Powering Decision Machines With Dynamo

Jan van Eijck

Abstract

This essay is an elaborated version of my last email to Johan van Benthem about using dynamic logic programming, more specifically, programs written in *Dynamo*, for building decision tools and reasoning about these tools. First we sketch the main features of dynamic logic programming, next we look at the use of this new programming paradigm in the analysis of decidable logics, taking propositional logic and the modal logic K as our illustration material.

Contents

1 Dynamic Logic Programming

Dynamic logic programming is the result of making dynamic versions of first order predicate logic executable. The main sources of inspiration for this are the dynamic variable binding strategies that have become fashionable in natural language analysis (DRT [8], Anaphora Logic [2], DPL [7]), the idea of implementing identity assertions as assignment commands familiar from constraint programming, and more in particular from Alma-0 [1], and the general injunction to explore logical dynamics emanating from the works of Johan van Benthem,, e.g. from [3].

The standard dynamic interpretation of FOL, where $\exists x$ is interpreted as the action of random assignment of a new value to register x, is computationally unfeasible, because the $\exists x$ action may lead to infinite branching. In the executable interpretation this is avoided by splitting the quantifier action $\exists x$ in two parts: (i) clearing the variable space for x, and (ii) identifying the new value. The second part is postponed until an identity statement $x = t$ or $t = x$ is encountered.

To get away with this, a careful treatment of the operation of negation is necessary. In dynamic FOL, a negation is simply treated as a test. If there are b with $a[\![\phi]\!]a$ then $\neg\phi$ fails for input state a, otherwise the negation test succeeds, yielding $a[\![\neg\phi]\!]a$. Since in a computable version of this states need not be total, we should be careful with drawing conclusions from $a \stackrel{\phi}{\longrightarrow} b$ (where the arrow indicates a computed transition). We can only draw the conclusion from this that $\neg\phi$ should fail on input a if we can be sure that ϕ succeeds for all extensions of input state a. This is guaranteed if $a \stackrel{\phi}{\longrightarrow} b$ has the so-called forward property: for all a' with $a \subseteq a'$ there is a $b' \supseteq b$ with $a' \stackrel{\phi}{\longrightarrow} b'$. It turns out that the forward property for ϕ is satisfied if we forbid assignments to global variables inside ϕ (see [6] for the definition of computable negation and the proof that this is faithful to dynamic FOL; the forward property is defined in [9]).

Dynamo (see [5]) is a programming language based on this executable interpretation of first order logic. The executable version of dynamic FOL is faithful to the 'standard' dynamic interpretation of first order logic, in the following sense [6]:

- If *Dynamo* computes output b for formula ϕ on input a, then any extension of a to a full valuation (defined on all variable of the language) will be related by the FOL dynamic interpretation of ϕ to a full extension of b.
- If *Dynamo* yields failure for formula ϕ on input a, then ϕ will fail under the FOL dynamic interpretation for any full extension of a.

Of course, the executable interpretation of dynamic FOL can never be complete, for dynamic FOL has the same expressive power as standard FOL. *Dynamo* admits defeat if an attempt is made to test a predicate in which an uninitialized variable occurs.

2 Modelling Variable Assignment in Dynamo

In dynamic logic programming, destructive assignment is replaced by safe assignment. Write $\exists j; j = i; \exists i; i = t$ as:

$$i \gg j = t.$$

The destructive assignment statement $x := x + 1$ can be replaced by the safe assignment statement $x \gg x' = x' + 1$. Here is the general procedure for removing the sting from assignment:

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

3 Translating Dynamo into Standard FOL

The *Dynamo* s statement for incrementing register x, using x_0 as an auxiliary register, `x >> x0 = x0 + 1`, has the following 'forward translation' into standard FOL:

$$\exists x_0(x = x_0 \wedge \exists x(x = x_0 + 1 \wedge \top))$$

Its 'backward translation' is:

$$x = x_0 + 1 \wedge \exists x(x = x_0 \wedge \exists x_0 \top).$$

The general definition of the forward translation of dynamic FOL into standard FOL is given by:

$$\begin{array}{lcl} \texttt{skip}^{\triangleright} & \rightsquigarrow & \top \\ (Pt_1 \cdots t_n; \phi)^{\triangleright} & \rightsquigarrow & Pt_1 \cdots t_n \wedge \phi^{\triangleright} \\ (t_1 \doteq t_2; \phi)^{\triangleright} & \rightsquigarrow & t_1 \doteq t_2 \wedge \phi^{\triangleright} \\ (\texttt{some } v; \phi)^{\triangleright} & \rightsquigarrow & \exists v \phi^{\triangleright} \\ (\texttt{donot } \phi; \psi)^{\triangleright} & \rightsquigarrow & \neg \phi^{\triangleright} \wedge \psi^{\triangleright} \\ (\texttt{either } \phi_1 \texttt{ orelse } \phi_2; \psi)^{\triangleright} & \rightsquigarrow & (\phi_1; \psi)^{\triangleright} \vee (\phi_1; \psi)^{\triangleright} \end{array}$$

The translation satisfies: $\mathcal{M} \models_a \phi^{\triangleright}$ iff there is a b with $a[\![\phi]\!]^{\mathcal{M}} b$, where $[\![\cdot]\!]$ denotes the input output relation for dynamic FOL. *Dynamo* is faithful to $[\![\cdot]\!]$, so if *Dynamo* execution of ϕ succeeds for input a, then $\mathcal{M} \models_a \phi^{\triangleright}$, if it fails for input a, then $\mathcal{M} \not\models_a \phi^{\triangleright}$.

The backward translation of dynamic FOL is given by:

$$\begin{array}{lcl} \texttt{skip}^{\triangleleft} & \rightsquigarrow & \top \\ (\phi; Pt_1 \cdots t_n)^{\triangleleft} & \rightsquigarrow & Pt_1 \cdots t_n \wedge \phi^{\triangleleft} \\ (\phi; t_1 \doteq t_2)^{\triangleleft} & \rightsquigarrow & t_1 \doteq t_2 \wedge \phi^{\triangleleft} \\ (\phi; \texttt{some } v)^{\triangleleft} & \rightsquigarrow & \exists v \phi^{\triangleleft} \\ (\psi; \texttt{donot } \phi)^{\triangleleft} & \rightsquigarrow & \neg \phi^{\triangleright} \wedge \psi^{\triangleleft} \\ (\psi; \texttt{either } \phi_1 \texttt{ orelse } \phi_2)^{\triangleleft} & \rightsquigarrow & (\psi; \phi_1)^{\triangleleft} \vee (\psi; \phi_2)^{\triangleleft} \end{array}$$

This satisfies: $\mathcal{M} \models_b \phi^{\triangleleft}$ iff there is an a with $a[\![\phi]\!]^{\mathcal{M}} b$. *Dynamo* is faithful to $[\![\cdot]\!]$, so if *Dynamo* execution of ϕ yields output b for some input, then $\mathcal{M} \models_b \phi^{\triangleleft}$.

4 Recursion in Dynamo

The current implemention of *Dynamo* assumes evaluation with respect to $\mathbb{Z}$, with relations $<$ and $=$, and with operations $+, -, *$. Alternatively, one might wish to start with just a successor operation $+1$, and then build the primitive recursive functions as follows (we assume x is always the output variable):

- projection: $x = x_i$.
- zero: $x = 0$.
- successor: $x \gg x' = x' + 1$
- composition: Let $g_1(x)$, ..., $g_k(x)$ be programs that use input variables $x_1, \ldots, x_n$. Let $f(x)$ be a program that uses input variables $y_1, \ldots, y_k$. Then composition $f(g_1, \ldots, g_n)$ is given by:

$$g_1(x); x = y_1; \texttt{some } x; \ldots; g_k(x); x = y_k; \texttt{some } x;\ f(x)$$

- primitive recursion, with program f for the basis case, program g for the recursive step (assume that the programs f and g both use x for output, that g use y for input, and that n is the recursion variable):

$$f(x);\ \texttt{do } n \texttt{ times begin } x = y; \texttt{some } x;\ g(x) \texttt{ end}$$

This shows that *Dynamo* computes all primitive recursive functions.

Primitive recursion is modelled using the *Dynamo* construction for bounded iteration `do n times ...`. This is an extension of dynamic FOL. Here is the extension of the forward and backward translation instructions to bounded iteration:

$$\begin{array}{lcl} (\texttt{do } 0 \texttt{ times } \phi)^{\triangleright} & \rightsquigarrow & \top \\ (\texttt{do } n+1 \texttt{ times } \phi)^{\triangleright} & \rightsquigarrow & (\phi;\ \texttt{do } n \texttt{ times } \phi)^{\triangleright} \end{array}$$

$$\begin{array}{lcl} (\texttt{do } 0 \texttt{ times } \phi)^{\triangleleft} & \rightsquigarrow & \top \\ (\texttt{do } n+1 \texttt{ times } \phi)^{\triangleleft} & \rightsquigarrow & (\texttt{do } n \texttt{ times } \phi; \phi)^{\triangleleft} \end{array}$$

Note, however, that these instructions yield translations of ϕ^n that are not *uniform* in n, i.e., n does not occur in the translations as a parameter. To see that this is no accident, recall that the theory of $\mathbb{N}$ with just successor is decidable, but the theory of $\mathbb{N}$ with $+$ and $*$ is not. Thus, $+$ and $*$ are not FOL definable in terms of successor.

Certainly, $+$ and $*$ are primitive recursive, so these operations are definable in *Dynamo*. If there were a *uniform* FOL translation for ϕ^n, then this translation would yield a FOL definition of $+$ and $*$ in terms of just successor, and we know that such a translation cannot exist. On the other hand, if $f(x)$ is a primitive recursive function with input parameter x, then for every specific value $m \in \mathbb{N}$ that one substitutes for x there is a FOL formula $\phi(y)$ that computes $f(m)$ in y (in the obvious sense that a variable state s satisfies $\phi(y)$ in $\mathbb{N}$ iff $s(y) = f(m)$). The snag is that if $m \neq m'$ the translations that compute $f(m)$ and $f(m')$ may look very different.

It is time to draw a moral. If one adds an explicit mechanism for definition by primitive recursion to *Dynamo*, by introducing recursive datatypes, and

recursive operations on these datatypes, one does not extend the expressive power of the formalism. E.g., the natural numbers might be represented by the following datatype.

$$\text{nat} \quad ::= \quad 0 \mid \text{succ nat}$$

Now addition on the natural numbers can be defined by:

```
plus x 0 := 0
plus x (succ y) := succ (plus x y)
```

Compare this with the plain *Dynamo* definition (assume x and y are input variables, and z is output variable):

```
begin
  some z; z = x; do y times z >> z0 = z0 + 1
end
```

Still, explicit primitive recursion is a very useful feature. It makes *Dynamo* programs much more readable. Moreover, the recursively defined functions and predicates can be viewed as extensions of the signature, so that we can specify the meanings of *Dynamo* programs in terms of FOL translations in which these new functions and predicates occur.

5 Decision Machines for Propositional Logic

For present purposes, it is useful to have a datatype for formulas:

$$\text{form} \quad ::= \quad (\text{p nat}) \mid \text{form} \ \wedge \ \text{form} \mid \text{form} \ \vee \text{form} \mid \neg\text{form} \mid \Box\text{form} \mid \Diamond\text{form}$$

We may assume that we have a read operation for this datatype. Thus, `read (form F)` would read a formula into variable F. Primitive recursion over `form` can now be used to give explicit translation instructions for formulas.

A *Dynamo* program for checking a formula of propositional logic against a propositional valuation could look like this (we assume that the definition of the datatype `nat` is built-in):

```
program propvalcheck;

form ::= (p nat) | form and form | form or form | not form

verify (p n) := p[n] = 1
falsify (p n) := p[n] = 0

verify (F and G) := begin verify F; verify G end
falsify (F and G) := donot begin verify F; verify G end

verify (F or G) := donot begin falsify F; falsify G end
falsify (F or G) := begin falsify F; falsify G end

verify (not F) := falsify F
falsify (not F) := verify F

begin
  read (form F); verify F
end.
```

The clauses of `verify` and `falsify` are an example of procedure definition by primitive recursion. The procedure `verify` is deterministic, so we have a program that checks a formula against a propositional valuation in polynomial time.

A thing to be noted is that the use of the *Dynamo* `donot` construction is essential to get the definitions of `verify` and `falsify` deterministic. The requirement on `donot` in *Dynamo* is that within its scope no global assignment should take place. Global assignment is the use of identities of the form $v = t$ or $t = v$ where v is a global variable (not introduced by a dynamic quantifier) in a state a with $a(t)$ defined and $a(v)$ undefined. If we use the procedure for checking against a valuation we can assume that truth values of the proposition letters p_i occurring in a formula are represented in the input state as values 0 or 1 of array elements `p[i]`. Therefore, the statements `p[n] = 1` and `p[n] = 0` are always tests and never assignment statements.

If one wants a procedure for satisfiability checking, matters are different. Then the procedure for verifying a proposition letter p_n is still given by `p[n] = 1`, but now we cannot be sure anymore that this is a test. Therefore, the use of *Dynamo* negation is out, and we have to use the *Dynamo* construction for indeterministic choice instead.

```
program propsatcheck;

form ::= (p nat) | form and form | form or form | not form

verify (p n) := p[n] = 1
falsify (p n) := p[n] = 0

verify (F and G) := begin verify F; verify G end
falsify (F and G) := either falsify F orelse falsify G

verify (F or G) := either verify F orelse verify G
falsify (F or G) := begin falsify F; falsify G end

verify (not F) := falsify F
falsify (not F) := verify F

begin
  read (form F); verify F
end.
```

Program `propsatcheck` is nothing but a *Dynamo* implementation of the Beth tableau procedure for satisfiability checking. Although this program looks very similar to the previous one, it behaves quite differently. It builds a satisfying propositional valuation, and does so indeterministically, by trying out all the possible avenues.

It is well known that the tableau method for propositional satisfiability checking is not always an improvement on the truth table method: as [4] observes, the complexity of a tableau proof depends essentially on the length (number of symbols) of the input formula, while the complexity of a truth table for that formula depends only on the number of distinct propositional variables that occur in it. Thus, for certain 'fat' formulas (formulas where the number of symbols is large when compared to the number of distinct proposition variables), the truth table method may actually be more efficient than the tableau method. The reason for this is, essentially, that the tableau method may list possibilities that are not mutually exclusive as 'different', or in other words, that the tableau method may explore redundant paths, because the branches do not represent mutually exclusive partial valuations. The remedy is easy: branchings should be made mutually exclusive. The following *Dynamo* program makes clear how this can be done:

```
program propsatcheckALT;

form ::= (p nat) | form and form | form or form | not form

verify (p n) := p[n] = 1
falsify (p n) := p[n] = 0

verify (F and G) := begin verify F; verify G end
falsify (F and G) := either falsify F orelse
                        begin verify F; falsify G end

verify (F or G) := either verify F orelse
                        begin falsify F; verify G end
falsify (F or G) := begin falsify F; falsify G end

verify (not F) := falsify F
falsify (not F) := verify F

begin
  read (form F); verify F
end.
```

6 Decision Machines for Modal Logic

To implement checking of a modal formula in a Kripke tree model, we assume the model is represented as an array with the tree nodes as indices, and that $d[n]$ gives the number of daughters of node n. The program assumes that register n holds the current point of evaluation in the Kripke model. Evaluation of proposition letter p_k at node n uses array element `p[n][k]`.

As an example, here is the bit of code for verifying and falsifying a diamond formula (using `n^i` for daughter i of node n, and `m(n)` for the mother of node n):

```
verify (diamond F)  :=
  begin
    some i; i = 0;
    donot do d[n] times
      begin
        i >> i' = i'+ 1; n >> n' = n'^i;
        falsify F; n >> n' = m(n')
      end
  end

falsify (diamond F) :=
  begin
    some i; i = 0;
    do d[n] times
      begin
        i >> i' = i'+ 1; n >> n' = n'^i;
        falsify F; n >> n' = m(n')
      end
  end
```

Again, the use of the *Dynamo* `donot` construction is licensed by the fact that inside its scope no global assignment takes place.

Dynamo can also be used for *constructing* Kripke trees, i.e., for the generation of finite Kripke models. For this application, the use of `donot` is to be avoided, for when *Dynamo* is used in model generation mode the nodes of the tree are being built by means of assignment of values to array registers. What the verifying procedure for $\Diamond F$ has to do now is: create a new daughter, move to that daughter, verify F at that daughter node (i.e., extend the tree at that node by the information supplied by F if that is possible), and return to the mother.

7 Conclusion

What I hope to have demonstrated is that Dynamic Logic Programming is not only an exciting new development in its own right, but that it is also useful. It can be used to build perspicuous decision algorithms for decidable logics: the form of the *Dynamo* program gives insight in the complexity of the decision problem for the logic. An obvious next step will be the construction of a *Dynamo* program that decides the Guarded Fragment of FOL. I am confident that it can be done.

References

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

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

[3] J. van Benthem. *Exploring Logical Dynamics*. CSLI & Folli, 1996.

[4] Marcello d'Agostino. Are tableaux an improvement on truth-tables? *Journal of Logic, Language, and Information*, 1:235–252, 1992.

[5] Jan van Eijck. Dynamo—a language for dynamic logic programming. Manuscript, CWI/ILLC, December 1998. Available from `www.cwi.nl/~jve/dynamo`.

[6] Jan van Eijck. Programming with dynamic predicate logic. Technical Report CT-1998-06, ILLC, 1998. Available from `www.cwi.nl/~jve/dynamo`.

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

[8] H. Kamp. A theory of truth and semantic representation. In J. Groenendijk et al., editors, *Formal Methods in the Study of Language*. Mathematisch Centrum, Amsterdam, 1981.

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