Narratives as Programs

Ray Reiter
Department of Computer Science
University of Toronto
Toronto, Canada
M5S 3G4

reiter@cs.toronto.edu

Abstract

Representing narratives, and reasoning about them, has been a prominent theme in logical formalisms for dynamical systems (e.g. [9, 5, 2]). However, the existing literature provides a rather limited concept of what a narrative is; the examples all concern linear action sequences, sometimes including incomplete information about action occurrence times. The point of departure of this paper is the observation that narratives often include more complex constructions, including nondeterminism, loops, and recursive procedures. Therefore, we propose that, in their full generality, narratives are best viewed as *programs*. In many cases, the situation calculus-based programming language GOLOG is suitable for this purpose. In this setting, we define what it means to query a narrative, and discover that this task is formally identical to proving the correctness of programs as studied in computer science. Since this general task is hopelessly intractable, we focus on procedure-free narrative programs and for a wide class of such programs and queries we prove that a regression-based approach to query evaluation is correct. Finally, we describe a Prolog implementation of these ideas.

1 Introduction

Current logical theories of actions can be divided into two broad categories: *narrative-based* approaches (e.g. features and fluents [17], the event calculus and its relatives [18, 6, 1]) that rely on linear temporal logics, and *situation-based* approaches (e.g. the families of action languages [3], various dialects of the situation calculus [8, 13, 9]) that provide for branching futures. As Reiter [15] observed, these two classes of action theories require quite different reasoning mechanisms in their treatment of planning problems: abduction for the former, deduction for the latter. Similarly, they differ in how they represent and reason about narratives – stories about action occurrences, including possibly their temporal properties. Narrative-based approaches treat narratives in a clean and straightforward way; reasoning consists of logical deduction using a knowledge base of assertions about a narrative's action occurrences and their temporal relations. The situation-based approaches suffer in comparison, at least judging by the existing literature; they require more elaborate mechanisms, usually extending their basic ontology to include the concept of an action occurrence, perhaps also together with an "actual path" in the tree of histories (e.g. [10, 11]; but see [9, 12] for different viewpoints). For a rich account of both narratives and hypothetical reasoning, which also requires an enriched ontology, see [5].

Regardless of the approach, the existing literature provides a rather limited concept of what a narrative is; the examples all concern linear action sequences, sometimes including incomplete information about action occurrence times. In this paper, we propose a much richer notion; specifically, we suggest that, in their full generality, narratives are best viewed as nondeterministic programs, and we investigate the suitability of the situation calculus-based programming language GOLOG for this purpose. In this setting, we define what it means to query a narrative, and discover that this task is formally identical to proving the correctness of programs as studied in computer science. Since this general task is hopelessly intractable, we focus on procedure-free narrative programs and for a wide class of such programs and queries we prove that a regression-based approach to query evaluation is correct. Finally, we describe a Prolog implementation of these ideas.

One consequence of the above narratives-as-GOLOG-programs perspective will be a cleaner account of narratives for situation-based theories of action than those of [10, 11]. Another will be a more general notion of narrative that can perhaps be profitably incorporated into narrative-based theories of actions as well.

2 Narratives as GOLOG Programs

The central intuition we wish to convey about narratives is that, in their most general forms, they are *programs*. Consider the following example stories about a blocks world, and their representations as programs:

1. Pat moved block A onto B, after which he moved C onto the table.

   $move(A, B)\,;\; moveToTable(C)$.[1]

   Here, ; stands for the standard sequence operator of conventional programming languages.

2. First, Pat moved a block onto the table, then she moved A onto it.

   $(\pi\, x).moveToTable(x)\,;\; move(A, x)$.

   Here, $(\pi\, x)$ is a nondeterministic operator that chooses an arbitrary argument of an action expression. So the above program means: nondeterministically choose an x, and for that x, move it to the table, then move A onto it.

3. First, Pat moved block A onto block B or block C – I don't know which – then he moved D onto some block on the table.

   $[move(A, B) \mid move(A, C)]\,;$
   $(\pi\, x)[onTable(x)?\,;\; move(D, x)]$.

   Here we have introduced two new operators: $A_1 \mid A_2$ means nondeterministically choose one of the actions A_1, A_2 to perform, and $\phi?$ means test the truth value of the logical expression ϕ.

4. Pat cleared all the blocks from the table by putting them on the floor.

   while $(\exists x)onTable(x)$ **do**
   $(\pi\, x).onTable(x)?\,;\; moveToFloor(x)$.

5. Pat unstacked all the towers onto the table.

   $unstackTowers$.

   Here, $unstackTowers$ is a recursive procedure:

   proc $unstackTowers$
   $(\forall x)onTable(x)? \mid$
   $(\pi\, x)moveToTable(x)\,;\; unstackTowers$
   endproc

6. Pat moved B to the table at time 10, and at some time before that, she moved a block onto A.

   $(\pi\, t)[t < 10?\,;\; (\pi\, x)move(x, A, t)]\,;$
   $moveToTable(B, 10)$.

   Here we have added a time parameter to action terms, as suggested for the situation calculus in [15]. We have cheated a bit in representing this narrative by recognizing that the second action mentioned in the natural language version actually preceded the first action mentioned. We take it as a separate (and nontrivial) problem to mechanically translate a natural language narrative into a program, especially getting the temporal precedences right, and we do not address that issue here.

7. Pat moved B onto A; then she observed that A is on some block on the table; then she moved C to B.

   $move(B, A)\,;\; [(\exists x)onTable(x) \wedge on(A, x)]?\,;$
   $move(C, B)$.

   Here we treat observational actions as test actions. Similarly, we represent narrative descriptions – It was a dark and stormy night when Pat first moved a block to the table – as test actions:

   $[dark \wedge stormy]?\,;\; (\pi\, x)moveToTable(x)$.

   The reason for this initially odd looking decision will become clear after we formalize a suitable programming language for representing narratives, and after we define the notion of querying a narrative.

The purpose of these examples is to argue that narratives are more complicated things that simple sequences of actions; they can have all the complexity of programs, including nondeterminism, loops and recursive procedures. But what kinds of programs can serve to represent narratives, and what can it mean to query a narrative viewed as a program? Our proposal will be to take the situation calculus-based programming language GOLOG as a suitable representation for narratives, and to appeal to GOLOG's semantics for the purposes of querying such narratives. We begin with a brief description of GOLOG.

2.1 GOLOG

GOLOG [7] is a language for defining complex actions in terms of a repertoire of primitive actions axiomatized in the situation calculus. It provides the standard – and some not so standard – control structures found in most Algol-like languages. We have just seen a number of examples of GOLOG programs and control structures, together with their intuitive semantics.

GOLOG's formal semantics is specified by introducing an abbreviation $Do(\delta, s, s')$, where δ is a program and s and s' are situation terms. $Do(\delta, s, s')$ is best viewed as a macro that expands into a second order situation calculus formula; moreover, *that formula says that sit-*

[1] To simplify the notation, we suppress the actors (here, Pat) in our action terms.

uation s' can be reached from situation s by executing some sequence of actions specified by δ. Note that our programs may be nondeterministic, that is, may have multiple executions terminating in different situations.

Do is defined inductively on the structure of its first argument as follows:

1. *Primitive actions:* If α is a primitive action term,
$$Do(\alpha, s, s') \stackrel{def}{=} Poss(\alpha, s) \wedge s' = do(\alpha, s).$$
2. *Test actions:* When ϕ is a situation-suppressed logical expression,
$$Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'.$$
Here, $\phi[s]$ denotes the result of restoring the situation argument s to all of the situation-suppressed fluents mentioned by ϕ.
3. *Sequence:*
$$Do(\delta_1; \delta_2, s, s') \stackrel{def}{=} (\exists s^*).Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s').$$
4. *Nondeterministic choice of two actions:*
$$Do(\delta_1 \mid \delta_2, s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$
5. *Nondeterministic choice of action arguments:*
$$Do((\pi\, x)\, \delta, s, s') \stackrel{def}{=} (\exists x)\, Do(\delta, s, s').$$
6. *Procedure calls:* For a predicate variable P whose last two arguments are the only ones of sort *situation*:
$$Do(P(t_1, \ldots, t_n), s, s') \stackrel{def}{=} P(t_1, \ldots, t_n, s, s').$$
7. *Blocks with local procedure declarations:*
$$Do(\{\textbf{proc}\ P_1\,(\vec{v}_1)\,\delta_1\ \textbf{endProc}\ ;\ \cdots\ ;\ \textbf{proc}\ P_n\,(\vec{v}_n)\,\delta_n\ \textbf{endProc}\ ;\ \delta_0\}, s, s') \stackrel{def}{=} (\forall P_1, \ldots, P_n).[\bigwedge_{i=1}^{n} (\forall s_1, s_2, \vec{v}_i).Do(\delta_i, s_1, s_2) \supset P_i(\vec{v}_i, s_1, s_2)] \supset Do(\delta_0, s, s').$$

Other control structures, e.g. conditionals and while loops, can be defined in terms of the above constructs:

$$\textbf{if}\ test\ \textbf{then}\ prog_1\ \textbf{else}\ prog_2 \stackrel{def}{=} test?\ ;\ prog_1 \mid \neg test?\ ;\ prog_2$$

To define **while** loops, first introduce a nondeterministic iteration operator *, where $program^*$ means do *program* 0 or more times:

$$program^* \stackrel{def}{=} \textbf{proc}\ P()\ true? \mid [program\,;\ P()]\ \textbf{endProc}\,;\ P()$$

Then **while** loops can be defined in terms of the * operator:

$$\textbf{while}\ test\ \textbf{do}\ program\ \textbf{endWhile} \stackrel{def}{=} [test?\ ;\ program]^*\ ;\ \neg test?$$

GOLOG programs are evaluated relative to a background theory of actions specifying a particular application domain. Specifically, if $\mathcal{D}$ is such a background action theory, and δ is a GOLOG program, then the evaluation of δ relative to $\mathcal{D}$ is defined to be the task of establishing the following entailment:

$$\mathcal{D} \models (\exists s) Do(\delta, S_0, s).$$

Any binding for the existentially quantified variable s obtained as a side effect of such a proof constitutes an execution trace of δ.

Example 2.1 Let ν be the narrative of example 3:

$$\nu = [move(A, B) \mid move(A, C)]\ ;\ (\pi\, x)[onTable(x)?\ ;\ (move(D, x)]$$

Then,

$$\begin{aligned} &Do(\nu, S_0, s) = \\ &\quad (\exists s').[Poss(move(A, B), S_0) \wedge \\ &\qquad s' = do(move(A, B), S_0) \vee \\ &\qquad Poss(move(A, C), S_0) \wedge \\ &\qquad s' = do(move(A, C), S_0)] \wedge \\ &\qquad (\exists x, s'').onTable(x, s') \wedge s' = s'' \wedge \\ &\qquad\quad Poss(move(D, x), s'') \wedge \\ &\qquad\quad s = do(move(D, x), s''). \end{aligned}$$

This is logically equivalent to the following formula, which is an example of a normal form to which $Do(\nu, S_0, s)$ can often be cast. Such normal forms will play an important role below, when we consider how to implement a system for querying narratives.

$$\begin{aligned} &Do(\nu, S_0, s) \equiv \\ &\quad (\exists x).Poss(move(A, B), S_0) \wedge \\ &\qquad onTable(x, do(move(A, B), S_0)) \wedge \\ &\qquad Poss(move(D, x), do(move(A, B), S_0)) \wedge \\ &\qquad s = do(move(D, x), do(move(A, B), S_0)) \\ &\quad \vee \\ &\quad (\exists x).Poss(move(A, C), S_0) \wedge \\ &\qquad onTable(x, do(move(A, C), S_0)) \wedge \\ &\qquad Poss(move(D, x), do(move(A, C), S_0)) \wedge \\ &\qquad s = do(move(D, x), do(move(A, C), S_0)). \end{aligned}$$

2.2 Reasoning about Narratives

A narrative is a description of action occurrences in some world. We take it that the central reasoning task for a narrative is to determine what the resulting world would be like. This task has three major components.

1. We need to infer the effects on the world of the action occurrences. This includes the actions' non-effects, so the frame problem must be taken into account. We therefore need a background, domain specific theory that characterizes action effects, and that solves the frame problem.
2. Observational actions and descriptions, as in Example 7 above, provide additional information about the world, and this must be taken into account in answering queries about a narrative.
3. Narratives convey implicit information about how the world must have been, by virtue of an action occurrence. For example, if a narrative claims that $move(A, B)$ was performed, then at the point that this action was performed, the action $move(A, B)$ must have been possible. So we can infer that, however else the world might have been, at the point of this action occurrence, both A and B must have been clear. Such information needs to be extracted from the narrative and made explicit for the purposes of answering queries about the narrative. Therefore, as part of the background theory, we require axioms specifying the action preconditions.

Finally, it is important to note that a convention about narratives is that one cannot assume more about the way the world is than is conveyed by the narrative and its immediate context. So, for example, when faced with example 1 above, one cannot, without further contextual justification, assume that A, B and C are all and only the available blocks. Neither can one assume particular initial locations for these blocks, for example, that A is initially on the table. In other words, narratives describe *open worlds*; one cannot assume complete information about the initial situation. Among other things, this observation precludes simple STRIPS-like action representations for describing and reasoning about narratives.

Definition 2.1 Query. A *query* is any situation calculus formula $Q(s)$ whose only free variable is the situation variable s.

Definition 2.2 Querying a Narrative
Let $Q(s)$ be a query, $\mathcal{D}$ a set of situation calculus axioms specifying an underlying domain of application, and ν a narrative viewed as a GOLOG program. Then the *answer to Q for the narrative ν relative to $\mathcal{D}$* is "yes" iff

$$\mathcal{D} \models (\forall s).Do(\nu, S_0, s) \supset Q(s).$$

The answer to Q is "no" iff the answer to $\neg Q$ is "yes".

Notice especially that in this form, querying a narrative is formally identical to the problem of proving properties of programs, as normally understood in computer science. We are simply asking whether all terminating situations s of the program ν have property Q. This is not good news for automating query evaluation for narratives; proving properties of programs is notoriously difficult, requiring mathematical induction for programs with loops and recursion, and it is unlikely that a general computational account can be given for this problem. For this reason, we shall limit ourselves in what follows to procedure-free programs, for the purposes of providing an implementation for this class of narratives.

3 Implementation Foundations

Here we provide the theoretical foundations for implementing and querying narratives when the underlying axioms form a basic action theory, and the GOLOG narrative program is procedure-free.

3.1 Basic Action Theories

Our concern in this paper will be with axioms for actions and their effects with a particular syntactic form ([13]).

Definition 3.1 Basic Action Theories
A *basic action theory* has the form

$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$, where:

- Σ are the *foundational axioms for situations* [13]. These play no role in this paper and we omit them.
- $\mathcal{D}_{ap}$ is a set of *action precondition axioms*, one for each action function $A(\vec{x})$, of the form

$$Poss(A(\vec{x}, s) \equiv \Pi_A(\vec{x}, s). \quad (1)$$

  Here, $\Pi_A(\vec{x}, s)$ is a formula whose free variables are among $\vec{x}, s$, it does not quantify over situations, nor does it mention the predicate symbol $Poss$, and the only term of sort *situation* that it mentions is s.
- $\mathcal{D}_{ss}$ is a set of *successor state axioms*, one for each fluent $F(\vec{x}, s)$. These have the syntactic form

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s), \quad (2)$$

  where $\Phi_F(\vec{x}, a, s)$ is a formula, all of whose free variables are among $a, s, \vec{x}$, it does not quantify over situations, nor does it mention the predicate symbol $Poss$, and the only term of sort *situation*

that it mentions is s. Such axioms embody a solution to the frame problem for deterministic actions [14].[2]

- $\mathcal{D}_{una}$ is the set of *unique names axioms* for all action function symbols.
- $\mathcal{D}_{S_0}$, the *initial database*, is a set of first order sentences such that no sentence of $\mathcal{D}_{S_0}$ quantifies over situations, or mentions $Poss$, and S_0 is the only term of sort *situation* mentioned by these sentences. $\mathcal{D}_{S_0}$ will function as the initial theory of the world.

Example 3.1 The following are the successor state and action precondition axioms for a blocks world basic action theory used in the implementation described below.

Action Precondition Axioms

$$Poss(move(x,y),s) \equiv clear(x,s) \wedge clear(y,s) \wedge x \neq y,$$

$$Poss(moveToTable(x),s) \equiv clear(x,s) \wedge \neg onTable(x,s).$$

Successor State Axioms

$$clear(x, do(a,s)) \equiv (\exists y)\{[(\exists z)a = move(y,z) \vee a = moveToTable(y)] \wedge on(y,x,s)\} \vee clear(x,s) \wedge \neg(\exists y)a = move(y,x),$$

$$on(x,y,do(a,s)) \equiv a = move(x,y) \vee on(x,y,s) \wedge a \neq moveToTable(x) \wedge \neg(\exists z)a = move(x,z),$$

$$onTable(x, do(a,s)) \equiv a = moveToTable(x) \vee onTable(x,s) \wedge \neg(\exists y)a = move(x,y).$$

Unique Names Axioms for Actions

$$move(x,y) \neq moveToTable(z),$$

$$move(x,y) = move(x',y') \supset x = x' \wedge y = y',$$

$$moveToTable(x) = moveToTable(x') \supset x = x'.$$

3.2 Regression for Basic Action Theories

Regression [13] is perhaps the single most important theorem-proving mechanism for the situation calculus; it provides a systematic way to establish that a basic action theory entails a so-called *regressable sentence*.

Definition 3.2 Concrete Situation Terms. These are inductively defined by: S_0 is a concrete situation term; when σ is a concrete situation term and α is an action term, then $do(\alpha,\sigma)$ is a concrete situation term.

[2] One can also give successor state axioms for functional fluents; because of space limitations, we do not discuss these here.

Definition 3.3 The Regressable Formulas. A regressable formula of the situation calculus is a first order formula W with the property that every situation term mentioned by W is concrete, and moreover, for every atom of the form $Poss(\alpha,\sigma)$ mentioned by W, α has the form $A(t_1,\ldots,t_n)$ for some n-ary action function symbol A and terms $t_1,\ldots,t_n$.

The essence of a regressable formula is that each of its *situation* terms is rooted at S_0, and therefore, one can tell, by inspection of such a term, exactly how many actions it involves. It is not necessary to be able to tell what those actions are, just how many they are. In addition, when a regressable formula mentions a $Poss$ atom, we can tell, by inspection of that atom, exactly what is the action function symbol occurring in its first argument position, for example, that it is a *move* action.

The intuition underlying regression is this: Suppose we want to prove that a regressable sentence W is entailed by a basic action theory. Suppose further that W mentions a relational fluent atom $F(\vec{t}, do(\alpha,\sigma))$, where F's successor state axiom is (2). By substituting $\Phi_F(\vec{t},\alpha,\sigma)$ for $F(\vec{t},do(\alpha,\sigma))$ in W we obtain a logically equivalent sentence W'. After we do so, the fluent atom $F(\vec{t},do(\alpha,\sigma))$, involving the complex situation term $do(\alpha,\sigma)$, has been eliminated from W in favour of $\Phi_F(\vec{t},\alpha,\sigma)$, and this involves the simpler situation term σ. In this sense, W' is "closer" to the initial situation S_0 than was W. Similarly, if W mentions an atom of the form $Poss(A(t_1,\ldots,t_n),\sigma)$,[3] there will be an action precondition axiom for A of the form (1) so we can eliminate this atom by replacing it with $\Pi_A(t_1,\cdots,t_n,\sigma)$. This process of replacing $Poss$ and fluent atoms by the right hand sides of their action precondition and successor state axioms can be repeated until the resulting goal formula mentions only the situation term S_0. Regression is a mechanism that repeatedly performs the above reduction to a goal W, ultimately obtaining a logically equivalent goal W_0 whose only situation term is S_0. See [13] for precise definitions, and for a proof of the soundness and completeness of regression for basic action theories.

3.3 Procedure-Free Narratives

Definition 3.4 Choice Prenex Form
A GOLOG program is in *choice prenex form* iff it has the form

$$(\pi\,\vec{x}_1)P_1 \mid \cdots \mid (\pi\,\vec{x}_m)P_m,$$

where each P_i is of the form $\alpha_{i_1}\,;\,\cdots\,;\,\alpha_{i_k}, k \geq 1$, each

[3] Because W is assumed to be regressable, all $Poss$ atoms mentioned by W must be of this form.

α is a primitive action or a test action, and association of the sequence operators in P_i is to the right.

Definition 3.5 Equivalence of Programs
Two GOLOG programs P_1 and P_2 are *equivalent* iff the following sentence is valid:

$$(\forall s, s').Do(P_1, s, s') \equiv Do(P_2, s, s').$$

Definition 3.6 Procedure-Free Programs
These are GOLOG programs defined without the procedure mechanism, i.e. using only primitive actions, tests (?), sequence (;) and nondeterministic choice (π and $|$).

Lemma 3.1 *There is an effective procedure for transforming a procedure-free GOLOG program into an equivalent program in choice prenex form.*

Proof: By repeatedly applying the following equivalence preserving transformations, until no further reductions are possible:[4]

$$\begin{array}{l}
P_1 \,;\, (P_2 \mid P_3) \rightarrow P_1 \,;\, P_2 \mid P_1 \,;\, P_3, \\
(P_1 \mid P_2) \,;\, P_3 \rightarrow P_1 \,;\, P_3 \mid P_2 \,;\, P_3, \\
(P_1 \,;\, P_2) \,;\, P_3 \rightarrow P_1 \,;\, (P_2 \,;\, P_3), \\
(\pi\, x)[P_1 \mid P_2] \rightarrow (\pi\, x)P_1 \mid (\pi\, x)P_2, \\
[(\pi\, x)P_1] \,;\, P_2 \rightarrow (\pi\, x')[P_1|^{x}_{x'} \,;\, P_2], \\
P_1 \,;\, [(\pi\, x)P_2] \rightarrow (\pi\, x')[P_1 \,;\, P_2|^{x}_{x'}].
\end{array}$$

Here, x' is a new variable, distinct from any variable mentioned in P_1 or P_2. $P|^{x}_{x'}$ denotes the result of substituting x' for all free occurrences of x in P.
■

For programs δ that are specifically in choice prenex form, define a ternary relation $DoCpf(\delta, s, s')$ as follows:

1. *Primitive actions:* If α is a primitive action term,

$$DoCpf(\alpha, s, s') \stackrel{def}{=} Poss(\alpha, s) \wedge s' = do(\alpha, s).$$

$$DoCpf(\alpha \,;\, \delta, s, s') \stackrel{def}{=} Poss(\alpha, s) \wedge DoCpf(\delta, do(\alpha, s), s').$$

2. *Test actions:* When ϕ is a situation-suppressed logical expression,

$$DoCpf(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s' = s.$$

$$DoCpf(\phi? \,;\, \delta, s, s') \stackrel{def}{=} \phi[s] \wedge DoCpf(\delta, s, s').$$

3. *Nondeterministic choice of two actions:*

$$DoCpf(\delta_1 \mid \delta_2, s, s') \stackrel{def}{=} DoCpf(\delta_1, s, s') \vee DoCpf(\delta_2, s, s').$$

4. *Nondeterministic choice of action arguments:*

$$DoCpf((\pi\, x)\, \delta, s, s') \stackrel{def}{=} (\exists x)\, DoCpf(\delta, s, s').$$

Lemma 3.2 *Let ν be a procedure-free GOLOG program that does not mention a choice operator $(\pi\, a)$ where a is a variable of sort* action.[5] *Let κ be a choice prenex form for ν. Then*

1. *$(\forall s).Do(\nu, S_0, s) \equiv DoCpf(\kappa, S_0, s)$ is valid.*
2. *$DoCpf(\kappa, S_0, s)$ has the syntactic form:*

$$(\exists \vec{x}_1)[C_1 \wedge s = \sigma_1] \vee \cdots \vee (\exists \vec{x}_n)[C_n \wedge s = \sigma_n], \quad (3)$$

where each σ_i is a concrete situation term and each C_i is a regressable formula.

Proof: By Lemma 3.1, ν is equivalent to κ, and therefore, $(\forall s).Do(\nu, S_0, s) \equiv Do(\kappa, S_0, s)$ is valid.

Suppose

$$\kappa = (\pi\, \vec{x}_1)P_1 \mid \cdots \mid (\pi\, \vec{x}_m)P_m.$$

Then by the expansion rules for GOLOG programs of Section 2.1,

$$Do(\kappa, S_0, s) = (\exists\, \vec{x}_1)Do(P_1, S_0, s) \vee \cdots \vee (\exists\, \vec{x}_m)Do(P_m, S_0, s).$$

Similarly, by the rules for expanding $DoCpf$,

$$DoCpf(\kappa, S_0, s) = (\exists\, \vec{x}_1)DoCpf(P_1, S_0, s) \vee \cdots \vee (\exists\, \vec{x}_m)DoCpf(P_m, S_0, s).$$

Accordingly, it is sufficient to prove the following:

Suppose that a program P has the form $\alpha_1 \,;\, \cdots \,;\, \alpha_k$, where each α is a primitive action term or a test action, where the sequence operator associates to the right, and where P mentions free variables $\vec{x}$, none of which is of sort action. *Suppose further that S is a concrete situation term, possibly with free variables among $\vec{x}$. Then $(\forall \vec{x}, s).Do(P, S, s) \equiv DoCpf(P, S, s)$ is valid. Moreover, $DoCpf(P, S, s)$ is a formula of the form $C \wedge s = \sigma$, where σ is a concrete situation term, and C is a regressable formula.*

We prove this by induction on k.

Base case: P is just α, where α is a test action $\phi?$ or α is a primitive action term. In the first case,

$$Do(\alpha, S, s) = \phi[S] \wedge s = S = DoCpf(\alpha, S, s).$$

Now S is a concrete situation term. Moreover, because no free variable of P is an action variable, if

[4] Strictly speaking, we should prove that these are equivalence preserving transformations. The proofs are straightforward, and we omit the details.

[5] Of course, the program may mention a choice operator $(\pi\, x)$ where x ranges over domain objects other than actions and situations, as in the examples of Section 2.

$\phi[S]$ mentions an atom of the form $Poss(\beta, S)$, then β cannot be an action variable, and therefore must be of the form $A(t_1, \ldots, t_n)$ for some n-ary action function symbol A and terms $t_1, \ldots, t_n$. Therefore, $\phi[S]$ is regressable, and we have established the base case when α is a test action.

Next, we establish the base case when α is a primitive action. As before, because no free variable of P is an action variable, α must be of the form $A(t_1, \ldots, t_n)$ for some n-ary action function symbol A and terms $t_1, \ldots, t_n$. Moreover,

$$\begin{aligned} Do(\alpha, S, s) &= Poss(\alpha, S) \wedge s = do(\alpha, S) \\ &= DoCpf(\alpha, S, s). \end{aligned}$$

Since α is of the form $A(t_1, \ldots, t_n)$, and since S is concrete, $Poss(\alpha, S)$ is regressable and $do(\alpha, S)$ is concrete. This establishes the remaining base case.

Induction step: Assume the result for $\alpha_1 \ ; \ \cdots \ ; \ \alpha_k$, $k \geq 1$, and suppose that α is a primitive action term. Then

$$\begin{aligned} &Do(\alpha \ ; \ \alpha_1 \ ; \ \cdots \ ; \ \alpha_k, S, s) = \\ &(\exists s^*)[Poss(\alpha, S) \wedge s^* = do(\alpha, S) \ \wedge \\ &\qquad Do(\alpha_1 \ ; \ \cdots \ ; \ \alpha_k, s^*, s)] \equiv \\ &Poss(\alpha, S) \wedge Do(\alpha_1 \ ; \ \cdots \ ; \ \alpha_k, do(\alpha, S), s). \end{aligned}$$

Now $do(\alpha, S)$ is concrete. Moreover, by the same argument as in the base case, $Poss(\alpha, S)$ is regressable. Finally, by induction hypothesis,

$$\begin{aligned} &Do(\alpha_1 \ ; \ \cdots \ ; \ \alpha_k, do(\alpha, S), s) \equiv \\ &\qquad DoCpf(\alpha_1 \ ; \ \cdots \ ; \ \alpha_k, do(\alpha, S), s), \end{aligned}$$

and $DoCpf(\alpha_1 \ ; \ \cdots \ ; \ \alpha_k, d(\alpha, S), s)$ is a formula of the form $C \wedge s = \sigma$, where C is regressible and σ is concrete. The result now follows.
The case where α is a test action is similar.
■

Example 3.2 With reference to Example 2.1, the last formula displayed is the one promised by the lemma.

Theorem 3.1 *Suppose that $\mathcal{D}$ is a basic action theory and that the program ν satisfies the conditions of Lemma 3.2. Suppose further that $Q(s)$ is a query with the property that $Q(\sigma)$ is regressable whenever σ is a concrete situation term. Without loss of generality, assume that the bound variables (if any) of $Q(s)$ are distinct from all of the variables $\vec{x}_i$ of (3). Then, with reference to Lemma 3.2,*

$$\mathcal{D} \models (\forall s).Do(\nu, S_0, s) \supset Q(s) \text{ iff for } i = 1, \ldots, n$$
$$\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models (\forall \vec{x}_i).\mathcal{R}[C_i] \supset \mathcal{R}[Q(\sigma_i)].$$

Moreover, $Q(\sigma_i)$ is a regressable formula. Here, $\mathcal{R}$ is the regression operator.

Proof: Let κ be a choice prenex form for ν. Then
$\mathcal{D} \models (\forall s).Do(\nu, S_0, s) \supset Q(s)$
iff, by Lemma 3.2,
$\mathcal{D} \models (\forall s).DoCpf(\kappa, S_0, s) \supset Q(s)$
iff, again by Lemma 3.2,

$$\begin{aligned} &\mathcal{D} \models \\ &\quad (\forall s).(\exists \vec{x}_1)[C_1 \wedge s = \sigma_1] \vee \cdots \vee (\exists \vec{x}_n)[C_n \wedge s = \sigma_n] \\ &\qquad \supset Q(s) \end{aligned}$$

iff, for $i = 1, \ldots, n$,
$\mathcal{D} \models (\forall s).(\exists \vec{x}_i)[C_i \wedge s = \sigma_i] \supset Q(s)$
iff, because the bound variables of Q are different then those of $\vec{x}_i$, and by properties of equality in first order logic,
$\mathcal{D} \models (\forall \vec{x}_i).C_i \supset Q(\sigma_i)$.
By hypothesis, $Q(\sigma_i)$ is regressable because σ_i is concrete, and C_i is regressable by Lemma 3.2. Therefore, by the Regression Theorem of Pirri and Reiter [13],
$\mathcal{D} \models (\forall \vec{x}_i).C_i \supset Q(\sigma_i)$
iff
$\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models (\forall \vec{x}_i).\mathcal{R}[C_i] \supset \mathcal{R}[Q(\sigma_i)]$.
■

This is our central computational result. Under suitable conditions on the program and query, narrative query evaluation can be done using regression; moreover, after performing the regression steps, all theorem proving is *relative only to the initial database and unique names axioms for actions.* The foundational axioms for the situation calculus, and the action precondition and successor state axioms are not required (although, of course, these last two are used in the regression steps).

4 An Implementation

We have implemented (in Eclipse Prolog) a narrative compiler and query evaluator based on Theorem 3.1 for procedure-free GOLOG programs, and we briefly describe it here.[6]

Given basic action theory $\mathcal{D}$, query Q and narrative ν, our task is to establish the entailment $\mathcal{D} \models (\forall s).Do(\nu, S_0, s) \supset Q(s)$. By Theorem 3.1, this is equivalent to establishing, for $i = 1, \ldots, n$, that $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models (\forall \vec{x}_i).\mathcal{R}[C_i] \supset \mathcal{R}[Q(\sigma_i)]$. Skolemize the leading universal quantifiers to obtain the following equivalent theorem proving task:

[6] The full program, together with supporting code for narratives about the blocks world based on the axioms of Example 3.1, is available on request from the author.

$\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[C_i^{sk}] \supset \mathcal{R}[Q(\sigma_i^{sk})]$ for $i = 1, \ldots, n$.

Here, C_i^{sk} and σ_i^{sk} are the results of substituting distinct, fresh Skolem constants for the free occurrences of $\vec{x}_i$ in C_i and σ_i, respectively.[7] Finally, this theorem proving task is equivalent to establishing that

$$\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \{\mathcal{R}[C_i^{sk}]\} \models \mathcal{R}[Q(\sigma_i^{sk})] \text{ for } i = 1, \ldots, n. \quad (4)$$

The implementation has three components: a *narrative compiler*, a *query processor*, and a *theorem prover*.

4.1 The Narrative Compiler

This accepts a narrative program ν as described in Lemma 3.2, and performs the following steps:

1. Convert ν to its choice prenex form κ by applying the transformations specified in the proof of Lemma 3.1. Then determine $DoCpf(\kappa, S_0, s)$ to obtain the sentence (3).
2. Replace the variables $\vec{x}_i$ mentioned by C_i and σ_i in (3) by Skolem constants, yielding C_i^{sk} and σ_i^{sk} as in (4).
3. Regress C_i^{sk}, then convert this to clausal form.
4. Transform the sentences of $\mathcal{D}_{S_0}$ to clausal form. (The unique names axioms are not converted to clausal form; instead, these are represented by suitable simplification rules used by the regression routine and the theorem-prover described below.) Typically, $\mathcal{D}_{S_0}$ will include all of the domain's state constraints, relativized to S_0. For the blocks world, these consist of:

$$(\forall x, y).on(x, y, S_0) \supset \neg on(y, x, S_0),$$
$$(\forall x, y, z).on(x, y, S_0) \wedge on(x, z, S_0) \supset y = z,$$
$$(\forall x, y, z).on(x, z, S_0) \wedge on(y, z, S_0) \supset x = y.$$

The end result of these four steps is n distinct databases of clauses.

4.2 The Query Processor

For $i = 1, \ldots, n$ the query processor takes the query $Q(s)$, substitutes σ_i^{sk} for s as in (4), regresses $\neg Q(\sigma_i^{sk})$, converts this to clausal form, and adds the resulting clauses to the i-th database created by the narrative compiler. To establish the entailment (4), each of the n resulting sets of clauses must be shown to be unsatisfiable, and this computation is performed by the theorem-prover.

4.3 The Theorem-Prover

This is a relatively unsophisticated, incomplete unit resolution-based system, using subsumption for clause elimination. It also incorporates a limited form of equality reasoning.[8] The theorem-prover works in two passes:

1. It first tries a pure unit resolution refutation. In doing so, it takes equality into account as follows: Whenever it derives a ground unit clause of the form $t_1 = t_2$, it uniformly substitutes t_2 for t_1 throughout the current set of clauses, and also performs routine simplifications like replacing atoms of the form $t = t$ by $true$ and $\neg t = t$ by $false$. Should it find a refutation this way, it exits with success.
2. Otherwise, the theorem-prover tries a little bit harder by performing a case analysis. It does this by repeatedly selecting and splitting a non-unit clause with at most one non-ground literal, and attempting a unit resolution refutation as in 1 for each of the cases. If this case analysis succeeds for some splittable clause, it exits with success; else it gives up.

4.4 An Example Program Execution

The following is the output obtained for compiling the ongoing narrative of Example 2.1, and issuing the query

$$Q(s) = (\exists x).onTable(x, s) \wedge (\exists y).on(A, y, s) \wedge x \neq y.$$

Example: Compiling and Querying a Narrative.[9]

```
[eclipse 2]: compile((move(a,b) # move(a,c)) :
                pi(x,?(onTable(x)) : move(d,x))).

Time (sec): 0.06

yes.
[eclipse 3]: prove(some(x,onTable(x) &
                    some(y,on(a,y) & -(x = y)))).
```

[7] Since s is the only free variable mentioned by a query $Q(s)$, and since the bound variables of $Q(s)$ are assumed distinct from the $\vec{x}_i$ (see statement of Theorem 3.1), the only way that $Q(\sigma_i)$ can mention one or more of the variables $\vec{x}_i$ is if σ_i mentions these variables.

[8] We did not incorporate time into the implementation, as would be needed for examples like 6 of Section 2. This would require special purpose temporal reasoning mechanisms in the theorem-prover, and our primary objective here was only to demonstrate the basic feasibility of our approach.

[9] CPU times here are for a SUN Enterprise (Ultra) 450, with four 400MHZ processors and 4GB of RAM.

```
Case: do(move(d, sk(13)), do(move(a, c), s0))

Unit resolution fails. Trying harder...

Splitting on [sk(13) = d, sk(13) = c]

  Succeeds.

Case: do(move(d, sk(12)), do(move(a, b), s0))

Unit resolution fails. Trying harder...

Splitting on [sk(12) = d, sk(12) = b]

  Succeeds.

*** Proof succeeds ***

Time (sec): 0.17

yes.
```

5 Discussion

We have so far avoided discussing concurrency, as would be required, for example, by even a simple narrative like: Pat moved blocks A and B to the table. One way to represent this as a program is with interleaving:

$$moveToTable(A)\,;\, moveToTable(B)\,|$$
$$moveToTable(B)\,;\, moveToTable(A).$$

But this treatment of the *moveToTable* action is too course grained; it precludes the possibility of the two actions overlapping. A finer grained representation can be obtained by introducing *process fluents* (moving a block to the table) and two instantaneous actions, one to initiate the process, one to terminate it (See the discussion in [16]). With this representational device in hand, we can write a GOLOG program that describes all the possible interleavings of the two move actions (no overlap with A preceding/following B; partial overlap with A starting first, then B starting, then A ending, then B ending; total inclusion, with B starting, then A starting, then A ending, then B ending; etc). The combinatorics are bad enough, even for this simple example; they quickly get out of hand for more interesting examples like McCarthy's Junior-goes-to-Moscow [9]. They become impossible when the temporal ordering between *programs* is underspecified by a narrative; then we need an account for the concurrent execution of arbitrary GOLOG programs. This is exactly what the programming language CONGOLOG provides [4], so to accommodate concurrency, we can generalize our narratives-as-programs viewpoint by representing narratives as CONGOLOG programs. To define the result of querying a narrative, we appeal to CONGOLOG's $Trans^*$ predicate in Definition 2.2 instead of GOLOG's Do relation. Querying a narrative now becomes formally identical to proving properties of concurrent programs. These are all issues that have yet to be explored.

Our theoretical and implementation results are for procedure-free programs. It would not be difficult to incorporate non-recursive procedures into our account, by simply "unfolding" procedure calls. But of course, this would make sense only when this unfolding is guaranteed to terminate, namely, when there are no recursive calls. Since the definition of while loops of Section 2.1 requires recursion, our approach cannot deal with loops either. Providing computational foundations for loops and recursive procedures requires reasoning about program *postconditions*, and it appears that regression is not a suitable mechanism for this.

The cardinal sin of omission of this paper is to have glossed over how one translates a natural language narrative into a program. This raises a variety of issues in nonmonotonic reasoning concerned with minimizing action occurrences (e.g. [18]), but exactly how these techniques might be adapted to the automatic generation of programs is a completely open problem. Incidentally, this is not a consequence of our commitment to the situation calculus; any action logic adopting our view that narratives are programs must confront this problem.

References

[1] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[2] C. Baral, A. Gabaldon, and A. Provetti. Formalizing narratives using nested circumscription. *Artificial Intelligence*, 104:107–164, 1998.

[3] M. Gelfond and V. Lifschitz. Action languages. *Linköping Electronic Articles in Computer and Information Sciences*, 3, 1998. www.ep.liu.se/ea/cis/1998/016/.

[4] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, Nagoya, Japan, 1997.

[5] L. Karlsson. Anything can happen: On narratives and hypothetical reasoning. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 36–47. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[6] R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:267, 1986.

[7] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: a logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[8] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[9] J. McCarthy and T. Costello. Combining narratives. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 48–59. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[10] R. Miller and M. Shanahan. Narratives in the situation calculus. *The Journal of Logic and Computation (Special Issue on Actions and Processes)*, 4:513–530, 1994.

[11] J.A. Pinto and R. Reiter. Reasoning about time in the situation calculus. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):251–268, September 1995.

[12] Javier Pinto. Occurrences and narratives as constraints in the branching structure of the situation calculus. *Journal of Logic and Computation*, 8(6):777–808, 1998.

[13] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999.

[14] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[15] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L.C. Aiello, J. Doyle, and S.C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 2–13. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[16] R. Reiter. Sequential, temporal GOLOG. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 547–556. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[17] E. Sandewall. *Features and Fluents: The Representation of Knowledge about Dynamical Systems.* Oxford University Press, 1994.

[18] M.P. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.