Combining Prolog and Imperative Computing in LPS

Robert Kowalski\(^1\), Fariba Sadri\(^1\), Miguel Calejo\(^2\) and Jacinto Davila\(^3\)

\(^1\) Imperial College London, \(^2\)logicalcontracts.com, Lisbon, \(^3\)Contratos Lógicos. C.A. and Universidad de Los Andes

Logic programs and imperative programs employ different notions of computation. Logic programs compute by proving that a goal is a logical consequence of the program, or by showing that the goal is true in a model defined by the program. Imperative programs compute by starting from an initial state, executing actions to transition from one state to the next, and terminating in a final state when the goal is solved.

Prolog has a “pure” logic programming (LP) kernel, extended with language features that do not have an obvious logical interpretation. Arguably, the most important of these features is the ability to assert and retract clauses, which allows Prolog to simulate destructive change of state in imperative programming languages.

In this paper, we present the language LPS (Logic Production Systems) [8-15], which combines the LP and imperative programming notions of computation and gives a restricted form of assert and retract in Prolog a logical interpretation. Computation in LPS follows the imperative paradigm of solving goals by generating a sequence of states and events, to make the goals true. States are sets of facts (called fluents) that change with time. Events include both external events and internally generated actions. State transitions are made by destructively asserting and retracting fluents using causal laws written in the LP form:

\[
event \text{ initiates fluent if conditions.} \\
evend terminates fluent if conditions.
\]

Operationally, when an event happens, then the causal laws are “executed” by retracting all fluents whose terminating conditions hold in the current state, and asserting all fluents whose initiating conditions hold. All other fluents persist without change. This operational behaviour can be specified by a Causal Theory, using LP clauses (where the integers \(T\) and \(T+1\) denote consecutive discrete times or states) such as:

\[
\text{holds(Fluent, } T+1) \text{ if happens(Event, } T, T+1), \text{ initiates(Event, Fluent, } T+1).
\\
\text{holds(Fluent, } T+1) \text{ if holds(Fluent, } T), \not \text{ happens(Event, } T, T+1), \text{ terminates(Event, Fluent, } T+1).
\]

It is important to appreciate that LPS does not use this Causal Theory to reason about persistence. The Causal Theory is an emergent property that is true in the time-stamped model that is implicitly generated by the LPS computation.

As a simple example, consider the following LP clauses in LPS syntax:

\[
\text{initially lightOn.} \quad \text{observe switch from 1 to 2.} \quad \text{observe switch from 3 to 4.} \\
\text{lightOff if not lightOn.} \\
\text{switch initiates lightOn if lightOff.} \\
\text{switch terminates lightOn if lightOn.}
\]

In general, causal laws directly initiate and terminate only extensional fluents, such as \(\text{lightOn}\). Intentional fluents, such as \(\text{lightOff}\), are initiated and terminated only as ramifications of changes to the extensional fluents.

The SWISH [20] implementation includes a visualisation of the model generated by the computation, displaying the extensional fluent \(\text{lightOn}\):
Unlike possible worlds in modal logic, models in LPS are single models in which fluents and events indicate the times (or states) for which they hold. In this example, the model can be written as a standard Herbrand interpretation:

\[
\{ \text{happens(switch, 1, 2), happens(switch, 3, 4), holds(lightOff, 2), holds(lightOn, 3), terminates(switch, 1, 2), lightOn, 2), initiates(switch, 3, 4), lightOn, 4), holds(lightOn, 1), holds(lightOn, 4), holds(lightOn, 5), holds(lightOn, 6), \ldots \} \]

In addition to causal laws, which destructively update the current state, LPS includes reactive rules of the form \textit{if antecedent then consequent}. Logically, these are goals that need to be satisfied by generating a model that makes them true. For example:

\[
\text{if lightOff then switch.}
\]

LPS uses the LP component of LPS to determine when the \textit{antecedent} of the rule becomes true, and then generates actions and executes the causal laws to make the \textit{consequent} of the rule true. As in this example, computation can be a non-terminating process, if time is infinite and an external event can occur at any time. In this case, an initial portion of the infinite model generated by the computation can be visualised as:

The reactive rule above is shorthand for the sentence:

\[
\text{For all } T1 \ [\text{if holds(lightOff, } T1) \text{ then there exists } T2 \text{ such that } \{ \text{happens(switch, } T2, T2+1) \text{ and } T1 \leq T2 \}].
\]

Here the switch action can be executed any time after the light is on. However, in practice, LPS generates models in which goals are satisfied as soon as possible. In general, both the \textit{antecedent} and \textit{consequent} of a reactive rule can be a conjunction of (possibly negated) timeless predicates, such as the inequality relation \( \leq \), and (possibly negated) fluents and events. All variables in the \textit{antecedent} are universally quantified with scope the entire rule. All other variables are existentially quantified with scope the \textit{consequent} of the rule. All times in the \textit{consequent} are later than or at the same time as the latest time in the \textit{antecedent}. 

\[\text{Timeline}\]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>switch</td>
<td>switch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lightOn</td>
<td>lightOn</td>
<td>lightOn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actions</td>
<td>lightOn</td>
<td>lightOn</td>
<td>lightOn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{go(Timeline)}\].
The antecedents and consequents of reactive rules can also include complex events defined by LP clauses of the form:

\[ \text{complex-event if conditions.} \]

where the conditions have the same syntax as the antecedents and consequents of reactive rules. The start time of the complex-event is the earliest time in the conditions, and the end time of the complex-event is the latest time in the conditions.

For example, the following two LP clauses define a complex event, sos, which is a simplified distress signal of a light flashing three times in succession. Each flash of light is on for two time steps and is separated from the next flash by one time step. We have not yet defined a shorthand, time-free syntax for such clauses:

\[ \text{sos from } T_1 \text{ to } T_4 \text{ if } \text{lightOff at } T_1, \text{ flash from } T_1 \text{ to } T_2, \text{ flash from } T_2 \text{ to } T_3, \text{ flash from } T_3 \text{ to } T_4. \]

\[ \text{flash if switch to } T, \text{ switch from } T+1. \]

Given this definition and replacing the reactive rule above by the rule:

\[ \text{if sos to } T \text{ then sos from } T+3. \]

LPS uses the complex event definition both to recognise an sos and to generate an sos in response. Moreover, once it has responded to an sos it has recognised, it then recognises its own sos and responds to it by generating another sos, ad infinitum [17].

In addition to the timeline visualisations, the SWISH implementation of LPS includes animations which display only one state at time. See for example dad chasing around the house to turn off the lights that bob turns on [16].

The preceding examples illustrate the main features of LPS (except for constraints of the from false if conditions, which restrict the actions an agent can perform). The examples show that reactive rules are the driving force representing an agent's goals, supported by Prolog-like logic programs, representing the agent’s beliefs.

LPS was inspired by trying to understand the difference and relationship between rules in LP and rules in production systems [18], as well as by complex actions in Transaction Logic [3], reactive rules in MetaTem [2] and agent plans in AgentSpeak [19]. It is also related to CHR [6], DALI [5], Epilog [7] and EVOLP [1, 4].
References

16. LE light: https://demo.logicalcontracts.com/example/badlight.pl last accessed 2022/09/07