Technical Details of Dynamic Logic Programming

Michael Genesereth
Computer Science Department
Stanford University

1. Introduction

Dynamic Logic Programming (DLP) combines the strengths of safe, stratified, side-effect-free logic programming in defining concepts with the power of transition rules for prescribing behavior. Because concept definitions in DLP are safe and stratified and side-effect-free, dynamic logic programs are simpler than general logic programs and they allow for more efficient implementation. At the same time, the possibility of writing transition rules adds expressive power without compromising the simplicity of logic programming.

In Dynamic Logic Programming, the states of the application environment are modeled as instances of a database (here called *datasets*), and additional information is expressed in the form of *rules* that can be applied to these instances. *View rules* define *views* of datasets, and *transition rules* specify how the database changes over time.

This short paper provides technical detail about the syntax and semantics of Dynamic Logic Programming. Section 2 describes the concept of datasets; Section 3 gives details of view rules; and Section 4 covers transition rules and shows how they are used in formalizing dynamic behavior.

2. Datasets

A *vocabulary* is a collection of *object constants*, *function constants*, and *relation constants*. Each function constant and relation constant has an associated *arity*, i.e. the number of *arguments* allowed in any expression involving that constant.

A *ground term* is either an object constant or a functional term (as defined in the next paragraph).

A *ground functional term* is an expression formed from an *n*-ary function constant and *n* ground terms. In this paper, we write functional terms in traditional mathematical notation - the function constant followed by its *arguments* enclosed in parentheses and separated by commas. For example, if `f` is a unary function constant and `a` is an object constant, then `f(a)`, `f(f(a))`, and `f(f(f(a)))` are all ground functional terms.

A *ground atom* (or *factoid*) is an expression formed from an *n*-ary relation constant and *n* ground terms. In analogy with functional terms, we write factoids in traditional mathematical notation - the relation constant followed by its *arguments* enclosed in parentheses and separated by commas.

For example, if `r` is a binary relation constant and `a` and `b` are object constants, then `r(a,b)` is a factoid.

The *Herbrand universe* for a given vocabulary is the set of all ground terms that can be formed from the constants in the vocabulary. In the absence of function constants, the Herbrand universe for a vocabulary is just the set of all object constants. In the presence of functional constants with arity greater than 0, the Herbrand universe is necessarily infinite, as it includes not just object constants but also functional terms nested arbitrarily deeply.

The *Herbrand base* for a database is the set of all factoids that can be formed from the constants in its vocabulary. For example, for a vocabulary with just two object constants `a` and `b` and a single binary relation constant `r`, the Herbrand base is {`r(a,a)`, `r(a,b)`, `r(b,a)`, `r(b,b)`}.

A *dataset* is any subset of the Herbrand base, i.e. an arbitrary set of the factoids that can be formed from the vocabulary of the database. The factoids in a dataset representing a state are typically assumed to be true in that state, and all other factoids in the Herbrand base are typically assumed to be false.

3. View Definitions

A basic logic program is a set of rules that define new relations in terms of existing relations. Such view definitions take the form of Prolog-like rules with the constraint that the rules are safe and stratified and side-effect-free.

The vocabulary of a basic logic program is a superset of the vocabulary of any dataset to which it is applied. It includes the object, function, and relation constants used in the dataset, but it can include additional object, function, and relation constants as well.

Basic logic programs can also include a new type of symbol, called a *variable*. Variables allow us to state relationships among objects without naming specific objects. In what follows, we write variables as strings of alphanumeric characters beginning with a capital letter, e.g. `X`, `Y`, `Z`, `Mass`, `Speed`, and so forth.

Atoms are analogous to dataset factoids except that they can optionally contain variables as well as object constants. For example, if `r` is a binary relation constant, if `a` is an object constant, and if `X` and `Y` are variables, then `r(a,X)` is an atom, as is `r(a,Y)` and `r(X,Y)` and `r(X,X)`.

A *literal* is either an atom or a negation of an atom (i.e. an expression stating that the atom is false). A simple atom is called a *positive* literal, The negation of an atom is called a *negative* literal. In what follows, we write negative literals using the negation sign ~. For example, if `p(a,b)` is an atom, then `~p(a,b)` denotes the negation of this atom.

A *rule* is an expression consisting of a distinguished atom, called the head, and zero or more atoms, together called the body. The literals in the body are called *subgoals*. In what follows, we write rules as in the example shown below.

$$\psi \text{ :- } [\sim]\phi_1 \ \& \ \ldots \ \& \ [\sim]\phi_n$$

The following expression is an example of a rule. Here, `r(X)` is the head, the expression `p(X,Y) & q(Y)` is the body; and `p(X,Y)` and `~q(Y)` are subgoals.

```
r(X) :- p(X,Y) & ~q(Y)
```

Intuitively, a rule is something like a reverse implication. It is a statement that the conclusion of the rule is true whenever the conditions are true. For example, the rule above states that `r` is true of any object `x` *if* there is an object `Y` such that `p` is true of `x` and `Y` and `q` is not true of `Y`. For example, if we know that `p(a,b)` is true and `q(b)` is false, then, using this rule, we can conclude that `r(a)` is true. See the end of this section for a more formal treatment of semantics.

A *logic program* is a set of facts and rules of the form just described. Unfortunately, the language of rules, as defined above, allows for logic programs with some unpleasant properties. To eliminate these problems, we concentrate exclusively on logic programs where the rules have two special properties, viz. safety and stratification.

A rule in a logic program is *safe* if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. A logic program is safe if and only if every rule in the program is safe.

The rule shown below is safe. Every variable in the head and every variable in the negative subgoal appears in a positive subgoal in the body. Note that it is okay for the body to contain variables that do not appear in the head.

```
r(X,Y) :- p(X,Y,Z) & ~q(X,Z)
```

By contrast, the two rules shown below are not safe. The first rule is not safe because the variable `z` appears in the head but does not appear in any positive subgoal. The second rule is not safe because the variable `z` appears in a negative subgoal but not in any positive subgoal.

```
s(X,Y,Z) :- p(X,Y)
t(X,Y) :- p(X,Y) & ~q(Y,Z)
```

(Note that this condition is stronger than necessary. We do not need every rule to be safe; we just require that the program as a whole is safe. The definition of this broader notion of safety is a little complicated and the distinction is unnecessary here, so we skip over this subtlety in the interests of simplicity.)

We say that a set of view definitions is *stratified with respect to negation* if and only if its rules can be partitioned into *strata* in such a way that (1) every stratum contains at least one rules, (2) the rules defining relations that appear in positive goals of a rule appear in the same stratum as that rule *or* in some lower stratum, and (3) the rules defining relations that appear in negative subgoals of a rule occur in some *lower* stratum (not the same stratum).

As an example, assume we have a unary relation `p` that is true of all of the objects in some application area, and assume that `q` is an arbitrary binary relation. Now, consider the ruleset shown below. The first two rules define `r` to be the transitive closure of `q`. The third rule defines `s` to be the complement of the transitive closure.

```
r(X,Y) :- q(X,Y)
r(X,Z) :- q(X,Y) & r(Y,Z)
s(X,Y) :- p(X) & p(Y) & ~r(X,Y)
```

This is a complicated ruleset, yet it is easy to see that it is stratified with respect to negation. The

first two rules contain no negations at all, and so we can group them together in our lowest stratum. The third rule has a negated subgoal containing a relation defined in our lowest stratum, and so we put it into a stratum above this one, as shown below. This ruleset satisfies the conditions of our definition and hence it is stratified with respect to negation.

Stratum	Rules
2	`s(X,Y) :- p(X) & p(Y) & ~r(X,Y)`
1	`r(X,Y) :- q(X,Y)` `r(X,Z) :- q(X,Y) & r(Y,Z)`

By comparison, consider the following ruleset. Here, the relation `s` is defined in terms of `p` and the negation of `r`, and the relation `r` is defined in terms of `p` and the negation of `s`.

```
r(X,Y) :- p(X) & p(Y) & q(X,Y)
s(X,Y) :- r(X,Y) & ~s(Y,X)
```

There is no way of dividing the rules of this ruleset into strata in a way that satisfies the definition above. Hence, the ruleset is *not* stratified with respect to negation.

The problem with unstratified rulesets is that there is a potential ambiguity. As an example, consider the rules above and assume that our dataset also included the facts `p(a)`, `p(b)`, `q(a,b)`, and `q(b,a)`. From these facts, we can conclude `r(a,b)` and `r(b,a)` are both true. So far, so good. But what can we say about `s`? If we take `s(a,b)` to be true and `s(b,a)` to be false, then the second rule is satisfied. If we take `s(a,b)` to be false and `s(b,a)` to be true, then the second rule is again satisfied. The upshot is that there is ambiguity about `s`. By concentrating exclusively on logic programs that are stratified with respect to negation, we avoid such ambiguities.

View definitions in basic logic programs are required to be both safe and stratified with respect to negation. This is a departure from view definitions in Logic Programming languages like Prolog, which permit rules that are unsafe and logic programs that are not stratified.

The semantics of view definitions in basic logic programs can be formalized by defining the result of applying a basic logic program to a dataset. The resulting *extension* is the set of all facts that can be "deduced" from the dataset on the basis of the rules in the basic logic program.

An *instance* of an expression (atom, literal, or rule) is one in which all variables have been consistently replaced by terms from the Herbrand universe. For example, if we have a language with object constants `a` and `b`, then `r(a) :- p(a,a)`, `r(a) :- p(a,b)`, `r(b) :- p(b,a)`, and `r(b) :- p(b,b)` are all instances of `r(X) :- p(X,Y)`.

Given this notion, we can define the result of single application of a single rule to a dataset. Given a rule r and a dataset Δ, we define $v(r,\Delta)$ to be the set of all ψ such that (1) ψ is the head of an arbitrary instance of r, (2) every positive subgoal in the instance is a member of Δ, and (3) no negative subgoal in the instance is a member of Δ.

Using this notion, we define the result of repeatedly applying the rules in a single stratum Σ to a dataset Δ of facts in the vocabulary of the stratum below. Consider a sequence of datasets defined recursively as follows. $\Gamma_0 = \Delta$, and $\Gamma_{n+1} = \cup v(r,\Gamma_n)$ for all r in Σ. Finally, we define the closure of Σ on Δ to be the union of the datasets in this sequence, i.e. $C(\Sigma,\Delta) = \cup\Gamma_i$.

Finally, we define the *extension* of a basic logic program Ω on dataset Δ as follows. Our definition relies on a decomposition of Ω into strata $\Sigma_1, \ldots, \Sigma_n$. Let $\Delta_0 = \Delta$, and let $\Delta_{n+1} = \Delta_n \cup C(\Sigma_{n+1}, \Delta_n)$. Since there are only finitely many rules in a basic logic program and every stratum must contain at least one rule, there are only finitely many sets to consider (though the sets themselves might be infinite).

It can be shown that there is only one extension for any basic logic program applied to any dataset. Although it is sometimes possible to stratify the rules in more than one way, this does not cause any problems. So long as a program is stratified with respect to negation, the definition just given produces the same extension no matter which stratification one uses.

Note that the extension of any function-free basic logic program on a finite dataset must be finite. Also, the extension of any non-recursive basic logic program applied to a finite dataset must be finite. In both cases, the extension can be computed in time that is polynomial in the size of the dataset.

In the case of recursive programs without function constants, the result must be finite. However, the cost of computing the extension may be exponential in the size of the data, but the result can be computed in finite time.

For recursive programs with function constants, it is possible that the extension is infinite. In such cases, the extension is still well-defined; but in practice it may be necessary to use a different algorithm to compute whether or not a given atom is in the extension. There are multiple choices here. See Ullman's book on Database Systems and Knowledge Base Systems for a discussion of some usable approaches.

4. Transition Rules

In our discussion thus far, we have been talking about the use of datasets and basic logic programs to describe individual states of the world. In many application areas, it is necessary to describe not just individual states but also transitions between states. Transition rules and dynamic logic programs provide the means for us to describe such changes.

The language of Dynamic Logic Programming is a superset of the language of Basic Logic Programing. Every basic logic program is also a dynamic logic program. As in Basic Logic Programming, we can write ground atoms and view definitions. However, in Dynamic Logic Programming, we can also write "transition rules", which encode information about how the state of the world changes (over time or in response to external stimuli).

A transition rule is an expression of the form shown below. Each rule consists of (1) a literal (an atom or a negation of an atom) or a conjunction of literals, (2) a double shafted forward arrow, and (3) a literal or a conjunction of literals. The literals on the left are called *conditions*, and the literals on the right are called *effects*.

$$[\sim]\phi_1 \;\&\; \ldots \;\&\; [\sim]\phi_m \;==>\; [\sim]\psi_1 \;\&\; \ldots \;\&\; [\sim]\psi_n$$

Intuitively, the meaning of a transition rule is simple. If the conditions of a rule are true in any state, then the effects must be true in the *next* state. (Remember that, for a literal to be true, the atom inside the negation must be false.)

For example, the following rule expresses the fact that, when p(a) is true and q(a) is false, then p(a) becomes false and q(a) becomes true in the next state.

```
p(a) & ~q(a) ==> ~p(a) & q(a)
```

As with view definitions, the conditions and effects of rules may contain variables to express dynamic information in a compact form. For example, we can write the following rule to express the fact that the preceding transition rule holds for all objects.

```
p(X) & ~q(X) ==> ~p(X) & q(X)
```

As with view definitions, transition rules are required to be *safe*. In this case, this means that every variable among the effects of a rule or in negative conditions must appear in the positive conditions of that rule.

The transition rule rules shown above are all safe. However, the rules shown below are not. The effect of the first rule contains a variable that does not appear in any condition. In the second rule, there is a variable that appears in a negative condition that does not appear in any positive condition.

```
p(X) & ~q(X) ==> ~p(X) & q(Y)
p(X) & ~q(Y) ==> ~p(X) & q(X)
```

If we were to allow the first of these rules, the resulting dataset might contain infinitely many effects (all the instances of the rule's effect). If we were to allow the second rule, we might have to check infinitely many conditions (all of the instances of the negated condition).

A dynamic logic program (DLP) is simply a collection of view definitions and transition rules. As with basic logic programs, we are interested primarily in dynamic logic programs that are finite. However, in analyzing dynamic logic programs, we occasionally talk about infinite DLPs, e.g. the set of all ground instances of the rules.

We can define the result of executing a dynamic logic program on a given dataset to be a sequence of datasets in which the first dataset is the given dataset and every other item in the sequence is obtained by applying the transition rules of the program to the preceding dataset.

More formally, given a transition rule t and a dataset Δ, we say that an instance of t is *active* if all of the positive conditions in the rule are members of $E(\Omega,\Delta)$ and none of the negative conditions are members of $E(\Omega,\Delta)$.

The *positive update* $A(t,\Delta)$ of applying a transition rule t to Δ is the set of all positive effects in any active instance of t, and the *negative update* $D(t,\Delta)$ is the set of all negative effects in any active instance of t.

The *positive update* $A(\Omega,\Delta)$ of applying a dynamic logic program Ω to Δ is the set of all positive effects in any active instance of any transition rule in Ω, and the *negative update* $D(\Omega,\Delta)$ is the set of all negative effects in any active instance of any transition rule in Ω.

Finally, we define the *execution* of Ω on Δ to be the dataset sequence $\Delta_1, \Delta_2, \Delta_3, ...$, where $\Delta_1=\Delta$ and $\Delta_{n+1} = \Delta_n - D(\Omega,\Delta_n) \cup A(\Omega,\Delta_n)$.

Note that there is a significant difference between dynamic logic programs and production systems. In most production systems, one rule is executed at a time. (Many rules may apply, but typically only one is "fired".) In dynamic logic programs, all transition rules are executed simultaneously, and all updates (both deletions and additions) are applied to the dataset before the rules fire again. This simplifies the specification of dynamics in many cases, and avoids many problems endemic to sequential update systems, such as unintended race conditions and deadlocks.

5. Conclusion

In practice, it is common to extend the simple version of Dynamic Logic Programming described here to include "built-in" relations (e.g. arithmetic) and other operators (e.g. aggregates). The syntax and semantics of such extensions are a little messy. Luckily, they pose no significant theoretical challenges; and, in the interest of brevity, they are not covered here.

The intent of this article is to provide a concise but reasonably rigorous account of the syntax and semantics of Dynamic Logic Programming. For motivation and examples of all of these concepts, see the textbook Dynamic Logic Programming.