LOGICAL ACTION SYSTEMS

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal

ABSTRACT.

Logic programming is being hailed by many people as a good way towards a side-effect-free programming style. On the other hand, talking about temporal effects or **actions** is the natural way of viewing many common computational phenomena, such as input/output or database update operations.

The purpose of this paper is to introduce some common ground in the form of **logical action systems**, a framework for dealing with actions that has its roots in logic programming. Programs consist of rules for action reduction; rules have preconditions as Prolog-like goal expressions and define state transitions in the form of deletion and/or creation of assertions. **Concurrency** of actions is supported. **Abstract data types** can be defined.

INTRODUCTION.

Despite the defense by many people of a side-effect-free programming style, as in a 'pure' logic programming system, the fact remains that many common computational phenomena are not naturally expressed without resorting to the notions of action and state transition.

Rather than considering actions as impure side-effects arising within a pure logic computation, why not invert the situation and consider logic computations as a normal side-effect of action systems?

We will put forward a proposal for a language in which to describe **logical action systems (LAS)**, providing a clear link between actions and normal logic programming.

Logic and **unification** are still the basis on top of which LAS are conceived; however, actions are clearly separated from purely deductive goals.

The language can be seen as yet another proposal for expressing **concurrency**.

We will begin by exposing the main ideas behind LAS, and then move on to an **object-oriented** approach with **abstract data type** definitions.

ACTIONS.

Actions take place on some world, modifying it. Between occurrences of actions we can refer to the **state** of the world.

We represent a world in two parts, each one of them a logic program:

(1) the **rules of the world**, defining relations that are not bound to change in time ;

(2) the **state of the world**, containing assertions that may change in time as the result of actions performed on the world.

Let us look at an example. (Edinburgh Prolog syntax will be used, except for clause functor.)

Consider a blocks world.

The world rules would contain definitions such as :

```
tower([B]) <- on(B,floor).

tower(B1.B2.Bn) <- on(B1,B2), tower(B2.Bn).
```

A particular world state would have assertions such as :

```
on(a,b).

on(b,floor).
```

At any time between actions it is possible to evaluate a goal expression against the world, using the rules and the state as a joint logic program.

In this example, one could evaluate the goal

```
        <- tower(X).
```

that would yield the solutions

```
        X=[b] ; X=[a,b] .
```

The action of moving 'a' to the 'floor' would replace the assertion 'on(a,b)' by 'on(a,floor)'. As a consequence we would have a new world state, and the same goal '<-tower(X)' would now produce the solutions

```
        X=[a] ; X=[b] .
```

In general, an action will consist of a number of action steps (possibly infinite).

Each action step will result in a change of state, consisting of falsifying (deleting) some assertions of the previous state and/or making true (creating) some new assertions.

The specification of an action step is an **action rule**. It is made up of two parts : the **action reduction** and the **state conditions**.

The **action reduction** defines what new actions the action reduces to, by virtue of the step.

The **state conditions** are the preconditions and the state transitions.

Preconditions are goals that are evaluated against the world in its current state.

State transitions tell what assertions must be deleted from, and what new ones added to, the current state to get the new state.

For example, the notion that the action of moving A to B can be accomplished if nothing is on top of A and B, and as a result A ceases to be on top of whatever it was before to be on top of B, can be described by the action rule :

```
        move(A,B)  <=  not on(_,A),
                       not on(_,B),
                       on(A,_) -> on(A,B).
```

ACTION REDUCTION.

When an action reduces to void (is finished), as in the preceding example, the action reduction part of the rule is just the action - the rule head.

In general, an action reduces to other actions. The action reduction part of a rule is then of the form

```
A -> NA
```

where A is some action (the rule head), and NA is an action expression refering to the new actions.

How can actions relate to one another to form an action expression? We find that we need two connectives: **parallel** and **sequence**.

Two actions in **sequence** are denoted by 'A,B' , meaning the second action (B) can only take place after the first one (A) is finished.

Two **parallel** actions, written 'A/B' , may take place with no time constraints on one another.

An action expression is recursively constructed from atomic actions and the parallel and sequence connectives. Relative precedence between these is such that 'A/B,C' is the same as '(A/B),C' .

In an action system there is always an action expression evolving in time and denoting at each moment the actions that are to be carried out in the world. We call it the **agenda**. For every action in the agenda that is **ready** to be carried out (for example, A1 and B1 in (A1,A2)/(B1,B2)), the system tries to apply an action step.

The action reduction involved in a step is like a **rewrite rule** for the ready action in the agenda, keeping the overall structure of this action expression.

Thus, if we have the agenda

```
A,B
```

and the action reduction

```
A -> A1/A2
```

is performed, the agenda becomes

```
A1/A2,B
```

meaning that after A1 and A2 are both finished (having done so independently of one another) B is ready to take place.

Actions occur in time and time always runs forward, so there is no question of backtracking over action steps. If an action is required and no rule for that action applies in the current state, it just means that the action must **remain** in the agenda **waiting** for the right conditions to appear (when some other action changes the state to that effect). This eventually entails the well-known phenomena of deadlock and starvation.

STATE TRANSITIONS.

State transitions inside an action rule may be of three types :

(1)	-> A	assertion A is created ;
(2)	A ->	assertion A is deleted ;
(3)	A -> NA	assertion A is deleted and assertion NA is created.

Of course a type 3 transition is no more than a type 1 and a type 2 put together, but it makes for a more clear reading of the rule, especially if A and NA are for the **same predicate**. In this case, a compiler or interpreter can easily translate the transition into simple **assignments** on the changing arguments, with considerable speed-up over deletion and creation.

RULE EVALUATION.

Each rule is associated with a **single** action (the rule head), so a ready action in the agenda can **efficiently** trigger its own rules, much as Prolog goals trigger their clauses.

Rule evaluation begins with **unification** of the ready action with the rule head.

If there are any type 2 or type 3 transitions in the rule, their left-hand side is regarded as a **goal** to be matched against an **assertion** in the current world state. All **precondition goals** together with these **transition goals**, in the order in which they appear in the rule, form a Prolog goal expression that is evaluated. If a solution is found, then the rule applies, and the transitions are carried out, deleting the assertions that matched the transition goals for the solution found.

The action is replaced in the agenda by the new action expression it reduced to, with the obvious simplifications when this is void.

There may be several rules for a given action. Rules should be tried in the **order** in which they appear in the program. This provides a simple, elegant form of **if-then-else**.

For example, the complete definition for the 'move' action in the blocks world might be :

```
        move(A,floor) <=  not on(_,A),
                          on(A,_) -> on(A,floor).

        move(A,B) <=  not on(_,A),
                      not on(_,B),
                      on(A,_) -> on(A,B).
```

giving preference to 'move's to the 'floor', if destination is unspecified.

SYNCHRONIZATION.

What is usually referred to as **process synchronization** is achieved in a LAS by the combined effect of the sequence connective and **state transitions** seen by the "processes".

Imagine a single cell buffer, defined by the following actions :

```
        put(X)  <=  empty -> with(X).

        get(X)  <=  with(X) -> empty.
```

A 'put' action will only be accomplished if the buffer is empty, and, conversely, a 'get' action can only be carried out if the buffer contains something. So actions sequenced after a 'put' will eventually have to wait for the 'get' of a previous token put in the buffer, and actions sequenced after a 'get' will eventually have to wait for the 'put' of the corresponding token, thus achieving synchronization of the two "processes" using the buffer.

CONCURRENCY.

Parallel actions are performed **concurrently**. So it is crucial that any sound implementation of the system be able to guarantee, just before performing a state transition, that the preconditions of the rule still apply. In other words, care must be taken with regard to state transitions occurring during the evaluation of a rule. A number of techniques exist for tackling this problem, depending on the actual hardware, but their discussion is outside the scope of this paper.

Let us look at an implementation of a queue in terms of its accessing actions 'put' and 'get'. The queue itself is implemented as a difference-list Q-T via an assertion 'q(Q,T)', acted upon by 'put' and 'get' :

```
        put(X)  <=  q(Q,X.T) -> q(Q,T).

        get(X) <=  q(Q,T) -> q(NQ,T),
                   nonvar(Q),
                   Q=(X.NQ).
```

This queue "process" puts in a list all elements X for which a 'put(X)' action is requested, in the order in which these actions are performed (since they can always be executed, apart from simultaneity with 'get' actions, this will be the order in which they become ready in the agenda).
This is in contrast to other formalisms, such as Concurrent Prolog [Shapiro 83], that deal with explicit streams and thus require the explicit merge of the various input streams to a queue.

Let us look at another classic example of concurrent programming, the problem of the dining philosophers. Five philosophers are seated around a table, with a fork between each two of them (five in all) and a central bowl of spagethi. Whenever a philosopher stops thinking because he gets hungry, he must pick up the two forks on his left and right and begin eating until satisfied, letting then down the two forks and resuming his thinking.

Philosophers - p1, p2, p3, p4, p5

Forks - f1, f2, f3, f4, f5

World rules :

```
        forks(p1,f1,f2).
        forks(p2,f2,f3).
        forks(p3,f3,f4).
        forks(p4,f4,f5).
        forks(p5,f5,f1).
```

Initial world state:

```
        down(f1).
        down(f2).
        down(f3).
        down(f4).
        down(f5).
```

Initial agenda :

```
        -> thinking(p1) / thinking(p2) / thinking(p3) /
           thinking(p4) / thinking(p5).
```

Action rules :

```
        thinking(X) -> hungry(X).

        hungry(X) -> eating(X)  <=
                     forks(X,L,R),
                     down(L) -> with(X,L),
                     down(R) -> with(X,R).

        hungry(X) -> wants_fork(X,R)  <=
                     forks(X,L,R),
                     down(L) -> with(X,L).

        hungry(X) -> wants_fork(X,L)  <=
                     forks(X,L,R),
                     down(R) -> with(X,R).

        wants_fork(X,F) -> eating(X)  <=
                           down(F) -> with(X,F).

        eating(X) -> thinking(X)  <=
                     with(X,L) -> down(L),
                     with(X,R) -> down(R).
```

Some comments are due.

The first and last rule, of course, do not show any details about when to get hungry or when to stop eating. For an actual simulation we should provide adequate mechanisms, say a random time lapse generator.

It is important to note that, in the last rule, the two 'with' transition goals must match two **distinct** assertions and not the same one. Type 2 or type 3 transitions inside the same rule always refer to distinct assertions, for it would make no sense to specify two deletions of a single assertion.

The aforementioned if-then-else effect of rule evaluation implies that, when a philosopher gets hungry and both his two forks are available, he will pick them up **simultaneously**. This fact entails that there is no deadlock or starvation if the system starts from a non-deadlock initial state, as can be easily proved. What happens is a transfer of deadlock/starvation monitoring to the underlying execution mechanism of LAS, when concurrently trying to apply action rules. We are in fact assuming that no ready action is indefinitely postponed if conditions indefinitely exist for its reduction.

ABSTRACT DATA TYPES.

One of the nice extensions of the action system presented so far is the introduction of **abstract data type** (ADT) definitions. This provides a much needed **modularity**, in the form of **local** assertions that cannot be globally accessed, and are manipulated only by the actions interfacing the ADT object with the rest of the system.

A definition of a queue ADT might be the following :

```
        type queue.

        put(X)   <=   q(Q,X.T) -> q(Q,T).

        get(X)   <=   q(Q,T) -> q(NQ,T),
                      nonvar(Q),
                      Q=(X.NQ).

        #

        q(X,X).

        *
```

The first part of an ADT definition, until the character '#', defines the **external** actions that may be used to access an object of the given type.
In this example we have the previously defined 'put' and 'get' actions.

The second part of the definition, until the character '*', defines **internal** rules and assertions, that cannot be accessed from the outside.
The assertions correspond to the **initial state** of an object, when it is created. There can also exist, in the second part of an ADT definition, an **initial agenda** '->A' to be launched upon creation of an object.
In the preceding example the initial state is an empty queue, as defined by the assertion 'q(X,X)', and there are no internal actions or initial agenda. The assertion being local, it won't be "seen" by any outside goal 'q(_,_)'.

Having defined an ADT, we must have means to create and kill objects of that type. We use the system-defined actions

create(Object,Type)

and

kill(Object) .

Now actions directed at an object must refer to it. We use the notation

```
                    Object:Action
```

for that kind of actions.

Let us look at a more complex example, a definition of a video terminal. The keyboard is scanned to get characters typed in it. In the local mode each character is output on the screen; however, if the character is a 'send', character output is diverted to the outside of the terminal, until the character 'eot' is found, in which case there is a switching back to local mode. The terminal can be accessed from the outside through a 'put(X)' action, resulting in character X being displayed in the screen.

This ADT has three parameters, 'Keyboard', 'Screen' and 'Out', which are supposed to be ADT objects themselves. 'Keyboard' is supposed to be accessed through a 'get' action, while 'Screen' and 'Out' through a 'put'.

```
  type terminal(Keyboard,Screen,Out).

  put(X) ->  Screen:put(X).

  #

  terminal(X) -> select(X) / Keyboard:get(NX), terminal(NX).

  select(send)  <=  local -> out.
  select(eot)  <=  out -> local.
  select(X) -> Screen:put(X)  <=  local.
  select(X) -> Out:put(X)  <=  out.

  local.

  -> Keyboard:get(X), terminal(X).

  *
```

External access is permitted only through a 'put' action. 'terminal' and 'select' are **internal** actions - 'terminal' performs the endless loop of getting characters from the keyboard and processing them; 'select' does this processing.

The **initial state** is local mode, and the terminal activity is started by getting a character from the keyboard and entering the loop.

A terminal, being accessed through a 'put', can serve as the 'Out' object of another terminal. We can for example link two terminals together :

```
        -> create(T1,terminal(k1,s1,T2)) /
           create(T2,terminal(k2,s2,T1)).
```

DEDUCTION AS ACTION.

We tackle here the problem of treating as an action the work of a Prolog interpreter while trying to execute a goal. Keep in mind that backtracking is "backward" as far as the object language goes, but is "forward" as regards the temporal activity (action) of the interpreter.

A drawback of Prolog is revealed when we want to keep track of different solutions to a goal while getting them on demand, along with some other computation. The problem lies in the fact that we are using a single interpreter, and backtracking, that is needed locally to provide the various solutions, is only available as a global operation.
'Metapredicates' like 'setof' or 'all' only give the whole set of solutions to a goal, and cannot be used in the desired coroutined way.

A way out in the framework of LAS is to have the Prolog interpreter defined as an ADT, accessible through the action of producing the next solution to a goal.

We can then create an instance of the interpreter by the action

create(I,interpreter(G,T))

where G is the goal expression to be interpreted, and T is the term whose instances we seek.

Transporting the name I of this particular interpreter we can then get on demand the next solution, with the action

I!next(X) .

As a result, X is bound in the action environment to a copy of the next instance of T found by I to be a solution for G (T and G will remain unbound in the action environment).
This is to say that several interpreter objects are truly decoupled in the sense that they don't share their binding environments. Some more thought should be given to this theme of sharing versus copy, in the context of LAS.

There remains the problem of failure. The action 'next' is always carried out, but it should produce information regarding its outcome, that can be used in the action context.

Maybe this type of actions should really be used as a logical goal, with associated meaning the truth or falsehood of the possibility of performing the action.
We can turn this into a general property of actions, with the assumption that the default boolean value of an action is true when the action is normally finished, and false if unfinished when the special action 'abort' is carried out, making 'impossible' the whole action expression (agenda) where it occurs (it becomes empty).

Remember that using ADTs one has distinct agendas for each object, and thus 'abort' can be used in a modular rather than global way. One can, for example, implement a Unix-like shell using the 'abort' action triggered by the control_C trap to abort execution of the current command !

```
type shell(Input,Command_interpreter).

#

commands(C) -> Input:set(NC) / Command_interpreter:C ,
               commands(NC).

control_C_trap(C) -> Input:set_ahead(NC) / check(C) ,
                     control_C_trap(NC).

check(^C) -> Command_interpreter:abort.
check(_) -> .

-> Input:set(C),
   commands(C) / control_C_trap(C).

*
```

REFERENCE.

[Shapiro 83]

Ehud Shapiro.
A Subset of Concurrent Prolog and its interpreter.
The Weizmann Institute of Science.