuation a *local* valuation.

The treatment of the existential quantifier that squares with the executable interpretation of identities turns out to be the following. An existential quantification over v clears the variable space for v. Thus, if the carrier of the input valuation does not include v, the interpretation of $\exists v$ gives back the input valuation as its only possible output, otherwise the effect of $\exists v$ is to delete the pair consisting of v and its value from the input valuation.

In the definition of term interpretations, we use $\uparrow$ for 'undefined' and $\downarrow$ for 'defined'.

Definition 13 (Term interpretation in a model $\mathcal{M} = (M, I)$ wrt valuation a)

$$
\begin{array}{lll}
v^a & := & \begin{cases} a(v) & \textit{if } v \textit{ is } a\textit{-closed} \\ \uparrow & \textit{otherwise} \end{cases} \\
(ft_1 \cdots t_n)^a & := & \begin{cases} I(f)t_1^a \cdots t_n^a & \textit{if } t_1, \ldots, t_n \ a\textit{-closed} \\ \uparrow & \textit{otherwise} \end{cases}
\end{array}
$$

Before we can give the executable interpretation function we need some more preliminary definitions.

Definition 14 *An identity $t_1 \doteq t_2$ is an a-assignment if either $t_1 \equiv v$, $t_1^a = \uparrow$, $t_2^a = \downarrow$, or $t_2 \equiv v$, $t_1^a = \downarrow$, $t_2^a = \uparrow$.*

For a sound executable treatment of negation, we have several options. Suppose we evaluate in the natural numbers. Then the formula $x \doteq 1$ should succeed for the empty input valuation ϵ, and give output valuation $\{x/1\}$. Does this mean that we should make $\neg x \doteq 1$ fail for this input? No, for the meaning of this would be that for every natural number n it holds that $n = 1$, which we know is false.

The example shows that we need to be careful. But how careful exactly? In the example case, the problem is that the input valuation does have extensions for which the formula does *not* succeed. Valuation $\{x/2\}$ is an extension of ϵ, but for this input the formula $x \doteq 1$ fails. This shows that our treatment of negation should distinguish between cases where $a \xrightarrow{\phi} b$ has the so-called forward property (if $a \xrightarrow{\phi} b$, i.e., on input a, formula ϕ has b as a possible output, and if $a \subseteq a'$, then there is an extension $b' \supseteq b$ with $a' \xrightarrow{\phi} b'$) and cases where $a \xrightarrow{\phi} b$ does *not* have this property.[2]

The simplest way to guarantee the forward property is to require that all dynamically free variables of ϕ should be in the domain of a, for $\neg\phi$ to give a proper output valuation on input a. Formally, the requirement would be that $df(a) \subseteq dom(a)$. This solves the problem for the example above. We have: $\epsilon \xrightarrow{x \doteq 1} \{x/1\}$, but since x is dynamically free in $x \doteq 1$ but not in the domain of ϵ we cannot conclude from this that $\neg x \doteq 1$ fails. Rather we conclude that we cannot conclude anything from $\neg x \doteq 1$ for input ϵ.

A treatment along these lines would work all right, but it is a far from optimal solution, for it rules out much too much. Consider the following example.

(1) $\quad \{x/1\} \xrightarrow{x \doteq 1 \cup y \doteq 2} \{x/1\}$.

Example (1) satisfies the forward property: for any extension a' of the input valuation $\{x/1\}$ the test $x \doteq 1$ will succeed. This means we can safely say that $\neg(x \doteq 1 \cup y \doteq 2)$ fails for input $\{x/1\}$. Still, the condition that we imposed is *not* fulfilled: $df(x \doteq 1 \cup y \doteq 2) = \{x, y\}$, and this set is *not* included in the domain of $\{x/1\}$.

The example illustrates that we should distinguish between the input variables that are relevant to a particular path taken by the computation and the input variables that are not. In other words, we should

[2]The forward property is called the 'Zig property' in Visser [13], because it is one half of the Zig-Zag property that forms the main requirement for bisimulation.

look at what happens in the transition from a particular input to a particular output. If the domain of the output is equal to the domain of the input, this guarantees the forward property. This condition is fulfilled in example (1).

(2) $\{x/1\} \quad \stackrel{\exists x}{\longrightarrow} \quad \epsilon.$

(3) $\epsilon \quad \stackrel{\exists x; x \doteq 1}{\longrightarrow} \quad \{x/1\}.$

Unfortunately, this still rules out too much: examples (2) and (3) both have the forward property, yet neither of them satisfies the domain condition.

The trouble is that our representation is not fine-grained enough to allow us to distinguish between the input variables that are relevant to a particular path taken by the computation and the input variables that are not. To remedy this, we introduce two distinguished registers g (for the set of *global* variables that get assigned along a computation path) and l (for the set of *local* variables that get declared and possibly assigned along a computation path).

The output of a computation, for a given input valuation a, will be a triple (b, g^b, l^b), where b is a valuation, $g^b \subseteq V$ and $l^b \subseteq V$. Formally, this makes our computation function $[\![\cdot]\!]$ a function in $\mathcal{A} \to \mathcal{P}(\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V)$. For a smooth definition of the semantics of sequential composition, it is useful to have the global and local variable registers in the input as well, thus making our computation function$[\![\cdot]\!]$ into a function in $\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V \to \mathcal{P}(\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V)$. Finally, we have to take the fact into account that a computation may result in the outcome 'not enough information', and that a computation can also start from 'not enough information'. Thus, our executable process interpretation function is a function in $(\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V) \cup \{\bullet\} \to \mathcal{P}((\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V) \cup \{\bullet\})$.

The register g collects the variables that get assigned in the global variable space along a computation path, l the variables that get introduced in the local variable space along the computation path, i.e., the variables that are made dynamically active along a computation path by means of a dynamically scoped existential quantification. These are the variables local to the computation. But note that a variable can occur both globally and locally. E.g., in the formula $x = 1; \exists x; x = 2$. the first occurrence of x is global, but the quantifier blocks this global x from the rest of the computation, and the next occurrences of x are local.

Assuming that we get all this in place, the calculations for examples (1), (2), (3) would give:

$$
\begin{array}{rcl}
(\{x/1\}, \{x\}, \emptyset) & \stackrel{x \doteq 1 \cup y \doteq 2}{\longrightarrow} & (\{x/1\}, \{x\}, \emptyset). \\
(\{x/1\}, \{x\}, \emptyset) & \stackrel{\exists x}{\longrightarrow} & (\epsilon, \{x\}, \{x\}) \\
(\epsilon, \emptyset, \emptyset) & \stackrel{\exists x; x \doteq 1}{\longrightarrow} & (\{x/1\}, \emptyset, \{x\}).
\end{array}
$$

Since all variables in the global variable space that get assigned in the transition from (a, g^a, l^a) to (b, g^b, l^b) are in $g^b - dom(a)$, and all variables in the local variable space that possibly get reassigned in the transition from (a, g^a, l^a) to (b, g^b, l^b) are in l^a, a sufficient condition to guarantee that $(a, g^a, l^a) \stackrel{\phi}{\longrightarrow} (b, g^b, l^b)$ has the forward property can now be stated as: $l^a \cup g^b \subseteq dom(a)$. That this condition is indeed sufficient for the forward property will be proved in Lemma 30.

We will use $\mathbf{a}$ and $\mathbf{b}$ to range over members of $(\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V) \cup \{\bullet\}$, and $\mathbf{B}$ to range over subsets of $(\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V) \cup \{\bullet\}$. If $\mathbf{b} \neq \bullet$ it is understood that $\mathbf{b} = (b, g^b, l^b)$.

Definition 15

1. *$\mathbf{b}$ is safe for (a, g^a, l^a) if $\mathbf{b} \neq \bullet$ and $l^a \cup g^b \subseteq dom(a)$.*

2. $\boldsymbol{B} \subseteq (\mathcal{PA} \times \mathcal{PV} \times \mathcal{PV}) \cup \{\bullet\}$ *is risky for* (a, g^a, l^a) *if* $\boldsymbol{B} \neq \emptyset$*, but no member of* $\boldsymbol{B}$ *is safe for* (a, g^a, l^a)*.*

The executable process interpretation of first order predicate logic is the function in

$$(\mathcal{A} \times \mathcal{PV} \times \mathcal{PV}) \cup \{\bullet\} \to \mathcal{P}((\mathcal{A} \times \mathcal{PV} \times \mathcal{PV}) \cup \{\bullet\})$$

given by the following definition. The definition assumes a ranges over $\mathcal{A}$, g^a, l^a over $\mathcal{PV}$, $\mathbf{b}$ over $(\mathcal{A} \times \mathcal{PV} \times \mathcal{PV}) \cup \bullet$.

Definition 16 (Executable process interpretation in a model $\mathcal{M} = (M, I)$)

$$
\begin{array}{lcl}
[\![\phi]\!](\bullet) & := & \{\bullet\} \\
[\![\bot]\!](a, g^a, l^a) & := & \emptyset \\
[\![Pt_1 \cdots t_n]\!](a, g^a, l^a) & := & \begin{cases} \{(a, g^a, l^a)\} & \textit{if } Pt_1 \cdots t_n \textit{ a-closed}, (t_1^a, \ldots, t_n^a) \in I(P), \\ \emptyset & \textit{if } Pt_1 \cdots t_n \textit{ a-closed}, (t_1^a, \ldots, t_n^a) \notin I(P), \\ \{\bullet\} & \textit{if } Pt_1 \cdots t_n \textit{ not a-closed.} \end{cases} \\
[\![t_1 \doteq t_2]\!](a, g^a, l^a) & := & \begin{cases} \{(a, g^a, l^a)\} & \textit{if } t_1 \doteq t_2 \textit{ a-closed}, t_1^a = t_2^a), \\ \emptyset & \textit{if } t_1 \doteq t_2 \textit{ a-closed}, t_1^a \neq t_2^a), \\ \{(a \cup \{v/t_2^a\}, g^a, l^a)\} & \textit{if } t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \quad t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \in l^a \\ \{(a \cup \{v/t_2^a\}, g^a \cup \{v\}, l^a)\} & \textit{if } t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \quad t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \notin l^a \\ \{(a \cup \{v/t_1^a\}, g^a, l^a)\} & \textit{if } t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \quad t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \in l^a \\ \{(a \cup \{v/t_1^a\}, g^a \cup \{v\}), l^a)\} & \textit{if } t_1 \doteq t_2 \textit{ an a-assignment with} \\ & \quad t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \notin l^a \\ \{\bullet\} & \textit{if } t_1 \doteq t_2 \textit{ not a-closed} \\ & \textit{and not an a-assignment} \end{cases} \\
[\![\exists v]\!](a, g^a, l^a) & := & \begin{cases} \{(a, g^a, l^a \cup \{v\})\} & \textit{if } v \notin \textit{dom}(a) \\ \{(a - \{v/v^a\}, g^a, l^a \cup \{v\})\} & \textit{otherwise.} \end{cases} \\
[\![\neg\phi]\!](a, g^a, l^a) & := & \begin{cases} \{(a, g^a, l^a)\} & \textit{if } [\![\phi]\!](a, g^a, l^a) = \emptyset \\ \emptyset & \textit{if } \exists \boldsymbol{b} \in [\![\phi]\!](a, g^a, l^a) \textit{ with } \boldsymbol{b} \textit{ safe for } (a, g^a, l^a), \\ \{\bullet\} & \textit{if } [\![\phi]\!](a, g^a, l^a) \textit{ is risky for } (a, g^a, l^a). \end{cases} \\
[\![\phi_1; \phi_2]\!](a, g^a, l^a) & := & \bigcup\{[\![\phi_2]\!](\boldsymbol{b}) \mid \boldsymbol{b} \in [\![\phi_1]\!](a, g^a, l^a)\} \\
[\![\phi_1 \cup \phi_2]\!](a, g^a, l^a) & := & [\![\phi_1]\!](a, g^a, l^a) \cup [\![\phi_2]\!](a, g^a, l^a)
\end{array}
$$

As this definition is rather involved, it is perhaps illuminating to reformulate it as a set of transition rules, where $\mathbf{a} \xrightarrow{\phi} \mathbf{b}$ expresses that $\mathbf{b} \in [\![\phi]\!](\mathbf{a})$. This reformulation is given in Figures 1 and 2.

It can easily be verified by structural induction on formulas that this defines a function with domain and range as specified, and that the function is finite-valued for any input.

The special object $\bullet$ should be read as 'not enough information'. Once generated, $\bullet$ is propagated. The main differences with the standard dynamic semantics have to do with the presence of $\bullet$, with the executable interpretation of equality statements, with the executable interpretation of quantification, and with the presence of the registers for global and local variables and their role in the proper treatment of negation. All of this will be clarified in the examples below.

The following theorem provides a useful check on the definition.

Theorem 17 (Associativity of Composition)
For all ϕ_1, ϕ_2, ϕ_3*, all* $\boldsymbol{a}, \boldsymbol{b} \in (\mathcal{A} \times \mathcal{PV} \times \mathcal{PV}) \cup \{\bullet\}$*:*

$$\boldsymbol{a} \overset{\phi_1;(\phi_2;\phi_3)}{\longrightarrow} \boldsymbol{b} \textit{ iff } \boldsymbol{a} \overset{(\phi_1;\phi_2);\phi_3}{\longrightarrow} \boldsymbol{b}.$$

Figure 1: Executable Process Interpretation as a Transition System (1): The treatment of Equality

$t_1 \doteq t_2$	$\dfrac{t_1 \doteq t_2 \ a\text{-closed and } t_1^a = t_2^a}{(a, g^a, l^a) \xrightarrow{t_1 \doteq t_a} (a, g^a, l^a)}$ $\dfrac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \in l^a}{(a, g^a, l^a) \xrightarrow{t_1 \doteq t_a} (a \cup \{v/t_2^a\}, g^a, l^a)}$ $\dfrac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_1 \equiv v, v^a = \uparrow, t_2^a = \downarrow, v \notin l^a}{(a, g^a, l^a) \xrightarrow{t_1 \doteq t_a} (a \cup \{v/t_2^a\}, g^a \cup \{v\}, l^a)}$ $\dfrac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \in l^a}{(a, g^a, l^a) \xrightarrow{t_1 \doteq t_a} (a \cup \{v/t_1^a\}, g^a, l^a)}$ $\dfrac{t_1 \doteq t_2 \text{ an } a\text{-assignment with } t_2 \equiv v, v^a = \uparrow, t_1^a = \downarrow, v \notin l^a}{(a, g^a, l^a) \xrightarrow{t_1 \doteq t_a} (a \cup \{v/t_1^a\}, g^a \cup \{v\}, l^a)}$ $\dfrac{t_1 \doteq t_2 \text{ not an } a\text{-assignment and not } a\text{-closed}}{(a, g^a, l^a) \xrightarrow{t_1 \doteq t_a} \bullet}$

Proof. For the case of $\mathbf{a} = \bullet$, the claim obviously holds, by the propagation of $\bullet$.

For the case of $\mathbf{a} = (a, g^a, l^a)$, $\mathbf{b} = \bullet$, a quick check of the definition ensures that the claim holds.

For the case of $\mathbf{a} = (a, g^a, l^a)$, $\mathbf{b} = (b, g^b, l^b)$, we reason as follows:

$(a, g^a, l^a) \xrightarrow{\phi_1;(\phi_2;\phi_3)} (b, g^b, l^b)$

iff there are c, g^c, l^c with $(a, g^a, l^a) \xrightarrow{\phi_1} (c, g^c, l^c) \xrightarrow{\phi_2;\phi_3} (b, g^b, l^b)$

iff there are c, g^c, l^c, d, g^d, l^d with $(a, g^a, l^a) \xrightarrow{\phi_1} (c, g^c, l^c) \xrightarrow{\phi_2} (d, g^d, l^d) \xrightarrow{\phi_3} (b, g^b, l^b)$

iff there are d, g^d, l^d with $(a, g^a, l^a) \xrightarrow{\phi_1;\phi_2} (d, g^d, l^d) \xrightarrow{\phi_3} (b, g^b, l^b)$

iff $(a, g^a, l^a) \xrightarrow{(\phi_1;\phi_2);\phi_3} (b, g^b, l^b)$. □

The following theorem shows that the global and local variable stores keep track of all assignment and quantifier actions (we use $a[X]b$ for: if $v \notin X$ then $v^a = v^b$):

Theorem 18 *If* $(a, dom(a), \emptyset) \xrightarrow{\phi} (b, g^b, l^b)$ *then* $a[g^b \cup l^b]b$.

Proof. Induction on the structure of ϕ. □

For purposes of example presentation, it is convenient to reformulate the process interpretation as a function

$$[\![\cdot]\!] : \mathcal{P}(\mathcal{A} \cup \{\bullet\}) \to \mathcal{P}(\mathcal{A} \cup \{\bullet\}),$$

Figure 2: Executable Process Interpretation as a Transition System (2): Other Constructs

$\bullet$ propagation	$\dfrac{}{\bullet \xrightarrow{\phi} \bullet}$
$\perp$	no $\xrightarrow{\perp}$ transitions from (a, g^a, l^a)
$Pt_1 \cdots t_n$	$\dfrac{Pt_1 \cdots t_n \ a\text{-closed and } (t_1^a, \ldots, t_n^a) \in I(P)}{(a, g^a, l^a) \xrightarrow{Pt_1 \cdots t_n} (a, g^a, l^a)}$ $\dfrac{Pt_1 \cdots t_n \ \text{ not } a\text{-closed}}{(a, g^a, l^a) \xrightarrow{Pt_1 \cdots t_n} \bullet}$
$\exists v$	$\dfrac{v \notin dom(a)}{(a, g^a, l^a) \xrightarrow{\exists v} (a, g^a, l^a \cup \{v\})}$ $\dfrac{v \in dom(a)}{(a, g^a, l^a) \xrightarrow{\exists v} (a - \{v/v^a\}, g^a, l^a \cup \{v\})}$
$\neg\phi$	$\dfrac{\text{not } a \xrightarrow{\phi} \bullet, \text{ and there are no } b, g^b, l^b \text{ with } (a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b)}{(a, g^a, l^a) \xrightarrow{\neg\phi} (a, g^a, l^a)}$ $\dfrac{\text{there are } \xrightarrow{\phi} \text{ transitions for } (a, g^a, l^a), \text{ but none of them safe for } a}{(a, g^a, l^a) \xrightarrow{\neg\phi} \bullet}$
seq composition	$\dfrac{(a, g^a, l^a) \xrightarrow{\phi_1} \bullet}{(a, g^a, l^a) \xrightarrow{\phi_1 ; \phi_2} \bullet}$ $\qquad$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi_1} (b, g^b, l^b) \qquad (b, g^b, l^b) \xrightarrow{\phi_2} \bullet}{(a, g^a, l^a) \xrightarrow{\phi_1 ; \phi_2} \bullet}$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi_1} (b, g^b, l^b) \qquad (b, g^b, l^b)b \xrightarrow{\phi_2} (c, g^c, l^c)}{(a, g^a, l^a) \xrightarrow{\phi_1 ; \phi_2} (c, g^c, l^c)}$
union	$\dfrac{(a, g^a, l^a)a \xrightarrow{\phi_i} \bullet}{(a, g^a, l^a) \xrightarrow{\phi_1 \cup \phi_2} \bullet} \ i \in \{1, 2\}$ $\qquad$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi_i} (b, g^b, l^b)}{(a, g^a, l^a) \xrightarrow{\phi_1 \cup \phi_2} (b, g^b, l^b)} \ i \in \{1, 2\}$

and to write the process of applying this function in left-to-right order. Here is the definition:

Definition 19

1. *Let* $\natural : \mathcal{P}((\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V) \cup \{\bullet\}) \to \mathcal{P}(\mathcal{A} \cup \{\bullet\})$ *be given by:*

$$\boldsymbol{B}^\natural := \{b \mid (b, g^b, l^b) \in \boldsymbol{B}\} \cup \{\bullet \mid \bullet \in \boldsymbol{B}\}.$$

2. *Let* $[\![\cdot]\!] : \mathcal{P}(\mathcal{A} \cup \{\bullet\}) \to \mathcal{P}(\mathcal{A} \cup \{\bullet\})$ *be given by:*

$$[\![\phi]\!](A) := \left(\bigcup_{a \in A} \{[\{\phi\}](a, dom(a), \emptyset)\}\right)^\natural.$$

3. *Let* $\boldsymbol{B}[\![\phi]\!] \quad := \quad [\![\phi]\!](\boldsymbol{B})$.

Call a subset of $\mathcal{P}(\mathcal{A} \cup \{\bullet\})$ a valuation state. Since the global and local variable registers are only relevant for the treatment of negation, we can safely ignore them in the examples without occurrences of the negation sign. In the examples with negation, we only need to calculate the sets of global and local variables for sub-formulas in the scope of a negation sign.

First consider the example from [1]:

(4) $\quad (x = 2 \vee x = 3) \wedge (y = x + 1 \vee 2 = y) \wedge (2 * x = 3 * y).$

In more suggestive process notation this becomes:

$$(x = 2 \cup x = 3); (y = x + 1 \cup 2 = y); (2 * x = 3 * y).$$

For example (4), if we start with the valuation state $\{\epsilon\}$, we get the following.

$$\begin{array}{ll} & \{\epsilon\}[\![(x = 2 \cup x = 3); (y = x + 1 \cup 2 = y); (2 * x = 3 * y)]\!] \\ = & \{\epsilon\}[\![x = 2 \cup x = 3]\!][\![(y = x + 1 \cup 2 = y); (2 * x = 3 * y)]\!] \\ = & \{\{x/2\}, \{x/3\}\}[\![(y = x + 1 \cup 2 = y); (2 * x = 3 * y)]\!] \\ = & \{\{x/2\}, \{x/3\}\}[\![y = x + 1 \cup 2 = y]\!][\![2 * x = 3 * y]\!] \\ = & \{\{x/2, y/3\}, \{x/3, y/4\}, \{x/2, y/2\}, \{x/3, y/2\}\}[\![2 * x = 3 * y]\!] \\ = & \{\{x/3, y/2\}\} \end{array}$$

Next, consider example (5).

(5) $\quad \exists x(x = 0 \wedge x = y)) \wedge \exists x x = 1.$

Again we start with the minimal valuation state $\{\epsilon\}$, and we write $\wedge$ as ;.

$$\begin{array}{ll} & \{\epsilon\}[\![\exists x(x = 0; x = y); \exists x x = 1]\!] \\ = & \{\epsilon\}[\![\exists x(x = 0; x = y)]\!][\![\exists x x = 1]\!] \\ = & \{\epsilon\}[\![\exists x]\!][\![x = 0; x = y]\!][\![\exists x x = 1]\!] \\ = & \{\epsilon\}[\![x = 0; x = y]\!][\![\exists x x = 1]\!] \\ = & \{x/0, y/0\}[\![\exists x x = 1]\!] \\ = & \{x/0, y/0\}[\![\exists x]\!][\![x = 1]\!] \\ = & \{y/0\}[\![x = 1]\!] \\ = & \{x/1, y/0\} \end{array}$$

Next, we look at some examples where the value $\bullet$ plays a role.

(6) $x \doteq 1 \vee x > 2$.

The value $\bullet$ indicates that the computed answer may not be the full answer:

$$\begin{array}{rl} & \{\epsilon\}[\![x \doteq 1 \cup x > 2]\!] \\ = & (\{\epsilon\}[\![x \doteq 1]\!]) \cup (\{\epsilon\}[\![x > 2]\!]) \\ = & \{\{x/1\}\} \cup \{\bullet\} \\ = & \{\{x/1\}, \bullet\}. \end{array}$$

The following simple example illustrates the importance of $\bullet$ for the proper interpretation of negation.

(7) $\neg(x > 2)$.

For a proper understanding of the following outcome one should firmly keep in mind that the value $\bullet$ stands proxy for an unknown number of valuations (possibly zero) that cannot be computed:

$$\begin{array}{rl} & \{(\epsilon,\emptyset,\emptyset)\}[\![\neg(x > 2)]\!] \\ = & \{(\epsilon,\emptyset,\emptyset)\} \text{ if } \{(\epsilon,\emptyset,\emptyset)\}[\![x > 2]\!] = \emptyset, \{\bullet\} \text{ if } \{(\epsilon,\emptyset,\emptyset)\}[\![x > 2]\!] \text{ risky for } (\epsilon,\emptyset,\emptyset)\} \\ = & \{\bullet\} \text{ if } [\![x > 2]\!](\{(\epsilon,\emptyset,\emptyset)\}) \text{ risky for } (\epsilon,\emptyset,\emptyset) \\ = & \{\bullet\} \text{ if } \{\bullet\} \text{ risky for } (\epsilon,\emptyset,\emptyset) \\ = & \{\bullet\}. \end{array}$$

Note that in case we have an assignment instead of a relational test we get the same outcome:

$$\begin{array}{rl} & \{(\epsilon,\emptyset,\emptyset)\}[\![\neg x \doteq 2]\!] \\ = & \{(\epsilon,\emptyset,\emptyset)\} \text{ if } \{(\epsilon,\emptyset,\emptyset)\}[\![x \doteq 2]\!] = \emptyset, \{\bullet\} \text{ if } \{(\epsilon,\emptyset,\emptyset)\}[\![x \doteq 2]\!] \text{ risky for } (\epsilon,\emptyset,\emptyset)\} \\ = & \{\bullet\} \text{ if } \{(\epsilon,\emptyset,\emptyset)\}[\![x \doteq 2]\!] \text{ risky for } (\epsilon,\emptyset,\emptyset) \\ = & \{\bullet\} \text{ if } \{(\{x/2\},\{x\},\emptyset)\} \text{ risky for } (\epsilon,\emptyset,\emptyset) \\ = & \{\bullet\}. \end{array}$$

According to the letter of the definition, $(\{x/2\},\{x\},\emptyset)$ is risky for $(\epsilon,\emptyset,\emptyset)$ because $x \notin \emptyset$. Let us try to grasp the spirit of the definition, and ask ourselves *why* the outcome $(\{x/2\},\{x\},\emptyset)$ is not safe for $(\epsilon,\emptyset,\emptyset)$. Assume for an instant that it were safe. Then, because $x \doteq 2$ would succeed, $x \neq 2$ would fail for input $(\epsilon,\emptyset,\emptyset)$. This failure would then indicate that for all x in the domain, $x = 2$ holds, and we would have carried out an unsound computation (still assuming that we are computing on the structure of the natural numbers).

Finally, we illustrate the non-commutativity of ; by considering two permutations of (4). Since there is no negation present, we can again safely disregard the global and local variable registers.

(8) $(y = x + 1 \vee 2 = y) \wedge (x = 2 \vee x = 3) \wedge (2 * x = 3 * y)$.

(9) $(2 * x = 3 * y) \wedge (x = 2 \vee x = 3) \wedge (y = x + 1 \vee 2 = y)$.

$$\begin{array}{rl} & \{\epsilon\}[\![(y = x + 1 \cup 2 = y); (x = 2 \cup x = 3); (2 * x = 3 * y)]\!] \\ = & \{\epsilon\}[\![y = x + 1 \cup 2 = y]\!][\![(x = 2 \cup x = 3); (2 * x = 3 * y)]\!] \\ = & \{\bullet, \{y/2\}\}[\![(x = 2 \cup x = 3); (2 * x = 3 * y)]\!] \\ = & \{\bullet, \{y/2\}\}[\![x = 2 \cup x = 3]\!][\![2 * x = 3 * y]\!] \\ = & \{\bullet, \{x/2, y/2\}, \{x/3, y/2\}\}[\![2 * x = 3 * y]\!] \\ = & \{\bullet, \{x/3, y/2\}\} \end{array}$$

$$
\begin{aligned}
& \{\epsilon\}[\![(2*x=3*y);(x=2\cup x=3);(y=x+1\cup 2=y)]\!] \\
= \; & \{\epsilon\}[\![2*x=3*y]\!][\![(x=2\cup x=3);(y=x+1\cup 2=y)]\!] \\
= \; & \{\bullet\}[\![(x=2\cup x=3);(y=x+1\cup 2=y)]\!] \\
= \; & \{\bullet\}[\![x=2\cup x=3]\!][\![y=x+1\cup 2=y)]\!] \\
= \; & \{\bullet\}[\![y=x+1\cup 2=y)]\!] \\
= \; & \{\bullet\}
\end{aligned}
$$

Looking at how these answers relate, we see that $\{\bullet\}$ is the least informative answer, $\{\bullet, \{y/2, x/3\}\}$ is already more specific, and $\{\{y/2, x/3\}\}$ is fully specified. This notion of informativeness will be made formally precise in the next section.

3 Information Orders on Valuation Spaces

Put a partial order $\sqsubseteq$ on $\mathcal{P}(\mathcal{A} \cup \{\bullet\})$ by means of the following definition:

Definition 20

$$A \sqsubseteq B :\equiv (\bullet \in A \wedge A - \{\bullet\} \subseteq B) \vee (\bullet \notin A \wedge A = B).$$

Then the following holds:

$$\{\bullet\} \sqsubseteq \{\bullet, \{y/2, x/3\}\} \sqsubseteq \{\{y/2, x/3\}\}.$$

It is easily verified that $(\mathcal{P}(\mathcal{A} \cup \{\bullet\}), \sqsubseteq)$ is a complete partial order or CPO. See Davey and Priestley [6]. This ordering turns out to be well-known in computer science where it is referred to as the Egli-Milner order (see [4]), but there is no doubt that it is occasionally reinvented by people engaging in independent reflection on information orderings.[3]

Theorem 21 *$(\mathcal{P}(\mathcal{A} \cup \{\bullet\}), \sqsubseteq)$ is a CPO:*

- *There is a bottom element.*
- *Any increasing sequence has a limit.*

Proof. For any $A \subseteq \mathcal{A} \cup \{\bullet\}$ it holds that $\{\bullet\} \sqsubseteq A$, so $\{\bullet\}$ is the bottom element.

The limit of the increasing sequence $A_0 \sqsubseteq A_1 \sqsubseteq \dots$ is given by:

$$\bigsqcup_{i=0}^{\infty} A_i := \{a \in \mathcal{A} \mid \exists i \in \mathbb{N} : \; a \in A_i\} \cup \{\bullet \mid \forall i \in \mathbb{N} : \; \bullet \in A_i\}.$$

□

Theorem 22 *For all $A, B \subseteq \mathcal{A} \cup \{\bullet\}$:*

1. *$A \sqsubseteq (B \cup \{\bullet\})$ implies $\bullet \in A$.*
2. *$(A - \{\bullet\}) \sqsubseteq B$ implies $(A - \{\bullet\}) \subseteq B$.*

[3]This includes the present author. The fact that this is the Egli-Milner order was brought to my attention by Krzysztof Apt.

3. $\emptyset \sqsubseteq A$ *iff* $A = \emptyset$.

Proof. Three easy checks of the definition. $\square$

For a proper comparison of the executable process interpretation with the standard denotational semantics it is useful to extend the space of full valuations $\mathcal{P}(M^V)$ to a CPO.

Definition 23 *Let* $\bullet'$ *be an object not in* M^V. *Define a partial order* $\sqsubseteq'$ *on* $\mathcal{P}(M^V \cup \{\bullet'\})$ *by means of:*

$$A \sqsubseteq' B :\equiv (\bullet' \in A \wedge A - \{\bullet'\} \subseteq B) \vee (\bullet' \notin A \wedge A = B).$$

One verifies in the same way as above that $(\mathcal{P}(M^V \cup \{\bullet'\}), \sqsubseteq')$ is a CPO. We can now study order preserving maps from one CPO to another.

Definition 24 *Let* $^\circ : \mathcal{P}(\mathcal{A} \cup \{\bullet\}) \to \mathcal{P}(M^V \cup \{\bullet'\})$ *be given by:*

$$A^\circ := \{a \in M^V \mid \exists b \in A \text{ with } b \subseteq a\} \cup \{\bullet' \mid \bullet \in A\}.$$

Note that for all $A \subseteq \mathcal{A}$ we have that $(A \cup \{\bullet\})^\circ = A^\circ \cup \{\bullet'\}$.

Theorem 25 $^\circ$ *is a strict CPO preserving map:* $^\circ$ *preserves* $\sqsubseteq$, $^\circ$ *preserves limits, and moreover,* $^\circ$ *is strict in that it preserves the bottom element.*

Proof. Preservation of $\sqsubseteq$. Assume $A, B \subseteq \mathcal{A} \cup \{\bullet\}$, with $A \sqsubseteq B$. We show that $A^\circ \sqsubseteq' B^\circ$.

Case 1: $\bullet \notin A$. Then $A = B$, and so $A^\circ = B^\circ$, and therefore $A^\circ \sqsubseteq' B^\circ$.

Case 2. $\bullet \in A$. Then $A - \{\bullet\} \subseteq B$, so $(A - \{\bullet\})^\circ \subseteq B^\circ$. Moreover, since $\bullet \in A$ we have $\bullet' \in A^\circ$. It follows that $A^\circ \sqsubseteq B^\circ$.

Let $A_0 \sqsubseteq A_1 \sqsubseteq \ldots$ be an ordered sequence in $\mathcal{P}(\mathcal{A} \cup \{\bullet\})$. Then by the fact that $^\circ$ preserves the order, $A_0^\circ \sqsubseteq' A_1^\circ \sqsubseteq' \ldots$ is an ordered sequence in $\mathcal{P}(M^V) \cup \{\bullet'\}$. We have to show that $(\bigsqcup_{i=1}^\infty A_i)^\circ = \bigsqcup_{i=1}^\infty A_i^\circ$.

Case 1: Assume $\forall i \in \mathbb{N}\ \bullet \in A_i$. Then:

$$(\bigsqcup_{i=1}^\infty A_i)^\circ = (\bigcup_{i=1}^\infty A_i)^\circ = \bigcup_{i=1}^\infty A_i^\circ = \bigsqcup_{i=1}^\infty A_i^\circ.$$

Case 2: Assume $\neg\forall i \in \mathbb{N}\ \bullet \in A_i$, and let j be the first index for which $\bullet \notin A_j$. Then, by the definition of $\sqsubseteq$ for $\mathcal{P}(\mathcal{A} \cup \{\bullet\})$, $A_j = A_{j+1} = \ldots$, and $\bigsqcup_{i=1}^\infty A_i = A_j$, so $(\bigsqcup_{i=1}^\infty A_i)^\circ = A_j^\circ$. By the definition of limits for $\mathcal{P}(M^V \cup \{\bullet'\})$, $\bigsqcup_{i=1}^\infty A_i^\circ = A_j^\circ$, and we are done.

Finally, $\{\bullet\}^\circ = \{\bullet'\}$, so $^\circ$ is strict. $\square$

A computation procedure F is *faithful* to DPL (i.e., to the function $[\![\cdot]\!]$) if for any formula ϕ and any input valuation $a \in M^V$, $F_\phi(a) \sqsubseteq' [\![\phi]\!](a)$.

A computation procedure F is *complete* for DPL if for any formula ϕ and any input valuation $a \in M^V$, $F_\phi(a) = [\![(\phi)]\!](a)$. Since DPL has the same expressive power as standard predicate logic, we know that no computation procedure can be complete for DPL. Still, the ordering on $\mathcal{P}(M^V \cup \{\bullet'\})$ makes it possible to make *comparative judgments* of computational power. We say that a computation procedure G extends a computation procedure F if for all ϕ in the language, all inputs $a \in M^V$, $F_\phi(a) \sqsubseteq' G_\phi(a)$. We say that a computation procedure G is a proper extension of a computation procedure F if G extends F and moreover, there are ϕ, a with $F_\phi(a) \neq G_\phi(a)$. These are the cases where $F_\phi(a)$ introduces uncertainties that $G_\phi(a)$ avoids. Of course, if we know that G extends F, and we also know that G is faithful to DPL then it follows from this that F is faithful to DPL.

This perspective on computational procedures still contains one idealization that has to be removed. Real-life computations always have finite inputs. In realistic comparative judgments on computational performance this is an essential feature, for a computation may result in an error due to lack of input data, and we would like to take such errors into account when judging the performance of the procedure.

For a solution we use the CPO preserving map $^\circ$ to look at all extensions of a given finite input. We say that a computation procedure in $\mathcal{A} \cup \{\bullet\} \to \mathcal{P}(\mathcal{A} \cup \{\bullet\})$ is faithful to DPL if for any input a, the set of all full extensions of the output valuations is an approximation (along the ordering $\sqsubseteq'$) to the output of the DPL interpretation function of the set of all full extensions of the input valuation.

To formalize this, it is convenient to reformulate the dynamic interpretation function as a function

$$[\![(\cdot)]\!] : \mathcal{P}(M^V \cup \{\bullet'\}) \to \mathcal{P}(M^V \cup \{\bullet'\}),$$

by means of the following definition.

Definition 26

$$\begin{aligned} [\![\phi]\!](\bullet') &:= \{\bullet'\}, \\ [\![(\phi)]\!](A) &:= \bigcup_{a \in A} \{[\![\phi]\!](a)\}. \end{aligned}$$

Observe that for all formulas ϕ of the language, all input states $A \subseteq M^V$, we have that $\bullet' \notin [\![(\phi)]\!](A)$. For any ϕ, the function $[\![(\phi)]\!]$ is distributive (the output for A is the union of the outputs of the members of A), it is monotonic for the ordering $\sqsubseteq'$ (it does not create any uncertainties except for those due to the presence of $\bullet'$ in the input), and it is strict (it maps $\{\bullet'\}$ to $\{\bullet'\}$).

We can now say that a computation procedure $F : \mathcal{A} \cup \{\bullet\} \to \mathcal{P}(\mathcal{A} \cup \{\bullet\})$ is faithful to DPL if for all inputs $a \in \mathcal{A}$, all formulas ϕ, $(F_\phi(a))^\circ \sqsubseteq' [\![(\phi)]\!](\{a\}^\circ)$.

4 Executable Process Interpretation Faithful to DPL

This section contains the main theorem of the paper, validating our definition of the executable process interpretation for first order logic by relating it to the standard DPL semantics. Informally, if for a given input valuation a, you extend a proper output valuation b for ϕ under the executable semantics to a full valuation b', then there is a full extension a' of a for which b' is in the output of the classical dynamic semantics for ϕ for input a'. Also, if the executable process interpretation does give no output for ϕ on input a, then the classical dynamic semantics will give no output either. Thus, if there are no solutions according to the executable process interpretation, there are no solutions according to the standard semantics either. What this boils down to is that the executable process interpretation gives the right answers within its more limited domain of application.

We need some preliminary lemmas.

Lemma 27 (Extension Lemma) *If* $(a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b)$ *then*

- $g^a \subseteq g^b$.

- $l^a \subseteq l^b$.
- $dom(a) \cup l^a \subseteq dom(b) \cup l^b$.

Proof. Induction on the structure of ϕ. □

Lemma 28 (Fill-in Lemma) *For all formulas ϕ, all $\boldsymbol{a}, \boldsymbol{b}$ with $v \in l^a - dom(a)$, the following are equivalent:*

1. $\boldsymbol{a} \xrightarrow{\phi} \boldsymbol{b}$,
2. *there is an object d in the domain of the model with $(a \cup \{v/d\}, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b)$.*

Proof. Induction on the structure of ϕ. □

To state the next lemma, we need an ordering on $\mathcal{P}(\mathcal{A} \times \mathcal{P}(V) \times \mathcal{P}(V))$. We define this by means of:

Definition 29

$$\boldsymbol{b} \preceq \boldsymbol{c} :\equiv b \subseteq c, g^b = g^c, l^b = l^c.$$

Lemma 30 (Conditional Forward Property Lemma) *Suppose $\boldsymbol{b} = (b, g^b, l^b) \in [\![\{\phi\}]\!](\boldsymbol{a})$ with $\boldsymbol{a} = (a, g^a, l^a)$ and $l^a \cup g^b \subseteq dom(a)$. Then $\boldsymbol{a} \preceq \boldsymbol{a}'$ implies that there is a $\boldsymbol{b}' \succeq \boldsymbol{b}$ with $\boldsymbol{b}' \in [\![\{\phi\}]\!](\boldsymbol{a}')$.*

In a schema, with $\boldsymbol{a} \xrightarrow{\phi} \boldsymbol{b}$ for $\boldsymbol{b} \in [\![\{\phi\}]\!](\boldsymbol{a})$:

$$\begin{array}{ccc}
\boldsymbol{a} & \xrightarrow{\preceq} & \boldsymbol{a}' \\
\downarrow{\scriptstyle \phi} & & \downarrow{\scriptstyle \phi} \\
\boldsymbol{b} & \xrightarrow{\preceq} & \boldsymbol{b}'
\end{array}$$

Proof. Induction on the structure of ϕ, under an ordering of the formulas where $\phi_1; (\phi_2; \phi_3)$ counts as less complex than $(\phi_1; \phi_2); \phi_3$.

Suppose ϕ is a test, i.e., ϕ is a formula with the property that $\mathbf{a} \xrightarrow{\phi} \mathbf{b}$ and $\mathbf{b} \neq \bullet$ together imply that $\mathbf{b} = \mathbf{a}$. Suppose $\mathbf{a} \xrightarrow{\phi} \mathbf{b}$. Then $\mathbf{b} = \mathbf{a} = (a, g^a, l^a)$, and for all $a' \supseteq a$ it holds that $(a', g^a, l^a) \xrightarrow{\phi} (a', g^a, l^a)$, so the claim holds for this case.

This takes care of the cases $\bot$, $Pt_1 \cdots t_n$, and $\neg\phi$.

$t_1 \doteq t_2$: Suppose $\mathbf{a} \xrightarrow{t_1 \doteq t_2} \mathbf{b}$. Assume $t_1 \doteq t_2$ is an a-assignment. Then $t_1 \doteq t_2$ is not a-closed, and one of t_1, t_2 equals a variable $v \notin dom(a)$. Assume $v \in l^a$. Then contradiction with $l^a \subseteq dom(a)$. Assume $v \notin l^a$. Then, according to the executable interpretation of a-assignments, we have that $\mathbf{b} = (a \cup \{v/t_i^a\}, g^a \cup \{v\}, l^a)$, for $i = 1, 2$, as the case may be. This gives a contradiction with the condition that $g^b \subseteq dom(a)$. So $t_1 \doteq t_2$ cannot be an a-assignment. But then it must be a test, and the claim holds in this case by the reasoning for tests.

$\exists v$: Suppose $\mathbf{a} \xrightarrow{\exists v} \mathbf{b}$. First case: $v \notin dom(a)$. Then $\mathbf{b} = (a, g^a, l^a \cup \{v\})$. Let $a' \supseteq a$. In case $v \in dom(a')$, $(a', g^a, l^a) \xrightarrow{\exists v} (a' - \{v/v^{a'}\}, g^a, l^a \cup \{v\})$, and the claim holds since $a' - \{v/v^{a'}\} \supseteq a$. In case $v \notin dom(a')$, $(a', g^a, l^a) \xrightarrow{\exists v} (a', g^a, l^a \cup \{v\})$, and the claim holds since $a' \supseteq a$. Second case: Suppose $\mathbf{a} \xrightarrow{\exists v} \mathbf{b}$ and $v \in dom(a)$. Then $\mathbf{b} = (a - \{v/v^a\}, g^a, l^a \cup \{v\})$. Let $a' \supseteq a$. Then:

$$(a', g^a, l^a) \xrightarrow{\exists v} (a' - \{v/v^a\}, g^a, l^a \cup \{v\}).$$

Now the claim holds since

$$(a' - \{v/v^a\}) \supseteq (a - \{v/v^a\}).$$

$\phi_1 \cup \phi_2$: Suppose that $\mathbf{a} \xrightarrow{\phi_1 \cup \phi_2} \mathbf{b}$ and that the condition $l^a \cup g^b \subseteq dom(a)$ holds. Then either $\mathbf{a} \xrightarrow{\phi_1} \mathbf{b}$ or $\mathbf{a} \xrightarrow{\phi_2} \mathbf{b}$. Without loss of generality, assume $\mathbf{a} \xrightarrow{\phi_1} \mathbf{b}$. We can apply the induction hypothesis and we get that for every $a' \supseteq a$ there is a $b' \supseteq b$ with $(a', g^a, l^a) \xrightarrow{\phi_1} (b', g^b, l^b)$. This proves that for every $\mathbf{a}' \succeq \mathbf{a}$ there is a $\mathbf{b}' \succeq \mathbf{b}$ with $\mathbf{a}' \xrightarrow{\phi_1 \cup \phi_2} \mathbf{b}'$.

$(\phi_1; \phi_2); \phi_3$: Observe that (by Theorem 17) $\mathbf{a} \xrightarrow{(\phi_1;\phi_2);\phi_3} \mathbf{b}$ iff $\mathbf{a} \xrightarrow{\phi_1;(\phi_2;\phi_3)} \mathbf{b}$, and apply the induction hypothesis.

The only cases we still have to deal with are sequential compositions of the form $\phi_1; \phi_2$ with ϕ_1 an atomic formula, a negation, or a union. The cases where ϕ_1 is of the form $\perp$, $Pt_1 \cdots t_n$, $\neg\psi$, $\psi_1 \cup \psi_2$ are straightforward. This leaves us with the cases $t_1 \doteq t_2; \phi$ and $\exists v; \phi$ to clinch the argument.

$t_1 \doteq t_2; \phi$: Suppose that $\mathbf{a} \xrightarrow{t_1 \doteq t_2; \phi} \mathbf{b}$ and that $l^a \cup g^b \subseteq dom(a)$. Assume that $t_1 \doteq t_2$ is an a-assignment for variable v. In case $v \in l^a$ we get a contradiction with $l^a \subseteq dom(a)$, so $v \notin l^a$. Then, according to the definition of the executable interpretation of sequential composition, we must have (for $i = 1$ or 2, as the case may be):

$$(a, g^a, l^a) \xrightarrow{t_1 \doteq t_2} (a \cup \{v/t_i^a\}, g^a \cup \{v\}, l^a) \xrightarrow{\phi} (b, g^b, l^b).$$

It follows that in this case $g^b \supseteq g^a \cup \{v\}$, by Lemma 27. By the condition $g^b \subseteq dom(a)$ we get that $v \in dom(a)$, and contradiction with the fact that $v^a =\uparrow$. This shows that $t_1 \doteq t_2$ must be a test. But then it follows that for all $a' \supseteq a$ we have that $(a', g^a, l^a) \xrightarrow{t_1 \doteq t_2} (a', g^a, l^a)$, and we can use the induction hypothesis to get that there is a $\mathbf{b}' \succeq \mathbf{b}$ with $(a', g^a, l^a) \xrightarrow{\phi} \mathbf{b}'$.

$\exists v; \phi$: Assume that $\mathbf{a} \xrightarrow{\exists v; \phi} \mathbf{b}$ and $l^a \cup g^b \subseteq dom(a)$. First case: $v \notin dom(a)$. Then:

$$\mathbf{a} \xrightarrow{\exists v} (a, g^a, l^a \cup \{v\}) \xrightarrow{\phi} (b, g^b, l^b).$$

Let $a' \supseteq a$, and assume $v \in dom(a')$. Then $(a', g^a, l^a) \xrightarrow{\exists v} (a' - \{v/v^{a'}\}, g^a, l^a \cup \{v\})$. By Lemma 28, it follows from $(a, g^a, l^a \cup \{v\}) \xrightarrow{\phi} (b, g^b, l^b)$ that there is a d in the domain of the model with $(a \cup \{v/d\}, g^a, l^a \cup \{v\}) \xrightarrow{\phi} (b, g^b, l^b)$. From $l^a \cup g^b \subseteq dom(a)$ it follows that $l^a \cup \{v\} \cup g^b \subseteq dom(a \cup \{v/d\})$. Thus, we get by induction hypothesis that there is a $\mathbf{b}' \succeq \mathbf{b}$ with $(a'[v := d], g^a, l^a \cup \{v\}) \xrightarrow{\phi} \mathbf{b}'$. By another application of Lemma 28, we get that $(a' - \{v/v^{a'}\}, g^a, l^a \cup \{v\} \xrightarrow{\phi} \mathbf{b}'$. This proves that there is a $\mathbf{b}' \succeq \mathbf{b}$ with $\mathbf{a}' \xrightarrow{\exists v; \phi} \mathbf{b}'$.

Now let $a' \supseteq a$, and assume $v \notin dom(a')$. Then $(a', g^a, l^a) \xrightarrow{\exists v} (a', g^a, l^a \cup \{v\})$. By Lemma 28, it follows from $(a, g^a, l^a \cup \{v\}) \xrightarrow{\phi} (b, g^b, l^b)$ that there is a d in the domain of the model with $(a \cup \{v/d\}, g^a, l^a \cup \{v\}) \xrightarrow{\phi} (b, g^b, l^b)$. From $l^a \cup g^b \subseteq dom(a)$ it follows that $l^a \cup \{v\} \cup g^b \subseteq dom(a \cup \{v/d\})$. Thus, we get by induction hypothesis that there is a $\mathbf{b}' \succeq \mathbf{b}$ with $(a' \cup \{v/d\}, g^a, l^a \cup \{v\}) \xrightarrow{\phi} \mathbf{b}'$. By another application of Lemma 28, we get that $(a', g^a, l^a \cup \{v\} \xrightarrow{\phi} \mathbf{b}'$. This proves that there is a $\mathbf{b}' \succeq \mathbf{b}$ with $\mathbf{a}' \xrightarrow{\exists v; \phi} \mathbf{b}'$.

Second case: $v \in dom(a)$. Reasoning is similar. □

What the conditional forward property lemma says is that if a computation starting from input $\mathbf{a}$ yields an output $\mathbf{b}$, then extending $\mathbf{a}$ will preserve an extension of output $\mathbf{b}$, on condition that the computation process from **a** to **b** did not involve new assignments of values to global variables, and moreover the local variables that possibly get reassigned will not be affected by an extension of the input because they are all in the domain of the input valuation. Of course, the extension of **a** may have the effect that more outputs get computed besides (the extension of) **b**, but that is another matter.

The conditional forward property lemma is similar to the forward property lemma proved in Visser [13] for a different variation of Dynamic Predicate Logic. In Visser's case the property holds without further ado, but for the present set-up a special condition has to be imposed to make it hold. Note that the condition on the Lemma states that **b** is safe for **a**. The Lemma is crucial for the proof of the negation case in the following adequacy theorem. For the meaning of $A^{\natural\circ}$ in the statement of the theorem, compare definitions 19, 24.

Theorem 31 (Executable Interpretation Faithful to DPL) *For every formula ϕ, every $\boldsymbol{a} \in (\mathcal{A} \times \mathcal{P}V \times \mathcal{P}V) \cup \{\bullet\}$:*

$$(\{\!\![\phi]\!\!\}(\boldsymbol{a}))^{\natural\circ} \sqsubseteq' [\![(\phi)]\!](\{a\})^{\circ}.$$

Proof. Induction on the structure of ϕ. Except in the case of negation, we can disregard the global and local variable stores. We will leave them out of the computations unless we need them.

$\perp$: The claim is trivially true.

$Pt_1 \cdots t_n$: If $Pt_1 \cdots t_n$ not a-closed, then $(\{\!\![Pt_1 \cdots t_n]\!\!\}(\mathbf{a}))^{\natural} = \{\bullet\}$, so $(\{\!\![Pt_1 \cdots t_n]\!\!\}(\mathbf{a}))^{\natural\circ} = \{\bullet'\}$, and the claim holds.

If $Pt_1 \cdots t_n$ is a-closed and true, $\{\!\![Pt_1 \cdots t_n]\!\!\}(\mathbf{a}))^{\natural} = \{a\}$, and $(\{\!\![Pt_1 \cdots t_n]\!\!\}(\mathbf{a}))^{\natural\circ} = \{a' \in M^V \mid a \subseteq a'\}$, and the claim holds by the fact that

$$[\![(Pt_1 \cdots t_n)]\!](\{a' \in M^V \mid a \subseteq a'\}) = \{a' \in M^V \mid a \subseteq a'\},$$

since the truth of $Pt_1 \cdots t_n$ depends only on the values assigned by a.

If $Pt_1 \cdots t_n$ is a-closed and false, $\{\!\![Pt_1 \cdots t_n]\!\!\}(\mathbf{a}) = \emptyset$, and $(\{\!\![Pt_1 \cdots t_n]\!\!\}(\mathbf{a}))^{\natural\circ} = \emptyset$, and the claim holds by the fact that

$$[\![(Pt_1 \cdots t_n)]\!](\{a' \in M^V \mid a \subseteq a'\}) = \emptyset,$$

since the truth of $Pt_1 \cdots t_n$ depends only on the values assigned by a.

$t_1 \doteq t_2$: If $t_1 \doteq t_2$ is not a-closed nor an a-assignment, then $(\{\!\![t_1 \doteq t_2]\!\!\}(\mathbf{a}))^{\natural} = \{\bullet\}$, so $(\{\!\![t_1 \doteq t_2]\!\!\}(\mathbf{a}))^{\natural\circ} = \{\bullet'\}$, and the claim holds.

If $t_1 \doteq t_2$ is an a-assignment, say with $t_1 \equiv v$, $t_1^a = \uparrow$ (the symmetric case is analogous), and $v \in l^a$, then

$$(\{\!\![t_1 \doteq t_2]\!\!\}(\mathbf{a}))^{\natural} = \{a \cup \{v/t_2^a\}\}.$$

Using $\dot{a}$ for valuation $a \cup \{v/t_2^a\}$, we see that $(\{\!\![t_1 \doteq t_2 \phi]\!\!\}(\mathbf{a}))^{\natural\circ} = \{a' \in M^V \mid \dot{a} \subseteq a'\}$ and the claim holds by the fact that

$$\{a' \in M^V \mid \dot{a} \subseteq a'\} = [\![(t_1 \doteq t_2)]\!](\{a' \in M^V \mid a \subseteq a'\}).$$

The case where $v \notin l^a$ is similar.

If $t_1 \doteq t_2$ is a-closed then reason as in the case of $Pt_1 \cdots t_n$ a-closed.

$\exists v$: If $v \notin dom(a)$ then $(\{\!\![\exists v]\!\!\}(\mathbf{a}))^{\natural} = \{a\}$. Then if $b, b' \in M^V$ with $b \sim_v b'$ then $b \in \{a\}^{\circ}$ iff $b' \in \{a\}^{\circ}$, and therefore

$$\{a\}^{\circ} = [\![(\exists v]\!](\{a\}^{\circ}),$$

and the claim holds.

If $v \in dom(a)$ then $([\![\exists v]\!](\mathbf{a}))^{\natural} = \{a - \{v/v^a\}\}$. Using $\dot{a}$ for $a - \{v/v^a\}$ we see that

$$\{\dot{a}\}^{\circ} = \{b \in M^V \mid \exists a' \in \{a\}^{\circ} \text{ with } b \sim_b a'\}.$$

This proves the claim for this case.

$\neg\phi$: If $[\![\phi]\!](\mathbf{a})$ is risky for $\mathbf{a}$ then $[\![\neg\phi]\!](\mathbf{a}) = \{\bullet\}$, and the claim holds in virtue of the fact that $([\![\neg\phi]\!](\mathbf{a}))^{\natural\circ} = \{\bullet'\}$.

If $[\![\phi]\!](\mathbf{a}) = \emptyset$, then $([\![\neg\phi]\!](\mathbf{a}))^{\natural} = \{a\}$. By the induction hypothesis, $[\![(\phi)]\!](\{a\}^{\circ}) = \emptyset$, so:

$$([\![\neg\phi]\!](\mathbf{a}))^{\natural\circ} = \{a\}^{\circ} = [\![(\neg\phi)]\!](\{a\}^{\circ}) = \{a\}^{\circ}.$$

If $[\![\phi]\!](\mathbf{a})$ is not risky, then it contains at least one triple (b, g^b, l^b) with the property that $l^a \cup g^b \subseteq dom(a)$. Then the condition for the forward property lemma is fulfilled, so the forward property holds for $(a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b)$. This gives us that for any $a' \supseteq a$ there is a $b' \subseteq b$ with $(a', g^a, l^a) \xrightarrow{\phi} (b', g^b, l^b)$. By the induction hypothesis, $[\![(\phi)]\!](\{a'\}^{\circ}) \neq \emptyset$. Since we may take a' to be any full extension of a, this means that $[\![(\phi)]\!](\{a\}^{\circ}) \neq \emptyset$. This shows that $[\![(\neg\phi)]\!](\{a\}^{\circ}) = \emptyset$, and we are done.

$\phi_1;\phi_2$: Let $\mathbf{B} = [\![\phi_1;\phi_2]\!](\{\mathbf{a}\}$. Suppose $\bullet \in \mathbf{B}$, i.e., $\mathbf{a} \xrightarrow{\phi_1;\phi_2} \bullet$. Then either $\mathbf{a} \xrightarrow{\phi_1} \bullet$ or there is a $\mathbf{b} \neq \bullet$ with $\mathbf{a} \xrightarrow{\phi_1} \mathbf{b} \xrightarrow{\phi_2} \bullet$.

Case 1: $\bullet \in [\![\phi_1]\!](\mathbf{a})$. Then by induction hypothesis $([\![\phi_1]\!](\mathbf{a}))^{\natural\circ} \sqsubseteq' [\![(\phi_1)]\!](\{a\})^{\circ}$, so $([\![\phi_1]\!](\mathbf{a}))^{\natural\circ} - \{\bullet'\} \subseteq [\![(\phi_1)]\!](\{a\})^{\circ}$. Moreover, for all proper $\mathbf{b} \in [\![\phi_1]\!](\mathbf{a})$, we have, again by induction hypothesis, that $([\![\phi_2]\!](\mathbf{b}))^{\natural\circ} \sqsubseteq' [\![(\phi_2)]\!](\{b\})^{\circ}$. This gives $([\![\phi_2]\!](\mathbf{b}))^{\natural\circ} - \{\bullet'\} \subseteq [\![(\phi_2)]\!](\{b\})^{\circ}$, and therefore $\mathbf{B}^{\natural\circ} - \{\bullet'\} \subseteq [\![(\phi_1;\phi_2)]\!](\{a\})^{\circ}$. Since we know $\bullet' \in B^{\natural\circ}$, this proves $\mathbf{B}^{\natural\circ} \sqsubseteq' [\![(\phi_1;\phi_2)]\!](\{a\})^{\circ}$.

Case 2: $\bullet \notin [\![\phi_1]\!](\mathbf{a})$, $\bullet \in [\![\phi_1]\!](\mathbf{b})$ for some $\mathbf{b} \in [\![\phi_1]\!](\mathbf{b})$. Then by induction hypothesis, $([\![\phi_1]\!](\mathbf{a}))^{\natural\circ} = [\![(\phi_1)]\!](\{a\})^{\circ}$, and, again by induction hypothesis, for all $\mathbf{b} \in [\![\phi_1]\!](\mathbf{a})$, $([\![\phi_2]\!](\mathbf{b}))^{\natural\circ} - \{\bullet'\} \subseteq [\![(\phi_2)]\!](\{b\})^{\circ}$. This shows $\mathbf{B}^{\natural\circ} \subseteq [\![(\phi_1;\phi_2)]\!](\{a\})^{\circ}$, and, since we know $\bullet' \in \mathbf{B}^{\natural\circ}$, also $\mathbf{B}^{\natural\circ} \sqsubseteq' [\![(\phi_1;\phi_2)]\!](\{a\})^{\circ}$.

Now suppose $[\![\phi_1;\phi_2]\!](\mathbf{a}) = \mathbf{B}$ and assume $\bullet \notin B$. Then we have by induction hypothesis that $([\![\phi_1]\!](\mathbf{a}))^{\natural\circ} = [\![(\phi_1)]\!](\{a\}^{\circ})$, and for all $\mathbf{b} \in ([\![\phi_1]\!](\mathbf{a})$, again by induction hypothesis, that $([\![\phi_2]\!](\mathbf{b}))^{\natural\circ} = [\![(\phi_2)]\!](\{b\}^{\circ})$. This shows $\mathbf{B}^{\natural\circ} = [\![(\phi_1;\phi_2)]\!](\{a\}^{\circ})$.

$\phi_1 \cup \phi_2$: $[\![\phi_1 \cup \phi_2]\!](\mathbf{a}) = [\![\phi_1]\!](\mathbf{a}) \cup [\![\phi_2]\!](\mathbf{a})$. Since ${}^{\natural\circ}$ commutes with $\cup$, this gives:

$$([\![\phi_1 \cup \phi_2]\!](\mathbf{a}))^{\natural\circ} = ([\![\phi_1]\!](\mathbf{a}))^{\natural\circ} \cup ([\![\phi_2]\!](\mathbf{a}))^{\natural\circ}.$$

By induction hypothesis, twice, $[\![\phi_1]\!](\mathbf{a}))^{\natural\circ} \sqsubseteq' [\![(\phi_1)]\!](\{a\}^{\circ})$, and $[\![\phi_2]\!](\mathbf{a}))^{\natural\circ} \sqsubseteq' [\![(\phi_2)]\!](\{a\}^{\circ})$. Since $\sqsubseteq'$ is preserved under $\cup$, this gives:

$$[\![\phi_1]\!](\mathbf{a}))^{\natural\circ} \cup [\![\phi_2]\!](\mathbf{a}))^{\natural\circ} \sqsubseteq' [\![(\phi_1)]\!](\{a\}^{\circ}) \cup [\![(\phi_2)]\!](\{a\}^{\circ}).$$

It follows that $[\![\phi_1 \cup \phi_2]\!](\mathbf{a}))^{\natural\circ} \sqsubseteq' [\![(\phi_1 \cup \phi_2)]\!](\{a\}^{\circ}$. □

5 Dynamic Logic Programming

The examples so far were meant to demonstrate the executable interpretation process, not to convince the reader of the potential of Dynamic Logic Programming as a new style that combines the best features of imperative programming and logic programming. In this section we address this further issue.

Assignment Without Sting A fundamental feature of imperative programming is the destructive assignment statement $x := t$. At first sight it may seem that this cannot be expressed in dynamic logic programming. In a variable environment a where an identity $x = t$ is interpreted as an assignment of a value to x it is essential, as we have seen, that x does not occur in t. One side of the equality statement has to be a-closed. Thus, the identity $x = x + 1$ either gives an error message (in cases where the input valuation is not defined for x) or it fails (because, interpreted in the domain of natural numbers at least, there is no n with $n = n + 1$).

It turns out, however, that we can express the command to increment x by one, if we allow ourselves the use of an auxiliary variable y. The command $x := x + 1$ then becomes:

(10) $\exists y; y = x + 1; \exists x; x = y.$

The presence of existential quantification inside the programming language has allowed us to 'internalize' the well-known correctness assertion for assignment statements that originates with Floyd [10] (see also De Bakker [4] and Exercise 3.6 of [3]):

$$\{\phi\}\ x := t\ \{\exists y(\phi[y/x] \wedge x = t[y/x])\}.$$

The interest of this new way of expressing assignment lies in the fact that it has lost its destructive sting. Just how tame this new version of assignment is transpires if we look again at the reversibility feature of DPL formulas that was demonstrated in Theorem 7. For reasons of symmetry, we append an $\exists y$ at the end of formula (10), and then reverse it:

$$(\exists y; y = x + 1; \exists x; x = y; \exists y)^r \text{ equals } \exists y; x = y; \exists x; y = x + 1; \exists y.$$

This new program needs one further touch to transform it into a program that decrements x by one (or fails, if x happens to be equal to 0, provided we stick to our assumption that we are interpreting in the natural numbers, that is). We have not reversed the identity statement $y = x + 1$, to make it into an assignment for x. Of course, an identity $y = t(x)$ can only be translated into a assignment for x if t expresses an injective function f, but if it is we can translate $y = t(x)$ into $t'(y) = x$, with t' an expression for f^{-1}. In the present case this is indeed possible, and we get:

$$\exists y; x = y; \exists x; y - 1 = x; \exists y.$$

This program is as close as you can get to a reverse of (10). Here is a program to swap the values of x and y, using an auxiliary variable z.

$$\exists z; x = z; \exists x; x = y; \exists y; y = z.$$

Again, this program can be reversed. Put an $\exists z$ at the end first to make sure that z is not dynamically free in the reversed program.

$$(\exists z; x = z; \exists x; x = y; \exists y; y = z; \exists z)^r \text{ equals } \exists z; y = z; \exists y; x = y; \exists x; x = z; \exists z.$$

Since the swap operation is symmetric, the reversed formula performs the same action as the original swap formula.

In dynamic logic programming, destructive assignment is replaced by safe assignment. The transition is made virtually painless by the following abbreviation:

$$\text{Write } \exists j; j = i; \exists i; i = t \text{ as:} \quad i \blacktriangleleft_j t.$$

Then the destructive assignment statement $x := x + 1$ can be replaced by the safe assignment statement $x \blacktriangleleft_{x'} x' + 1$. The general procedure for removing the sting from assignment is the following translation instruction:

$$(x := t)^{\heartsuit} \quad := \quad \begin{cases} \exists x; x = t & \text{if } x \text{ does not occur in } t, \\ x \blacktriangleleft_{x'} t[x'/x] & \text{otherwise.} \end{cases}$$

Choice The deterministic choice command of imperative programming, IF THEN ELSE, is easily expressed. Here is the rendering of 'IF $y < 0$ THEN $y := -y$ ELSE SKIP':

$$(y < 0; y \blacktriangleleft_{y'} -y') \cup (y \geq 0).$$

Here is a program that checks whether z is the largest of x and y:

$$(x \geq y; z = x) \cup (x < y; z = y).$$

To make this into a program that assigns the maximum of x and y to z, just make z local by means of an existential quantifier:

$$\exists z; ((x \geq y; z = x) \cup (x < y; z = y)).$$

Non-deterministic choice is also expressed easily by means of $\cup$.

Bounded Iteration Consider the imperative program that calculates $n!$:

$$j := 1;\ \text{FOR } i := 1 \text{ TO } n \text{ DO } j := j * i \text{ END}.$$

Here is the corresponding dynamic logic program:

(11) $n = n; \exists j; j = 1; \exists i; i = 1; j \blacktriangleleft_{j'} j' * i; \ldots ; \exists i; i = n; j \blacktriangleleft_{j'} j' * i.$

Program (11) computes just like the imperative program, but it is completely declarative. A nice extra touch is that the dynamic logic program checks whether n is instantiated at the start of the iterative process, by means of the test $n = n$. This test succeeds without further ado in case the initial valuation is defined for n, and generates $\bullet$ otherwise. A flaw of (11) is that the number of iterations is fixed at compile time.

If we want bounded iteration with the number of iterations determined only at run-time, we need to be specific about variable sorts and introduce a sort for integer expressions:

$$\begin{array}{lll} N & ::= & 0 \mid 1 \mid \cdots \mid v_i \mid -N \mid (N_1 + N_2) \mid (N_1 - N_2) \mid (N_1 * N_2) \\ \phi & ::= & \cdots \mid \phi^n \mid \phi^N. \end{array}$$

Dynamic interpretation in model $\mathcal{M} = (M, I)$:

$$\begin{array}{lll} [\![\phi^0]\!](a) & := & \{a\} \\ [\![\phi^{n+1}]\!](a) & := & [\![\phi; \phi^n]\!](a) \\ [\![\phi^N]\!](a) & := & [\![\phi^{\max(0,N^a)}]\!](a) \end{array}$$

Executable process interpretation in model $\mathcal{M} = (M, I)$:

$$\begin{array}{lll} [\{\phi^0\}](\mathbf{a}) & := & \{\mathbf{a}\} \\ [\{\phi^{n+1}\}](\mathbf{a}) & := & [\{\phi; \phi^n\}](\mathbf{a}) \\ [\{\phi^N\}](\mathbf{a}) & := & \begin{cases} [\{\phi^{\max(0,N^a)}\}](\mathbf{a}) & \text{if } N^a = \downarrow, \\ \{\bullet\} & \text{if } N^a = \uparrow \end{cases} \end{array}$$

The transition system format for bounded iteration is given in Figure 3. We can now reformulate the factorial program as follows:

(12) $\exists j; j = 1; \exists i; i = 0; (i \blacktriangleleft_{i'} i' + 1; j \blacktriangleleft_{j'} j' * i)^N.$

In this new version, N is considered as a program term, with its value determined by the input valuation. This ensures that the number of iterations is determined at run-time.

Figure 3: Transitions for Bounded Iteration

ϕ^n	$\dfrac{}{(a, g^a, l^a) \xrightarrow{\phi^0} (a, g^a, l^a)}$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi} \bullet}{(a, g^a, l^a) \xrightarrow{\phi^{n+1}} \bullet}$ $\quad$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b) \xrightarrow{\phi^n} \bullet}{(a, g^a, l^a) \xrightarrow{\phi^{n+1}} \bullet}$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b) \xrightarrow{\phi^n} (c, g^c, l^c)}{(a, g^a, l^a) \xrightarrow{\phi^{n+1}} (c, g^c, l^c)}$
ϕ^N	$\dfrac{}{(a, g^a, l^a) \xrightarrow{\phi^N} \bullet}\ N^a = \uparrow$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi^n} \bullet}{(a, g^a, l^a) \xrightarrow{\phi^N} \bullet}\ N^a = \downarrow, \max(0, N^a) = n$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi^n} (b, g^b, l^b)}{(a, g^a, l^a) \xrightarrow{\phi^N} (b, g^b, l^b)}\ N^a = \downarrow, \max(0, N^a) = n$

Bounded Choice Suppose we want to check whether program ϕ succeeds for any i in a bounded range of possibilities, say $1..n$. This check can be expressed as (13).

(13) $(i = 1; \phi) \cup \cdots \cup (i = n); \phi)$.

Program (13) has the same flaw as (11): the range of choice is fixed at compile time. To express bounded choice, with the range of choice determined only at run-time, we need a language construct with an explicit integer variable to control the range of choice. This suggests another small extension of the language with the following construction for bounded choice:

$$\phi \quad ::= \quad \cdots \mid \bigcup^{v_i}_{M..N} \phi.$$

Dynamic interpretation of the bounded choice construction in model $\mathcal{M} = (M, I)$:

$$[\![\bigcup^{v_i}_{M..N} \phi]\!](a) \quad := \quad \begin{cases} [\![(v_i = M; \phi]\!](a) \cup \cdots \cup [\![(v_i = N; \phi]\!](a) & \text{if } M^a \leq N^a \\ \emptyset & \text{otherwise.} \end{cases}$$

Executable process interpretation the bounded choice construction in model $\mathcal{M} = (M, I)$:

$$[\{\bigcup^{v_i}_{M..N} \phi\}](\mathbf{a}) \quad := \quad \begin{cases} [\{v_i = M; \phi\}](\mathbf{a}) \cup \cdots \cup [\{v_i = N; \phi\}](\mathbf{a}) \\ \qquad \text{if } M^a = \downarrow, N^a = \downarrow, M^a \leq N^a, \\ \emptyset \quad \text{if } M^a = \downarrow, N^a = \downarrow, M^a > N^a, \\ \{\bullet\} \quad \text{if } M^a = \uparrow \text{ or } N^a = \uparrow. \end{cases}$$

Figure 4: Transitions for Bounded Choice

$\bigcup^{v_i}_{M..N} \phi$	$\dfrac{}{(a, g^a, l^a) \xrightarrow{\cup^{v_i}_{M..N}\phi} \bullet}\ M^a = \uparrow \qquad \dfrac{}{(a, g^a, l^a) \xrightarrow{\cup^{v_i}_{M..N}\phi} \bullet}\ N^a = \uparrow$ $\dfrac{(a, g^a, l^a) \xrightarrow{v_i = j; \phi} \bullet}{(a, g^a, l^a) \xrightarrow{\cup^{v_i}_{M..N}\phi} \bullet}\ M^a = m, N^a = n, m \leq j \leq n$ $\dfrac{(a, g^a, l^a) \xrightarrow{v_i = j; \phi} (b, g^b, l^b)}{(a, g^a, l^a) \xrightarrow{\cup^{v_i}_{M..N}\phi} (b, g^b, l^b)}\ M^a = m, N^a = n, m \leq j \leq n$

Figure 4 gives the execution instructions for $\bigcup^{v_i}_{M..N} \phi$ in transition system format. We can now reformulate (13) as (14).

(14) $\bigcup\limits^{i}_{1..N} \phi$.

Translating Alma-0 into L In [2], a hybrid programming language called *Alma-0* is introduced, that combines features of imperative programming like destructive assignment with features of declarative programming like use of boolean expressions as statements and vice versa, and a construction to express "don't know" non-determinism. The potential relevance of Dynamic Predicate Logic for the analysis

Figure 5: Run-time Translation of Alma-0 Programs into L

$B^\diamond$	$:=$	B
$(x := t)^\diamond$	$:=$	$(x := t)^\heartsuit$
$(\text{NOT } S)^\diamond$	$:=$	$\neg S^\diamond$
$(S_1;\ S_2)^\diamond$	$:=$	$S_1^\diamond ; S_2^\diamond$
$(\text{IF } T \text{ THEN } S \text{ END})^\diamond$	$:=$	$(\neg\neg T^\diamond ; S^\diamond) \cup \neg T^\diamond$
$(\text{IF } T \text{ THEN } S_1 \text{ ELSE } S_2)^\diamond$	$:=$	$(\neg\neg T^\diamond ; S_1^\diamond) \cup (\neg T^\diamond ; S_2^\diamond)$
$(\text{EITHER } S_1 \text{ ORELSE} \cdots \text{ ORELSE } S_n)^\diamond$	$:=$	$S_1^\diamond \cup \cdots \cup S_n^\diamond$
$(\text{FOR } i := m \text{ TO } n \text{ DO } S \text{ END})^\diamond$	$:=$	$\begin{cases} \neg\bot & \text{if } m \geq n \\ \exists i; i = m; S^\diamond ; \cdots ; \exists i; i = n; S^\diamond & \text{otherwise} \end{cases}$
$(\text{SOME } i := m \text{ TO } n \text{ DO } S)^\diamond$	$:=$	$\begin{cases} \bot & \text{if } m \geq n \\ \exists i; (i = m; S^\diamond) \cup \cdots \cup (i = n; S^\diamond) & \text{otherwise} \end{cases}$
$(\text{FORALL } S \text{ DO } T \text{ END})^\diamond$	$:=$	$\neg(\exists v_1; \cdots ; \exists v_n; S^\diamond ; \neg T^\diamond)$
		where $v_1, \ldots, v_n$ are the (dynamically) free variables of $S^\diamond$.

Figure 6: Compile-time Translation of Alma-0 Programs into L

$B^\clubsuit$	$:=$	B
$(x := t)^\clubsuit$	$:=$	$(x := t)^\heartsuit$
$(\text{NOT } S)^\clubsuit$	$:=$	$\neg S^\clubsuit$
$(S_1;\ S_2)^\clubsuit$	$:=$	$S_1^\clubsuit ; S_2^\clubsuit$
$(\text{IF } T \text{ THEN } S \text{ END})^\clubsuit$	$:=$	$(\neg\neg T^\clubsuit ; S^\clubsuit) \cup \neg T^\clubsuit$
$(\text{IF } T \text{ THEN } S_1 \text{ ELSE } S_2)^\clubsuit$	$:=$	$(\neg\neg T^\clubsuit ; S_1^\clubsuit) \cup (\neg T^\clubsuit ; S_2^\clubsuit)$
$(\text{EITHER } S_1 \text{ ORELSE} \cdots \text{ ORELSE } S_n)^\clubsuit$	$:=$	$S_1^\clubsuit \cup \cdots \cup S_n^\clubsuit$
$(\text{FOR } i := M \text{ TO } N \text{ DO } S \text{ END})^\clubsuit$	$:=$	$\exists i; i = M - 1; (i \blacktriangleleft_{i'} i' + 1; S^\clubsuit)^{N-M}$
$(\text{SOME } i := M \text{ TO } N \text{ DO } S)^\clubsuit$	$:=$	$\exists i; \bigcup_{M..N}^{i} S^\clubsuit$
$(\text{FORALL } S \text{ DO } T \text{ END})^\clubsuit$	$:=$	$\neg(\exists v_1; \cdots ; \exists v_n; S^\clubsuit ; \neg T^\clubsuit)$
		where $v_1, \ldots, v_n$ are the (dynamically) free variables of $S^\clubsuit$.

of *Alma-0* programs is mentioned as an aside in Section 7 of [2]. This remark is substantiated by the translation functions from *Alma-0* programs into L formulas given in Figures 5 and 6. The first of these translations shows that it is possible to write an interpreter for a considerable subset of *Alma-0* in pure dynamic predicate logic. The interpreter replaces the *Alma-0* instructions at run-time by the DPL instructions indicated by the translation function. The second translation function can be used to compile the same subset of *Alma-0* programs into dynamic predicate logic extended with the constructs for bounded iteration and bounded choice that were introduced above.

Admittedly, we can only translate the fragment of *Alma-0* that is not concerned with side effects on the control flow during backtracking. But this is only natural. In pure dynamic logic programming there is no place for side effects having to do with the manner of execution of the search through the transition space of all possible answers. Thus, we cannot deal with *Alma-0* commands like COMMIT.

The virtue of the dynamic logic programming perspective is that it allows us to separate the pure kernel of a hybrid programming language like *Alma-0* from its 'operationally contaminated' periphery. Interestingly, from the present viewpoint the use of assignment *per se* does not make *Alma-0* into a hybrid language. If we are prepared to replace assignment by safe assignment, we see that *Alma-0* has a large part that can be given a (dynamic) declarative semantics.

One of the show-pieces of *Alma-0* programming is the solution of the N queens problem by means of an elegant *Alma-0* program (cf. [2]). We will use the same problem for a demonstration of dynamic logic programming. The challenge is to position N queens on a chess-board of size $N \times N$ in such manner that none of them is under attack from any of the others: no two queens are on the same row, on the same column or on the same diagonal.

If we reason in a model with two kinds of objects, rows and columns, then there are N rows $1, \ldots, N$ and N columns $1, \ldots, N$, so we can put $C = \{1, \ldots, N\}$ and $R = \{1, \ldots, N\}$, and represent a placement of the queens on the board as a function $f : C \rightarrow R$. Then $f(4) = 5$ expresses that the queen of column 4 is in row 5. If $f(k) = r$, then the three constraints that have to be observed are:

1. No queen from an earlier column, i.e, from $i \in \{1, \ldots, k-1\}$, should be on row r.
2. No queen from $i \in \{1, \ldots, k-1\}$ should be on the $\nearrow$ diagonal through r.
3. No queen from $i \in \{1, \ldots, k-1\}$ should be on the $\searrow$ diagonal through r.

Here is the dynamic logic rendering of these constraints (we assume that we want to fix the range of choice at compile time, so we use the construct for variable-controlled bounded choice).

1. $\neg(\exists i; \bigcup^{i}_{1..k-1} f(i) = k)$.
2. $\neg(\exists i; \bigcup^{i}_{1..k-1} f(i) = r + i - k)$.
3. $\neg(\exists i; \bigcup^{i}_{1..k-1} f(i) = k + r - i)$.

These programs can be combined into a single check:

(F) $\quad \neg(\exists i; \cup^{i}_{1..k-1}(f(i) = k \cup f(i) = r + i - k \cup f(i) = k + r - i))$.

The full dynamic logic program that checks whether f is a correct solution to the N queens problem is:

(15) $\quad \exists k; k = 0; (k \blacktriangleleft_{k'} k' + 1; \exists r; \cup^{r}_{1..N}(f(k) = r; F))^{N}$.

To make this into a program that *generates* a correct solution to the N queens problem, the extra touch we have to add is to make f into an indexed array, so that $f(i)$ is considered as a single variable. The command $\exists f$ now resets the array, and the N queens query can look like this:

(16) $\exists f; \exists k; k = 0; (k \blacktriangleleft_{k'} k' + 1; \exists r; \cup^{r}_{1..N}(f(k) = r; F))^N$.

If we call program (15) with array f not initialized, or program (16) to make sure that array f gets reset, then the equality statements $f(k) = r$ become assignments to $f(k)$, and the program tries out all the possibilities until it finds one that works. If the program succeeds the output state for $f(1), \dots, f(N)$ gives a correct solution to the N queens problem. The correctness guarantee of this solution lies in the adequacy theorem for our executable interpretation of dynamic first order logic (Theorem 31).

Advantage of Declarative Style The advantages of dynamic logic programming, compared to imperative programming become evident in the fact that pre- and postcondition reasoning over L programs is extremely easy, for we can apply Definition 10. We have:

Theorem 32

1. *If* $\mathcal{M} \models_a \mathrm{WPR}_\phi(\psi)$ *and* $a \xrightarrow{\phi} (b, g^b, l^b)$ *then* $\mathcal{M} \models_b \psi$.
2. *If* $\mathcal{M} \models_b \mathrm{STP}_\phi(\psi)$ *and* $a \xrightarrow{\phi} (b, g^b, l^b)$ *then* $\mathcal{M} \models_a \psi$.

Proof. Immediate from Theorem 11 (correctness of WPR and STP definitions) and Theorem 31 (faithfulness of executable interpretation to DPL). □

Adding Unbounded Iteration The only thing that is conspicuously missing from our dynamic logic programming tool-set is a means of expressing repetition or unbounded iteration. Nothing prevents us from just adding this tool, although we have to keep in mind that it comes at a price: pre- and post-condition reasoning for the enhanced language becomes much more involved, and, perhaps more important, we lose the guarantee that the computation procedure terminates on any input.

Again, we assume that a signature $(\mathbf{P}, \mathbf{F}, \mathbf{Ar})$ is given suitable for calculation in the model of our choice. This model must include $\mathbb{Z}$, because the extended language has control variables ranging over integers.

Definition 33 (Terms and formulas of $\mathbf{L}^*$)

$$\begin{array}{lll} t & ::= & v \mid ft_1 \cdots t_n \\ N & ::= & 0 \mid 1 \mid \cdots \mid v_i \mid -N \mid (N_1 + N_2) \mid (N_1 - N_2) \mid (N_1 * N_2) \\ \phi & ::= & \perp \mid Pt_1 \cdots t_n \mid t_1 \doteq t_2 \mid \exists v \mid \neg\phi \mid (\phi_1;\phi_2) \mid (\phi_1 \cup \phi_2) \mid \phi^n \mid \phi^N \mid \bigcup^{v_i}_{M..N} \phi \mid \phi^*. \end{array}$$

Here is the extension of the dynamic interpretation function for the extended language:

Definition 34 (Dynamic interpretation of $\mathbf{L}^*$ in a model $\mathcal{M} = (M, I)$)

$$\begin{array}{lll} [\![\phi^0]\!](a) & := & \{a\} \\ [\![\phi^{n+1}]\!](a) & := & [\![\phi;\phi^n]\!](a) \\ [\![\phi^N]\!](a) & := & [\![\phi^{\max(0,N^a)}]\!](a) \\ [\![\bigcup^{v_i}_{M..N}\phi]\!](a) & := & \begin{cases} [\![v_i = M;\phi]\!](a) \cup \cdots \cup [\![v_i = N;\phi]\!](a) & \textit{if } M^a \leq N^a \\ \emptyset & \textit{otherwise.} \end{cases} \\ [\![\phi^*]\!](a) & := & \bigcup_{n\in\mathbb{N}}[\![\phi^n]\!](a). \end{array}$$

For the executable process interpretation, we add the following:

Definition 35 (Executable process interpretation of $\mathbf{L}^*$, in a model $\mathcal{M} = (M, I)$)

$$
\begin{array}{lcl}
[\![\phi^0]\!](\boldsymbol{a}) & := & \{\boldsymbol{a}\} \\
[\![\phi^{n+1}]\!](\boldsymbol{a}) & := & [\![\phi;\phi^n]\!](\boldsymbol{a}) \\
[\![\phi^N]\!](\boldsymbol{a}) & := & \begin{cases} [\![\phi^{\max(0,N^a)}]\!](\boldsymbol{a}) & \textit{if } N^a =\downarrow, \\ \{\bullet\} & \textit{if } N^a =\uparrow . \end{cases} \\
[\![\bigcup_{M..N}^{v_i} \phi]\!](\boldsymbol{a}) & := & \begin{cases} [\![v_i = M;\phi]\!](\boldsymbol{a}) \cup \cdots \cup [\![v_i = N;\phi]\!](\boldsymbol{a}) \\ \qquad \textit{if } M^a =\downarrow, N^a =\downarrow, M^a \leq N^a, \\ \emptyset \quad \textit{if } M^a =\downarrow, N^a =\downarrow, M^a > N^a, \\ \{\bullet\} \quad \textit{if } M^a =\uparrow \ \textit{or } N^a =\uparrow . \end{cases} \\
[\![\phi^*]\!](\boldsymbol{a}) & := & \bigcup_{n\in\mathbb{N}}\{[\![\phi^n]\!](\boldsymbol{a})\}
\end{array}
$$

Figure 7: Transitions for Unbounded Iteration

ϕ^*	$\dfrac{}{(a, g^a, l^a) \xrightarrow{\phi^*} (a, g^a, l^a)}$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi} \bullet}{(a, g^a, l^a) \xrightarrow{\phi^*} \bullet}$ $\quad$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b) \xrightarrow{\phi^*} \bullet}{(a, g^a, l^a) \xrightarrow{\phi^*} \bullet}$ $\dfrac{(a, g^a, l^a) \xrightarrow{\phi} (b, g^b, l^b) \xrightarrow{\phi^*} (c, g^c, l^c)}{(a, g^a, l^a) \xrightarrow{\phi^*} (c, g^c, l^c)}$

To capture this extension of the definition in the transition system format, all we have to add the transitions shown in Figures 3, 4 and 7. Here is a dynamic logic program for computing the GCD of x and y:

(17) $((x > y; x \blacktriangleleft_{x'} x' - y) \cup (y > x; y \blacktriangleleft_{y'} y' - x))^*; x = y.$

Translating Dijkstra's Guarded Command Programs into $\mathbf{L}^*$ The translation of Dijkstra's guarded command language (see e.g. [7]) into L^* given in Figure 8 demonstrates that the addition of unbounded iteration makes our logical language into a full-fledged programming language.

A Recipe for Dynamic Logic Programming Here is the short of it all:

> Take imperative programming. First remove destructive assignment. Then add dynamic interpretation. What you get is dynamic logic programming: just as powerful, but much more perspicuous, especially for programs without unbounded iteration.

Figure 8: Translating Dijkstra's Guarded Command Programs into L*

$\mathrm{SKIP}^{\spadesuit}$	$:=$	$\neg\bot$
$B^{\spadesuit}$	$:=$	B
$(x := t)^{\spadesuit}$	$:=$	$(x := t)^{\heartsuit}$
$(S_1;\, S_2)^{\spadesuit}$	$:=$	$S_1^{\spadesuit}; S_2^{\spadesuit}$
$(\mathrm{IF}\ B_1 \to S_1 \| \cdots \| B_n \to S_n\ \mathrm{FI})^{\spadesuit}$	$:=$	$(B_1; S^{\spadesuit}) \cup \cdots \cup (B_n; S_n^{\spadesuit})$
$(\mathrm{DO}\ B_1 \to S_1 \| \cdots \| B_n \to S_n\ \mathrm{OD})^{\spadesuit}$	$:=$	$((B_1; S^{\spadesuit}) \cup \cdots \cup (B_n; S_n^{\spadesuit}))^*; \neg B_1; \cdots; \neg B_n$

6 Operational Semantics and Executable Process Interpretation

We now turn to the relation between the operational semantics for first order logic given in [1] and the executable process semantics from Section 2 above. For the rules of the operational semantics, see Figure 9. Cases not shown generate an error. The rules define a computation tree for every pair ϕ, a consisting of a first order formula ϕ and a valuation a. Leafs of a computation tree can be labelled with a valuation, with *fail*, or with *error*. Call a leaf labelled with a valuation a success leaf, a leaf labelled with *fail* a failure leaf, and a leaf labelled with *error* an error leaf. A computation tree is *successful* if it contains at least one success leaf, it is *failed* if it contains only failure leafs, it is *determined* if it is either successful or failed, and it is *undetermined* if it contains only error leafs. For further details on the definition of computation trees, compare [1] or the proof of Theorem 36 below.

Before we describe the relation between the operational and the executable process semantics, a preliminary remark is in order. It is important to note that the perspective at first order logic taken by Apt and Bezem is different from ours in the following respect. Their thinking starts from certain problems in constraint programming, and they use first order logic to provide denotational meanings for their programs. It follows that the fact that certain first order formulas do not get an operational meaning (but cause error abortion instead) is immaterial, as long as there are alphabetic variants of the formulas that express the programming constructs under consideration. However, when one shifts perspective and asks which part of first order logic can be used for programming, the fact that certain formulas do not have operational meanings suddenly matters.

In view of this it is not at all surprising that the definition of the operational semantics, when taken at face value, implies that the denotational semantics is (much) more general than the operational semantics, The denotational semantics is defined for all predicate logical formulas, while the operational semantics is only defined for formulas satisfying certain not-quite-standard variable restrictions.

Apart from the obvious differences between operational and denotational perspectives (the first is more fine-grained, and takes order of execution of computation paths into account, where the denotational semantics views the computation paths as parallel possibilities), the key difference between operational and denotational semantics is in the treatment of the existential quantifier.

The definition of the operational semantics for the existential quantifier [1] states the following rule for $\exists x\phi \wedge \psi$, given valuation a, and on condition that x not in the domain of a, and that x is not free in ψ.

Figure 9: Apt and Bezem's Operational Semantics for FOL

$\Box$	$\begin{array}{c} \Box, a \\ \mid \\ a \end{array}$	
$Pt_1 \cdots t_n \wedge \phi$	$\begin{array}{c} Pt_1 \cdots t_n \wedge \phi, a \\ \mid \\ \llbracket \phi \rrbracket_a \end{array}$	$Pt_1 \cdots t_n$ a-closed and true.
	$\begin{array}{c} Pt_1 \cdots t_n \wedge \phi, a \\ \mid \\ \mathit{fail} \end{array}$	$Pt_1 \cdots t_n$ a-closed and false.
$t_1 \doteq t_2 \wedge \phi$	$\begin{array}{c} t_1 \doteq t_2 \wedge \phi, a \\ \mid \\ \llbracket \phi \rrbracket_{a \cup \{v/t_2^a\}} \end{array}$	$t_1 \equiv v$, $v^a =\uparrow, t_2^a =\downarrow$.
	$\begin{array}{c} t_1 \doteq t_2 \wedge \phi, a \\ \mid \\ \llbracket \phi \rrbracket_{a \cup \{v/t_1^a\}} \end{array}$	$t_2 \equiv v$, $v^a =\uparrow, t_1^a =\downarrow$.
		$t_1 \doteq t_2$ not an a-assignment: similar to treatment of $Pt_1 \cdots t_n$.
$\exists v \phi \wedge \psi$	$\begin{array}{c} \exists v \phi \wedge \psi, a \\ \mid \\ \llbracket \phi \wedge \psi \rrbracket_a \end{array}$	$v \notin \mathit{dom}(a)$, v not free in ψ.
$\neg \phi \wedge \psi$	$\begin{array}{c} \neg \phi \wedge \psi, a \\ \mid \\ \llbracket \psi \rrbracket_a \end{array}$	ϕ a-closed, $\llbracket \phi \rrbracket_a$ failed.
	$\begin{array}{c} \neg \phi \wedge \psi, a \\ \mid \\ \mathit{fail} \end{array}$	ϕ a-closed, $\llbracket \phi \rrbracket_a$ contains success leaf.
$(\phi_1 \rightarrow \phi_2) \wedge \psi$	$\begin{array}{c} (\phi_1 \rightarrow \phi_2) \wedge \psi, a \\ \mid \\ \llbracket \psi \rrbracket_a \end{array}$	ϕ_1 a-closed,$\llbracket \phi_1 \rrbracket_a$ failed.
	$\begin{array}{c} (\phi_1 \rightarrow \phi_2) \wedge \psi, a \\ \mid \\ \llbracket \phi_2 \wedge \psi \rrbracket_a \end{array}$	ϕ_1 a-closed, $\llbracket \phi_1 \rrbracket_a$ contains success leaf.
$(\phi_1 \vee \phi_2) \wedge \psi$	$\begin{array}{cc} \multicolumn{2}{c}{(\phi_1 \vee \phi_2) \wedge \psi, a} \\ / & \backslash \\ \llbracket \phi_1 \wedge \psi \rrbracket_a & \llbracket \phi_2 \wedge \psi \rrbracket_a \end{array}$	
All cases not listed:	*error*	

$$\begin{array}{c}\exists x\phi \wedge \psi, a\\ |\\ [\![\phi \wedge \psi]\!]_a\end{array}$$

The definition does not state what the computation tree looks like in cases where one of the two conditions is not fulfilled. Taken literally, the definition implies that the operational tree for $x = 0 \wedge \exists xx = 1$ is not defined.

This is awkward, for since the computation trees of the operational semantics are defined recursively, undefinedness of the computation tree for $\exists x\phi \wedge \psi$ with respect to a with x in the domain of a has wide repercussions, e.g., for $\exists xx = 0 \wedge \exists xx = 1$. Undefinedness of the computation tree for $x = 0 \wedge \exists xx = 1$ implies that the computation tree for $\exists xx = 0 \wedge \exists xx = 1$ is not defined either, for the tree for $\exists xx = 0 \wedge \exists xx = 1$ is defined in terms of the tree for $x = 0 \wedge \exists xx = 1$.

We will assume that in cases where a variable condition is violated the computation tree is defined, but undetermined, i.e., we augment the definition of computation trees with the following, for the case that x in the domain of a or x free in ψ:

$$\begin{array}{c}\exists x\phi \wedge \psi, a\\ |\\ \mathit{error}\end{array}$$

With this amendment, the operational semantics for quantification treats $\exists x\phi \wedge \psi$ as equivalent with $\phi \wedge \psi$ provided the variable conditions hold, so it assumes that the quantifier is a spurious decoration, or rather it assumes that the quantifier serves as a prohibition sign: its only function is to rule out occurrences of x in the outside context.

Important Remark Apt (p.c.) suggests another way to amend the treatment of $\exists x\phi \wedge \psi, a$. In case x in the domain of a or x free in ψ, replace free occurrences of x in ϕ by a fresh variable z, and after the processing of $\phi[z/x]$, discard the pairs (z/d) from the valuations at the success nodes of its computation tree:

$$\begin{array}{c}\exists x\phi \wedge \psi, a\\ |\\ [\![\phi[z/x] \wedge \psi]\!]_a^{-z}\end{array}$$

This suggestion, though excellent, will not be explored or followed up in this paper, for the simple reason that it yields an operational semantics that is not faithful to the dynamic interpretation of FOL. The two formulas $\exists xx = 1 \wedge Px$ and $\exists zz = 1 \wedge Px$ are classically equivalent, but have different dynamic meanings. In $\exists xx = 1 \wedge Px$, the atom Px is in the dynamic scope of a quantifier, in $\exists zz = 1 \wedge Px$ it is not. Another thing to note here is that the implemented language *Alma-0* that occasioned the research in [1] does not comply with this operational rule for quantification. In fact, the meaning of the *Alma-0*

SOME i := 1 TO n DO S

corresponds exactly to the denotational meaning of the following dynamic logic formula:

$n = n; \exists i; ((i = 1; S) \cup \cdots \cup (i = n; S))$.

The value of dynamic predicate logic for gaining a proper perspective on the denotational semantics of *Alma-0* programs is revealed more fully by the translation functions given in Figures 5 and 6. **End of Remark**

For the example under discussion we get that the computation tree is undetermined:

$$\begin{array}{c} \exists x x = 0 \wedge \exists x x = 1, \epsilon \\ | \\ x = 0 \wedge \exists x x = 1, \epsilon \\ | \\ \exists x x = 1, \{x/0\} \\ | \\ \mathit{error} \end{array}$$

The final node aborts with error because x is in the domain of the current valuation, a condition that is forbidden by the rule for $\exists x$. Without the amendment, the computation tree for $\exists x x = 0 \wedge \exists x x = 1$ with respect to ϵ would have been undefined.

Relation Between Operational And Denotational Semantics The relation between the operational and executable process semantics can be summarized as follows:

1. The operational semantics is a faithful approximation of the executable process interpretation in the sense that valuations computed by the operational semantics are also computed by the executable process interpretation, failure under the operational semantics implies failure under the executable process interpretation, and error free computation by the operational semantics for a given input a implies that the executable process interpretation for input a does not have $\bullet$ in its output set.
2. The operational semantics is a faithful approximation to the DPL interpretation of first order logic.
3. The executable process interpretation properly extends the operational semantics in the sense that it allows one to compute output valuations in many cases where the operational semantics generates an error (where the operational semantics throws its towel in the ring and gives up, so to speak).
4. The executable process interpretation is more abstract than the operational semantics, because it is defined for both finite and infinite inputs, and it has commutativity of choice built in.

The first of these claims will be established by Theorem 36 at the end of this section. The second claim follows immediately from this and the faithfulness to DPL of the executable process interpretation. We state it in Theorem 37. The third claim follows from the difference in treatment of existential quantification. The third and fourth points will now be further illustrated by examples.

Executable Process Interpretation Properly Extends Operational Semantics The formula $\exists x x = 0 \wedge \exists x x = 1$, when computed on the domain of natural numbers, is trivially true, and a program for this that is worth its mettle should succeed (and compute an output valuation).

We have seen above that the operational semantics for this example is either undefined (the original definition) or undetermined (our amended version). The executable process interpretation for this example works out as follows:

$$
\begin{aligned}
\{\epsilon\}[\![\exists x x = 0 \wedge \exists x x = 1]\!] &= \{\epsilon\}[\![\exists x]\!][\![x = 0]\!][\![\exists x]\!][\![x = 1]\!] \\
&= \{\epsilon\}[\![x = 0]\!][\![\exists x]\!][\![x = 1]\!] \\
&= \{\{x/0\}\}[\![\exists x]\!][\![x = 1]\!] \\
&= \{\epsilon\}[\![x = 1]\!] \\
&= \{\{x/1\}\}.
\end{aligned}
$$

The executable process interpretation is also more enterprising than the operational semantics in the treatment of negation and implication.

The operational semantics for $\neg\phi$ in the context of a valuation a, insists that the computation tree is only determined if ϕ is a-closed. The executable process interpretation has no such limitation. However, in the appendix of [1], the operational rule for negation is liberalized, by modifying the restriction ϕ is a-closed in the rule for $\neg\phi \wedge \psi$ in the context of valuation a. In case the computation tree for ϕ in context a, $[\![\phi]\!]_a$, is failed, even if ϕ is not a-closed, then $\neg\phi \wedge \psi$ has as computation tree the tree with root $\neg\phi \wedge \psi, a$ of degree one, and the tree $[\![\psi]\!]_a$ as its subtree. In case the computation tree for ϕ in context a has a success leaf b that does not assign values to any x free in ϕ^a then $[\![\neg\phi \wedge \psi]\!]_a$ counts as failed, for then $\neg\phi \wedge \psi$ has the following computation tree:

$$
\begin{array}{c}
\neg\phi \wedge \psi, a \\
| \\
\textit{fail}
\end{array}
$$

It is easy to see that this is an approximation to the executable process treatment of negation: in all cases where ϕ does not contain quantifiers it gives the same results.

For the case of implication, take $\phi \to \psi$ as shorthand for $\neg\phi \cup (\neg\neg\phi; \psi))$. Other reasonable definitions of implication are also possible, but this is the one which turns out to be the right choice for the faithfulness proof in Theorem 36. Consider the example $x = 1 \to x = 1$, and assume a valuation a that does not have x in its range, e.g., the valuation ϵ. Then, according to the operational semantics for implication, the computation tree will undetermined, for it will look like this:

$$
\begin{array}{c}
x = 1 \to x = 1, \epsilon \\
| \\
\textit{error}
\end{array}
$$

The executable process interpretation is completely well-behaved, as the following computation shows:

$$
\begin{aligned}
\{\epsilon\}[\![x = 1 \to x = 1]\!] &= \{\epsilon\}[\![\neg x = 1 \cup (\neg\neg x = 1; x = 1)]\!] \\
&= \{\epsilon\}[\![\neg x = 1]\!] \cup \{\epsilon\}[\![\neg\neg x = 1; x = 1]\!] \\
&= \emptyset \cup \{\epsilon\}[\![\neg\neg x = 1; x = 1]\!] \\
&= \emptyset \cup \{\epsilon\}[\![\neg\neg x = 1]\!][\![x = 1]\!] \\
&= \emptyset \cup \{\epsilon\}[\![x = 1]\!] \\
&= \emptyset \cup \{\{x/1\}\} \\
&= \{\{x/1\}\}.
\end{aligned}
$$

Note however that Apt and Bezem [1] (appendix) discuss the possibility to also liberalize their treatment of implication.

Executable Process Interpretation More Abstract The executable process interpretation is more abstract than the computational semantics in that it does not insist on valuations being finite and that it has commutativity of choice built in. No value judgment is implied, of course, for a less abstract semantics may be the right choice if one is concerned with the description of purely operational features, such as the way in which backtracking is implemented in a language. The executable process interpretation abstracts from such features. It does not distinguish between $\phi \cup \psi$ and $\psi \cup \phi$, while these formulas have different (although equivalent) computation trees under the operational semantics. To illustrate this difference, we compare the semantic denotation for formula 4 with the computation tree for that formula in [1]. Observe that the following formula has the same executable process interpretation but a different computation tree:

$$(x = 3 \vee x = 2) \wedge (2 = y \vee y = x + 1) \wedge (2 * x = 3 * y).$$

Operational Semantics Approximates Executable Process Interpretation The operational semantics is a faithful approximation to the executable process interpretation in the following sense. Define the yield of a computation tree $[\![\phi]\!]_a$ as follows:

$$y([\![\phi]\!]_a) := \begin{cases} \{b \in \mathcal{A} \mid b \text{ decorates success leaf of } [\![\phi]\!]_a\} & \text{if } [\![\phi]\!]_a \text{ determined,} \\ \{\bullet\} & \text{if } \mid [\![\phi]\!]_a \text{ undetermined .} \end{cases}$$

Then $y([\![\phi]\!]_a) \subseteq \mathcal{A} \cup \{\bullet\}$, and we have:

Theorem 36 (Executable Process Interpretation Extends Operational Semantics) *For every formula ϕ, every finite $a \in \mathcal{A}$, all sets $g^a, l^a \subseteq V$:*

$$y([\![\phi]\!]_a) \sqsubseteq ([\{\phi\}](a, g^a, l^a))^\natural.$$

Before we give the proof, note that the theorem implies all of the following (compare Theorem 22 above):

1. If $\bullet \in [\{\phi\}](a, g^a, l^a)$ then $\bullet \in y([\![\phi]\!]_a)$. This is because $A \sqsubseteq (B \cup \{\bullet\})$ implies $\bullet \in A$.
2. If $\bullet \notin y([\![\phi]\!]_a)$ then $y([\![\phi]\!]_a) \subseteq ([\{\phi\}](a, g^a, l^a))^\natural$. This is because $(A - \{\bullet\}) \sqsubseteq B$ implies $(A - \{\bullet\}) \subseteq B$.
3. If $[\![\phi]\!]_a$ is failed then $[\{\phi\}](\{(a, g^a, l^a)\}) = \emptyset$. This is because $\emptyset \sqsubseteq A$ implies $A = \emptyset$.

Proof. (of Theorem 36) We use induction on the complexity of the computation tree, following the decomposition of formulas used in the definition of computation trees, with empty conjunction $\Box$ defined as $\neg\bot$, and implication $\phi \to \psi$ defined as $\neg\phi \cup (\neg\neg\phi; \psi)$.

$\Box$: For any a, the computation tree $[\![\Box]\!]_a$ is given by:

$$\begin{array}{c} \Box, a \\ | \\ a \end{array}$$

Thus, $y([\![\Box]\!]_a) = \{a\}$, and it is easily checked that $([\{\neg\bot\}](a, g^a, l^a))^\natural = \{a\}$. This proves the claim.

$Pt_1 \cdots t_n \wedge \phi, a$: Case 1. $Pt_1 \cdots t_n$ a-closed and true. Then the computation tree looks like:

$$\begin{array}{c} Pt_1 \cdots t_n \wedge \phi, a \\ | \\ [\![\phi]\!]_a \end{array}$$

In this case we have: $[\![\{Pt_1 \cdots t_n; \phi\}]\!](a, g^a, l^a) = [\![\{\phi\}]\!](a, g^a, l^a)$ and the claim follows by an application of the induction hypothesis.

$Pt_1 \cdots t_n \wedge \phi, a$: Case 2. $Pt_1 \cdots t_n$ a-closed and false. Then the computation tree looks like:

$$\begin{array}{c} Pt_1 \cdots t_n \wedge \phi, a \\ | \\ \mathit{fail} \end{array}$$

In this case we have: $[\![\{Pt_1 \cdots t_n\}]\!](a, g^a, l^a) = \emptyset$, so $[\![\{Pt_1 \cdots t_n; \phi\}]\!](a, g^a, l^a) = \emptyset$, which establishes the claim.

$Pt_1 \cdots t_n \wedge \phi, a$: Case 3. $Pt_1 \cdots t_n$ not a-closed. Then the computation tree looks like:

$$\begin{array}{c} Pt_1 \cdots t_n \wedge \phi, a \\ | \\ \mathit{error} \end{array}$$

Since in this case $y([\![Pt_1 \cdots t_n \wedge \phi]\!]_a) = \{\bullet\}$ there is nothing to prove.

$t_1 \doteq t_2 \wedge \phi, a$: Case 1. t_1, t_2 both a-closed. Then reason as in the case of relational atoms.

$t_1 \doteq t_2 \wedge \phi, a$: Case 2. $t_1 \equiv v$, $v^a = \uparrow, t_2^a = \downarrow$. Then the computation tree looks as follows:

$$\begin{array}{c} v \doteq t_2 \wedge \phi, a \\ | \\ [\![\phi]\!]_{a \cup \{v/t_2^a\}} \end{array}$$

In this case, in the executable process semantics we have: $[\![\{v \doteq t\}]\!](a, g^a, l^a) = \{(a \cup \{v/t^a\}, g^{a\prime}, l^a)\}$, with $g^{a\prime} = g^a$ if $v \in l^a$, $g^{a\prime} = g^a \cup \{v\}$ otherwise, and the claim follows by an application of the induction hypothesis.

$t_1 \doteq t_2 \wedge \phi, a$: Case 3. $t_2 \equiv v$, $v^a = \uparrow, t_1^a = \downarrow$. Symmetric to previous case.

$t_1 \doteq t_2 \wedge \phi, a$: Case 4. $t_1 \doteq t_2$ not a-closed, but cases 2 or 3 do not apply. The computation tree for this case looks like this:

$$\begin{array}{c} t_1 \doteq t_2 \wedge \phi, a \\ | \\ \mathit{error} \end{array}$$

Since in this case $y([\![t_1 \doteq t_2 \wedge \phi]\!]_a) = \{\bullet\}$ there is nothing to prove.

$(\phi_1 \vee \phi_2) \wedge \psi, a$: The computation tree:

$$
\begin{array}{ccc}
 & (\phi_1 \vee \phi_2) \wedge \psi, a & \\
\swarrow & & \searrow \\
\llbracket \phi_1 \wedge \psi \rrbracket_a & & \llbracket \phi_2 \wedge \psi \rrbracket_a
\end{array}
$$

In the executable process interpretation:

$$
\begin{array}{rcl}
\llbracket (\phi_1 \cup \phi_2); \psi \rrbracket(\{(a, g^a, l^a)\}) & = & \llbracket \psi \rrbracket(\llbracket \phi_1 \cup \phi_2 \rrbracket(\{(a, g^a, l^a)\})) \\
 & = & \llbracket \psi \rrbracket(\llbracket \phi_1 \rrbracket(\{(a, g^a, l^a)\}) \cup \llbracket \phi_2 \rrbracket(\{(a, g^a, l^a)\})) \\
 & = & \llbracket \psi \rrbracket(\llbracket \phi_1 \rrbracket(\{(a, g^a, l^a)\})) \cup \llbracket \psi \rrbracket(\llbracket \phi_2 \rrbracket(\{(a, g^a, l^a)\})) \\
 & = & \llbracket \phi_1; \psi \rrbracket(\{(a, g^a, l^a)\}) \cup \llbracket \phi_2; \psi \rrbracket(\{(a, g^a, l^a)\})
\end{array}
$$

and the claim follows by two applications of the induction hypothesis.

$(\phi_1 \wedge \phi_2) \wedge \psi, a$: The computation tree for $(\phi_1 \wedge \phi_2) \wedge \psi, a$ is by definition the same as that for $\phi_1 \wedge (\phi_2 \wedge \psi), a$. To prove the claim in this case, use the fact that the executable process interpretation of sequential composition is associative, plus the induction hypothesis. The induction hypothesis applies since $\phi_1 \wedge (\phi_2 \wedge \psi)$ is by definition less complex than $(\phi_1 \wedge \phi_2) \wedge \psi$.

$(\phi_1 \rightarrow \phi_2) \wedge \psi, a$: Case 1. ϕ_1 is a-closed and $\llbracket \phi_1 \rrbracket_a$ is failed. Then the computation tree looks as follows:

$$
\begin{array}{c}
(\phi_1 \rightarrow \phi_2) \wedge \psi, a \\
| \\
\llbracket \psi \rrbracket_a
\end{array}
$$

Using the definition of $\phi_1 \rightarrow \phi_2$ as $\neg\phi_1 \cup \neg\neg\phi_1; \phi_2$ we get the following executable process semantics (in the calculation we use the fact that $\llbracket \phi_1 \rrbracket(\{(a, g^a, l^a)\}) = \emptyset$, obtained from the induction hypothesis, applied to ϕ_1):

$$
\begin{array}{rcl}
\llbracket (\phi_1 \rightarrow \phi_2); \psi \rrbracket(\{(a, g^a, l^a)\}) & = & \llbracket (\neg\phi_1 \cup \neg\neg\phi_1; \phi_2); \psi \rrbracket(\{(a, g^a, l^a)\}) \\
 & = & \llbracket \psi \rrbracket(\llbracket \neg\phi_1 \cup \neg\neg\phi_1; \phi_2 \rrbracket(\{(a, g^a, l^a)\})) \\
 & = & \llbracket \psi \rrbracket(\llbracket \neg\phi_1 \rrbracket(\{(a, g^a, l^a)\}) \cup \llbracket \neg\neg\phi_1; \phi_2 \rrbracket(\{(a, g^a, l^a)\})) \\
 & = & \llbracket \psi \rrbracket(\{(a, g^a, l^a)\}) \cup \llbracket \neg\neg\phi_1; \phi_2 \rrbracket(\{(a, g^a, l^a)\})) \\
 & = & \llbracket \psi \rrbracket(\{(a, g^a, l^a)\} \cup \emptyset) \\
 & = & \llbracket \psi \rrbracket(\{(a, g^a, l^a)\})
\end{array}
$$

The claim follows from an application of the induction hypothesis to ψ.

$(\phi_1 \rightarrow \phi_2) \wedge \psi, a$: Case 2. ϕ_1 is a-closed and $\llbracket \phi_1 \rrbracket_a$ contains a success leaf. Then the computation tree looks as follows:

$$
\begin{array}{c}
(\phi_1 \rightarrow \phi_2) \wedge \psi, a \\
| \\
\llbracket \phi_2 \wedge \psi \rrbracket_a
\end{array}
$$

In this case, we get from the induction hypothesis, applied to ϕ_1 that $[\{\phi_1\}](a, g^a, l^a)$ contains a safe member $\mathbf{b}$. Therefore $[\{\neg\phi_1\}](a, g^a, l^a) = \emptyset$, so $[\{\neg\neg\phi_1\}](a, g^a, l^a) = \{(a, g^a, l^a)\}$, and we get the following denotation:

$$
\begin{array}{rcl}
\llbracket(\phi_1 \to \phi_2);\psi\rrbracket(\{(a,g^a,l^a)\}) & = & \llbracket(\neg\phi_1 \cup \neg\neg\phi_1;\phi_2);\psi\rrbracket(\{(a,g^a,l^a)\}) \\
& = & \llbracket\psi\rrbracket(\llbracket\neg\phi_1 \cup \neg\neg\phi_1;\phi_2\rrbracket(\{(a,g^a,l^a)\})) \\
& = & \llbracket\psi\rrbracket(\llbracket\neg\phi_1\rrbracket(\{(a,g^a,l^a)\}) \cup \llbracket\neg\neg\phi_1;\phi_2\rrbracket(\{(a,g^a,l^a)\})) \\
& = & \llbracket\psi\rrbracket(\emptyset \cup \llbracket\neg\neg\phi_1;\phi_2\rrbracket(\{(a,g^a,l^a)\})) \\
& = & \llbracket\psi\rrbracket(\emptyset \cup \llbracket\phi_2\rrbracket(\{(a,g^a,l^a)\})) \\
& = & \llbracket\psi\rrbracket(\llbracket\phi_2\rrbracket(\{(a,g^a,l^a)\})) \\
& = & \llbracket\phi_2;\psi\rrbracket(\{(a,g^a,l^a)\})
\end{array}
$$

The claim follows from this by an application of the induction hypothesis.

$(\phi_1 \to \phi_2) \wedge \psi, a$: Case 3. ϕ_1 not a-closed. Then the computation tree looks as follows:

$$
\begin{array}{c}
(\phi_1 \to \phi_2) \wedge \psi, a \\
| \\
\mathit{error}
\end{array}
$$

Since in this case $y(\llbracket\phi_1 \to \phi_2) \wedge \psi\rrbracket_a) = \{\bullet\}$ there is nothing to prove.

$\neg\phi \wedge \psi, a$: Case 1. ϕ is a-closed and $\llbracket\phi\rrbracket_a$ contains only failure leaves. Then the computation tree looks like this:

$$
\begin{array}{c}
\neg\phi \wedge \psi, a \\
| \\
\llbracket\psi\rrbracket_a
\end{array}
$$

Then by the induction hypothesis, applied to ϕ, we have that $[\{\phi\}](a,g^a,l^a) = \emptyset$, and therefore:

$$[\{\neg\phi\}](a,g^a,l^a) = \{(a,g^a,l^a)\}.$$

For the denotational semantics, this gives:

$$
\begin{array}{rcl}
\llbracket\neg\phi;\psi\rrbracket(\{(a,g^a,l^a)\}) & = & \llbracket\psi\rrbracket(\llbracket\neg\phi\rrbracket(\{(a,g^a,l^a)\})) \\
& = & \llbracket\psi\rrbracket(\{(a,g^a,l^a)\}).
\end{array}
$$

The claim follows from this by an application of the induction hypothesis.

$\neg\phi \wedge \psi, a$: Case 2. ϕ is a-closed and $\llbracket\phi\rrbracket_a$ contains at least one success leaf. Then the computation tree looks like this:

$$
\begin{array}{c}
\neg\phi \wedge \psi, a \\
| \\
\mathit{fail}
\end{array}
$$

By the induction hypothesis, applied to ϕ, we have that $[\{\phi\}](a,g^a,l^a) = \{(a,g^a,l^a)\}$, and therefore $[\{\neg\phi\}](a,g^a,l^a) = \emptyset$.

For the denotational semantics, this gives:

$$\begin{aligned}
\llbracket \neg\phi;\psi \rrbracket(\{(a, g^a, l^a)\}) &= \llbracket \psi \rrbracket(\llbracket \neg\phi \rrbracket(\{(a, g^a, l^a)\})) \\
&= \llbracket \psi \rrbracket(\emptyset) \\
&= \emptyset.
\end{aligned}$$

This establishes the claim.

$\exists v\phi \land \psi, a$. Case 1. v not in the domain of a and v not free in ψ. Then the computation tree is like that for $\phi \land \psi$, and we can apply the induction hypothesis. This gives:

$$y(\llbracket \exists v\phi \land \psi \rrbracket_a) = y(\llbracket \phi \land \psi \rrbracket_a) \sqsubseteq ([\{\phi \land \psi\}](a))^\natural.$$

Since v not in the domain of a, we get:

$$[\{\phi \land \psi\}](a, g^a, l^a) = [\{\exists v\phi \land \psi\}](a, g^a, l^a),$$

and we are done.

$\exists v\phi \land \psi, a$. Case 2. v in the domain of a or v free in ψ. Then (with a slight amendment, see above) the computation tree looks like this:

$$\begin{array}{c}
\exists v\phi \land \psi, a \\
| \\
\mathit{error}
\end{array}$$

Thus, $y(\llbracket \exists v\phi \land \psi \rrbracket_a) = \{\bullet\}$ and there is nothing to prove. □

It should be noted that the proof of Theorem 36 does not depend on our amendment of the operational semantics. Also, if is easy to check that the proof goes through for the liberalized treatment of negation and implication proposed in the Appendix of [1].

By combining our results we can finally relate the operational semantics to the standard denotational semantics for dynamic predicate logic:

Theorem 37 (Operational Semantics Faithful to DPL) *For every formula ϕ, every finite $a \in \mathcal{A}$:*

$$(y(\llbracket \phi \rrbracket_a)^\circ \sqsubseteq' \llbracket(\!(\phi)\!)\rrbracket(\{a\}^\circ).$$

Proof. Immediate from Theorems 31 and 36, using the fact that $^\circ$ is an order preserving map (Theorem 25). □

What our final theorem tells us is that if you extend a partial valuation b computed by the operational semantics for a partial input valuation a to a full valuation b', then there is an extension a' of a for which the standard dynamic semantics yields b' as output. Also, if the operational semantics yields a failed tree for a, then the standard dynamic semantics will yield no outputs for any full valuation a' that extends a.

7 Conclusion and Future Work

Note that we have not proved completeness of our process interpretation of first order logic with respect to DPL. In fact, we feel that completeness *per se* is a misnomer for computation procedures that are free to 'give up' on a computation: the procedure that gives up on *any* formula would be trivially sound and complete. We propose to use 'correctness' or 'faithfulness' for the conjunction of 'all output computations are correct' and 'failure correctly indicates that there are no successful outputs'. A further comparison of correct computation procedures can then reveal that one procedure is 'more complete than' or 'a proper extension of' another. Our investigation has revealed that the operational semantics and the executable process interpretation are both faithful to the dynamic semantics of first order logic, but the executable process interpretation is more complete than the operational semantics. This perspective has the merit of putting the proper further questions on the research agenda, namely: a characterization of the fragments of first order logic for which the various dynamic computation procedures (i.e., the procedures that are faithful to the DPL interpretation of first order logic) do not generate errors.

Another pressing item for future research is to extend the computation mechanism to deal with atoms $Pt_1 \cdots t_n$ that are not closed for the input valuations and with identities $t_1 \doteq t_2$ that are neither closed for the input nor input-assignments. To give an example, under the present computation mechanism the identity $y = y$ effects a transition to $\bullet$ for all input valuations a with $y^a =\uparrow$. Still, we know that any extension of a with a value for y will make $y = y$ into a test that trivially succeeds. In follow-on research to this paper, we are developing a computation mechanism that collects those atoms that cannot be evaluated or used as assignments on a stack, to be dealt with later, at the first point where they have become closed for the increased input valuation, or have become assignments for the increased input valuation. This will drastically reduce the generation of transitions to $\bullet$, and will e.g. ensure that example programs (4), (8) and (9) compute the same answers.

One final point remains. Our Theorem 37 shows that the operational semantics for predicate logic proposed by Apt and Bezem is a faithful approximation of dynamic predicate logic. Apt and Bezem [1] prove a *different* faithfulness result for their operational semantics: they show that it is faithful to standard first order logic. How do these two different results square? The answer is that for the cases where $[\![\phi]\!]_a$ is determined one can show that the classical reading and the dynamic reading of ϕ coincide. This is just a matter of checking that for all ϕ, a with $[\![\phi]\!]_a$ determined it holds that $\phi^{\triangleright} \equiv \phi$. But it will be clear from the above that the game of programming with dynamic predicate logic only begins in earnest when one is prepared to grant the existential quantifier its true dynamic force.

Acknowledgements

I wish to thank Krzysztof Apt, Johan van Benthem, Marc Bezem, Peter van Emde Boas, Hans Kamp, Reinhard Muskens, Marc Pauly, Maarten de Rijke and Albert Visser for stimulating interactions about the issues raised in this paper.

References

[1] K.R. Apt and M. Bezem. Formulas as programs. In K.R. Apt, V. Marek, M. Truszczyski, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective.* Springer Verlag, 1998. to appear.

[2] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM Toplas*, 1998. in press.

[3] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 1997. Second edition (first edition appeared in 1991).

[4] J.W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall, London, 1980.

[5] J. Barwise. Noun phrases, generalized quantifiers and anaphora. In P. Gärdenfors, editor, *Generalized Quantifiers: linguistic and logical approaches*, pages 1–30. Reidel, Dordrecht, 1987.

[6] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, 1990.

[7] E.W Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[8] J. van Eijck. Axiomatizing dynamic predicate logic with quantified dynamic logic. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 30–48. MIT Press, Cambridge Mass, 1994.

[9] J. van Eijck and F.J. de Vries. Dynamic interpretation and Hoare deduction. *Journal of Logic, Language, and Information*, 1:1–44, 1992.

[10] R. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proceedings of Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science*, pages 19–32, New York, 1967. American Mathematical Society.

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

[12] P. Staudacher. Zur Semantik indefiniter Nominalphrasen. In B. Asbach-Schnitker and J. Roggenhofer, editors, *Neue Forschungen zur Wortbildung und Historiographie der Linguistik*, pages 239–258. Gunter Narr Verlag, 1987.

[13] A. Visser. Prolegomena to the definition of dynamic predicate logic with local assignments. Technical Report 178, Utrecht Research Institute for Philosophy, October 1997.