LPS (Logic-based Production Systems)

Robert Kowalski and Fariba Sadri
Imperial College London

Miguel Calejo
Interprolog.com
Overview

Introduction

What is LPS?
Logic programming
Production systems
LPS (Logic Production System)

LPS examples

LPS in relation to computer science

LPS implementation

Foundations of LPS and related work
What is LPS?

An open-source, web-based prototype.

LPS is a logic and computer language for
• programming
• databases
• AI (intelligent agent) applications.

LPS includes
• logic programming (as in Prolog)
• production systems (as a model of human thinking).

LPS teaches
• computational thinking
• logical thinking
LPS bridges the gap between Computational and Logical Thinking

- Computational thinking
  - Algorithms using state transitions
  - Abstraction
  - Goals and beliefs
  - Problem decomposition by backward reasoning
  - Forward reasoning to derive logical consequences

- Logical thinking

LPS
Short History of Logic Programming

1969 – 72  Procedural versus logical representations in AI

1971 – 72  Prolog - the first Logic Programming Language

1980s  Datalog
  Logic as a Computer Language for Children
  The British Nationality Act as a Logic Program
  The Fifth Generation Project chooses Logic Programming

1990s – 2000s  AI and Logic Programming in decline

2010s  Resurgence of Logic Programming

beliefs
LPS
goals and beliefs
Fifth generation computer
From Wikipedia, the free encyclopedia

The *Fifth Generation Computer Systems* [Present and Beyond] (FGCS) was an initiative by Japan's Ministry of International Trade and Industry, begun in 1982, to create a computer using massively parallel computing/processing. It was to be the result of a massive government/industry research project in Japan during the 1980s. It aimed to create an "epoch-making computer" with supercomputer-like performance and to provide a platform for future developments in artificial intelligence.

The target defined by the FGCS project was to develop "Knowledge Information Processing systems" (roughly meaning, applied Artificial Intelligence). The chosen tool to implement this goal was logic programming.
Resurgence of Logic Programming

Design and Implementation of the LogicBlox System
Molham Aref Balder ten Cate Todd J. Green Benny Kimelfeld Dan Olteanu Emir Pasalic Todd L. Veldhuizen Geoffrey Washburn LogicBlox, Inc. 2015

A major goal of our system is to unify the programming model for applications that automate and enhance decision making by using a single, expressive, declarative language that can be used by domain experts to understand and evolve the application.

To achieve this goal, we have developed LogiQL, an extended form of Datalog that is expressive enough to allow coding of entire applications (including queries and views; stored procedures; reactive rules and triggers; and statistical and mathematical modeling).

But already today, the LogicBlox platform has matured to the point that it is being used daily in dozens of mission-critical applications in some of the largest enterprises in the world, whose aggregate revenues exceed $300B.
Resurgence of Logic Programming

Yedalog: Exploring Knowledge at Scale
Brian Chin\textsuperscript{1}, Daniel von Dincklage\textsuperscript{1}, Vuk Ercegovac\textsuperscript{1}, Peter Hawkins\textsuperscript{1}, Mark S. Miller\textsuperscript{1}, Franz Och, Christopher Olston\textsuperscript{1}, and Fernando Pereira\textsuperscript{1}
1 Google, Inc. 2015

Abstract We introduce Yedalog, a declarative programming language that allows programmers to mix data-parallel pipelines and computation seamlessly in a single language. Yedalog extends Datalog, incorporating not only computational features from logic programming, but also features for working with data structured as nested records.
PageRank in Yedalog. Assumes the existence of Node and Edge predicates that describe a directed graph.

# The random jump probability.
Alpha = 0.15;

# Number of outlinks for every node.
Outlinks(j) += 1.0 :- Edge(j, _i);

# The (non-normalized) PageRank algorithm.
PageRank(i) += Alpha :- Node(i);
PageRank(i) += PageRank(j) * (1.0 - Alpha) / Outlinks(j) :- Edge(j, i);
Production Systems —Herbert A. Simon

Production systems are computer languages that are widely employed for representing the processes that operate in models of cognitive systems (Newell and Simon 1972).

In a production system, all of the instructions (called productions) take the form:

IF <<conditions>>, THEN <<actions>>, 

That is to say, “if certain conditions are satisfied, then take the specified actions” (abbreviated C → A). Production sys-
Production systems

States described by a working memory of facts. State transitions represented by condition-action rules.

Popular for implementing expert systems as a computational model of human thinking (e.g. SOAR, ACTR, Steven Pinker’s *How the Mind Works*)
Production systems do not have a logical semantics.

\[
\begin{align*}
\text{fire} & \Rightarrow \text{deal-with-fire} \\
\text{deal-with-fire} & \Rightarrow \text{eliminate} \\
\text{deal-with-fire} & \Rightarrow \text{escape}
\end{align*}
\]

Adding \textit{fire} to working memory. 
Triggers two candidate actions \textit{eliminate} and \textit{escape}. 
\text{Conflict resolution} decides between them.
Production systems.
Goals and Beliefs - It can be hard to tell them apart.

"Rules are if-then structures... very similar to the conditionals... but they have very different representational and computational properties."
Goals and beliefs
It can be hard to tell them apart

Unlike logic, rule-based systems can also easily represent strategic information about what to do. Rules often contain actions that represent goals, such as *IF you want to go home for the weekend, and you have bus fare, THEN you can catch a bus*. Such information about goals serves to focus the rule-

Logic Program:

*You go home from T1 to T2*  
*if you have the bus fare at T1,*  
*you catch a bus from T1 to T2,*  
*the bus arrives at T2.*
Goals and Beliefs:
It can be hard to tell them apart.

All humans are mortal.
All humans are kind.

Goals:

\( \text{if } \text{human}(X) \text{ then } \text{mortal}(X). \)
\( \text{if } \text{human}(X) \text{ then } \text{kind}(X). \)

or

Beliefs:

\( \text{mortal}(X) \text{ if } \text{human}(X). \)
\( \text{kind}(X) \text{ if } \text{human}(X). \)
LPS combines reactive rules, logic programs and causal laws

Reactive rule: \( \text{if fire then deal-with-fire.} \)

Logic program: \( \text{deal-with-fire if eliminate.} \)
\( \text{deal-with-fire if escape.} \)

Causal law: \( \text{eliminate terminates fire.} \)

Adding \textit{fire} to the current state.
Generates two alternative actions \textit{eliminate} or \textit{escape}.
Generates alternative world models
to make the reactive rule true:

\begin{itemize}
  \item \textit{eliminate} \quad \textit{fire} \quad \textit{fire}
  \item or \quad \textit{fire} \quad \textit{fire} \quad \textit{fire}
  \item \textit{escape} \quad \textit{fire} \quad \textit{fire} \quad \textit{fire}
\end{itemize}
World models are sequences of states, actions and external events, described by atomic sentences (Herbrand models)

without time stamps \hspace{2cm} with time stamps
for efficiency \hspace{2cm} for logical semantics

States are sets of facts (or fluents):

\textit{fire} \hspace{2cm} \textit{fire}(10:15)

Events (including actions) cause state transitions:

\textit{eliminate} \hspace{2cm} \textit{eliminate}(10:15, 10:16)
The syntax of LPS

Reactive rules in First-order logic:

\[ \text{for all } X \left[ \text{antecedent } \rightarrow \text{there exists } Y \text{ consequent}\right] \]

or \[ \text{if } \text{antecedent } \text{then } \text{consequent}. \]

Clauses in logic programming form.

\[ \text{for all } X \left[ \text{there exists } Y \text{ conditions } \rightarrow \text{conclusion}\right] \]

or \[ \text{for all } X, \text{for all } Y \left[ \text{conditions } \rightarrow \text{conclusion}\right] \]

or \[ \text{conclusion if conditions.} \]
The syntax of LPS

without time stamps for readability
with time stamps for logical semantics

Reactive rules:

\[ \text{if } \text{fire} \text{ then } \text{deal-with-fire.} \]
\[ \text{if } \text{fire at } T1 \text{ then } \text{deal-with-fire from } T2 \text{ to } T3, T1 \leq T2. \]

Logic programs:

\[ \text{deal-with-fire if } \text{eliminate.} \]
\[ \text{deal-with-fire from } T1 \text{ to } T2 \text{ if } \text{eliminate from } T1 \text{ to } T2. \]
State transitions are described by programmable causal laws

Postconditions (effects):

\[ \text{ignite}(\text{Object}) \text{ initiates fire if flammable}(\text{Object}). \]
\[ \text{eliminate} \text{ terminates fire.} \]

Preconditions (constraints):

\[ \text{false} \text{ eliminate, fire, not water.} \]

Persistence (inertia):

A fact/fluent persists from the time it is initiated to the time it is terminated.

An emergent property, not used to generate or query states.
The logical nature of LPS

Computation generates a model of the world described by logic programs to make reactive rules true.

The imperative nature of LPS

Computation generates commands to perform actions to change the world to make consequents true whenever the antecedents of reactive rules become true.
LPS: Computation generates actions to make reactive rules true

- Actions
- The current state is updated destructively
- Actions are chosen and combined with external events
- External events
- Reactive rules whose conditions are true are triggered.
- Logic programs reduce goals to subgoals and actions
- Forward reasoning
- Backward reasoning
The Operational Semantics of LPS is Incomplete

It cannot *preventively* make rules true by making *antecedents* false:

\[
\text{if attacks}(X, \text{me}) \text{ from } T1 \text{ to } T2, \text{ not prepared-for-attack}(\text{me}) \text{ at } T2 \\
\text{then surrender}(\text{me}) \text{ from } T2
\]

It cannot *proactively* make rules true by making *consequents* true before *antecedents* become true:

\[
\text{if enter-bus}(\text{me}) \text{ from } T1 \text{ to } T2 \\
\text{then have-ticket}(\text{me}) \text{ at } T2.
\]
The explicit representation of time and causal laws facilitates the treatment of concurrency
% dining philosophers

fluents available(_).
actions pickup(_,_), putdown(_,_).

initially available(fork1), available(fork2), available(fork3),
available(fork4), available(fork5).

philosopher(socrates).
philosopher(plato).
philosopher(aristotle).
philosopher(hume).
philosopher(kant).

adjacent(fork1, socrates, fork2).
adjacent(fork2, plato, fork3).
adjacent(fork3, aristotle, fork4).
adjacent(fork4, hume, fork5).
adjacent(fork5, kant, fork1).
if philosopher(P) then
dine(P) from T1 to T2.

dine(P) from T1 to T3 if
adjacent(F1, P, F2),
pickup(P, F1) from T1 to T2,
pickup(P, F2) from T1 to T2,
putdown(P, F1) from T2 to T3,
putdown(P, F2) from T2 to T3.

pickup(P, F) terminates available(F).
putdown(P, F) initiates available(F).

false pickup(P, F), not available(F).
false pickup(P1, F), pickup(P2, F), P1 \= P2.
What happens if we replace:

`dine(P)` from T1 to T3 if
adjacent(F1, P, F2),
pickup(P, F1) from T1 to T2,
pickup(P, F2) from T1 to T2,
putdown(P, F1) from T2 to T3,
putdown(P, F2) from T2 to T3.

with:

`dine(P)` from T1 to T5 if
adjacent(F1, P, F2),
pickup(P, F1) from T1 to T2,
pickup(P, F2) from T2 to T3,
putdown(P, F1) from T3 to T4,
putdown(P, F2) from T4 to T5.
LPS includes composite event recognition

if possible fire detected at $T$ then respond to possible fire from $T$

possible fire detected at $T$ if
  heat-sensor detects high temperature in area A at $T_1$,
  smoke detector detects smoke in area A at time $T_2$,
  $|T_1 - T_2| \leq 60$ sec $\land$ max($T_1$, $T_2$, $T$)

respond to possible fire at $T$ if
  activate sprinkler in area A at $T_3$, $T < T_3 \leq T + 10$ sec,
  send security guard to area A at time $T_4$, $T_3 < T_4 \leq T_3 + 30$ sec

respond to possible fire at time $T$ if
  call fire department to area A at time $T_5$, $T < T_5 \leq T + 120$ sec
Overview

Introduction

LPS examples

LPS in relation to computer science

LPS implementation

Foundations of LPS and related work
Overview of the Tutorial

Introduction

LPS examples

LPS in relation to computer science

• Declarative representations have the frame problem

• Imperative representations perform destructive state transitions without a logical meaning

LPS implementation

Foundations of LPS and related work
The Problem: Two kinds of system with no obvious relationship:

- Logic-based systems (declarative)
- State transition systems (imperative)

The Solution: Reactive rules and logic programs

Computation generates a model of the world described by logic programs (beliefs), to make reactive rules (goals) true.
Declarative systems

*if A then B* means if *A* is true then *B* is true.

**Programming**
- Logic programing
- Functional programming

**Databases**
- Relational databases
- Datalog

**Artificial Intelligence**
- Knowledge representation
- Causal theories, etc.
The Frame Problem
(lack of destructive state transitions)

It is necessary to reason that

- $u$ is true at time $t+1$ because $u$ was true at time $t$ and $u$ was not terminated from $t$ to $t+1$.

- $v$ is true at time $t+1$ because $v$ was true at time $t$ and $v$ was not terminated from $t$ to $t+1$.

- $w$ is true at time $t+1$ because $w$ was true at time $t$ and $w$ was not terminated from $t$ to $t+1$. 
The Problem: Imperative languages are natural and efficient, but do not have a logical meaning

*if A then B* means change of state. e.g.
If A holds then do B (imperative).

**Programming**
- state charts
- abstract state machines

**Databases**
- active databases

**Artificial Intelligence**
- production systems
- agent languages
Two kinds of programming systems

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL

For transformational systems (e.g., many kinds of data-processing systems) one really has to specify a transformation, or function, so that an input/output relation is usually sufficient. While transformational systems can also be highly complex, there are several excellent methods that allow one to decompose the system’s transformational behavior into ever-smaller parts in ways that are both coherent and rigorous. Many of these approaches are supported by languages and implemented tools that perform very well in practice. We are of the opinion that for reactive systems, which present the more difficult cases, this problem has not yet been satisfactorily solved. Several important and promising approaches have been proposed, and Section 8 of this paper discusses a number of them. However, the e.g. logic programs e.g. production systems
STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL

Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Much of the literature also seems to be in agreement that states and events are \textit{a priori} a rather natural medium for describing the dynamic behavior of a complex system. See, for example, [7–9, 19, 23]. A basic fragment of such a description is a \textit{state transition}, which takes the general form “when event $\alpha$ occurs in state $A$, if condition $C$ is true at the time, the system transfers to state $B$”. Indeed, many of the informal exchanges concerning the dynamics of systems are of this nature; e.g., “when the plane is in cruise mode and switch $x$ is thrown it enters navigate mode”,

\begin{center}
36
\end{center}
From Abstract State Machine Tutorial presented by Egon Börger

**Definition of Abstract State Machines (Yuri Gurevich 1988)**

A sequential ASM is defined as a set of transition rules of form

```
if Condition then Updates
```

which transform first-order structures (the states of the machine), where the guard *Condition*, which has to be satisfied for a rule to be applicable, is a variable free first-order formula, and *Updates* is a finite set of function updates (containing only variable free terms) of form

```
f (t1, ..., tn) := t
```

The execution of these rules is understood as updating, in the given state and in the indicated way, the value of the function *f* at the indicated parameters, leaving everything else unchanged. (This proviso avoids the frame problem of declarative approaches.) In every state, all the rules which are applicable are simultaneously applied (if the updates are consistent) to produce the next state.
Conway’s game of life. We use Conway’s game of life to illustrate the unbounded synchronous parallelism of ASMs. Imagine a grid of square cells, elements of an abstract domain \( Cell \), which can be alive or dead. The rule of survival describing the behavior of a single cell states that a cell with 3 alive neighbors gets (or remains) alive, whereas a cell with less than 2 or more than 3 alive neighbors dies. We represent this rule using an abstract predicate \( alive \) on \( Cell \) together with a derived function \( aliveNeighb : Cell \rightarrow \mathbb{N} \) (not specified further here) which indicates for each cell the number of its alive neighbors. Then the rule of the game for a cell \( c \) is expressed by the following ASM:

\[
CONWAY(c) = \\
\quad \text{if } aliveNeighb(c) = 3 \text{ then } alive(c) := true \\
\quad \text{if } aliveNeighb(c) < 2 \text{ or } aliveNeighb(c) > 3 \text{ then } alive(c) := false
\]

\[
GAMEOFLIFE = \forall c \in Cell \text{ do } CONWAY(c)
\]

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
moves

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>dies</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>dies</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>dies</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td>block (stable)</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>blinker (period 2)</td>
</tr>
</tbody>
</table>
Two kinds of database systems: Active Databases and Deductive Databases (e.g. Datalog)
An Overview of Production Rules in Database Systems

Eric N. Hanson  Jennifer Widom

Database researchers have discovered that with the addition of production rules facilities, database systems gain the power to perform a number of useful database tasks with one uniform mechanism: they can enforce integrity constraints, monitor data access and evolution, maintain derived data, enforce protection schemes, maintain version histories, and more. (Previous support

There is a substantial body of work on another kind of database system with rules—deductive database systems. Deductive database systems are similar to conventional database systems in that they are passive, responding only to commands from users or applications. However, they extend conventional database systems by allowing the definition of PROLOG-like rules on the data and by providing a deductive inference engine for processing recursive queries using these rules. Deductive and active database rule systems are fundamentally different, and both types of rules could theoretically be present in a single system. We focus on active database systems and do
The distinction between goals and beliefs is fundamental in database systems.

From Datalog± (including ontologies and integrity constraints; Cali, Gottlob, Lukasziewicz; 2009, example 2.

\[
\begin{align*}
\text{Datalog rules = beliefs} \\
\text{manager}(X) & \rightarrow \text{employee}(X) \\
\text{Integrity constraints = goals} \\
\text{manager}(X) & \rightarrow \exists Y \text{ supervises}(X, Y) \\
\text{employee}(X), \text{ employer}(X) & \rightarrow \text{false} \\
\text{supervises}(X, Y), \text{ supervises}(X', Y) & \rightarrow X = X'
\end{align*}
\]
Reactive rules and logic programs:
It can be hard to tell the difference

AgentSpeak(L): BDI Agents speak out in a logical computable language

**Definition 5** If $e$ is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1,\ldots,h_n$ are goals or actions then $e:b_1 \land \ldots \land b_m \leftarrow h_1;\ldots;h_n$ is a plan. The expression to the left of the arrow is referred to as the *head* of the plan and the expression to the right of the arrow is referred to as the *body* of the plan. The expression to the right of the colon in the head of a plan is referred to as the context. For convenience, we shall rewrite an empty body with the expression true.

With this we complete the specification of an agent. In summary, a designer specifies an agent by writing a set of base beliefs and a set of plans. This is similar to a logic programming specification of facts and rules. However, some of the major differences between a logic
LPS and BDI agents compared

This “logic programming-like” plan in AgentSpeak

+location(waste,X):location(robot,X) &
  location(bin,Y)
  <- pick(waste);
  !location(robot,Y);
  drop(waste).

is a reactive rule (or goal) in LPS:

\[
\begin{align*}
\text{if} & \quad \text{location}(\text{waste}, X) \text{ at } T1, \text{location}(\text{robot, } X) \text{ at } T1, \text{location}(\text{bin, } Y) \text{ at } T1 \\
\text{then} & \quad \text{pick}(\text{waste}) \text{ from } T1 \text{ to } T2, \\
& \quad \text{move-to-location}(\text{robot, } Y) \text{ from } T2 \text{ to } T3, \\
& \quad \text{drop}(\text{waste}) \text{ from } T3 \text{ to } T4.
\end{align*}
\]
The distinction between goals and beliefs is the foundation of SBVR

SBVR – From Wikipedia:

“The Semantics of Business Vocabulary and Business Rules (SBVR) is an adopted standard of the Object Management Group (OMG) intended to be the basis for formal and detailed natural language declarative description of a complex entity, such as a business. “

From Baisley, Hall and Chapin:

“Distinguishing between guidance (rules that people break) and structural rules (rules about meaning) is very important in understanding business rules.”
SBVR - Example from Baisley, Hall and Chapin

It is obligatory that each person on a bus has a ticket.

A person on a bus either has a ticket or is breaking the rule.

It is logically necessary that each person on a bus has a ticket.

Being on a bus implies that there is a ticket.

These modalities are not nested as in normal modal logic.

LPS as an alternative to modal logic:

Goal: if a person is on a bus then the person has a ticket.

Belief: a person has a ticket if the person is on a bus.
Attributing Mental Attitudes to Social Entities: Constitutive Rules are Beliefs, Regulative Rules are Goals

Guido Boella\(^1\) and Leendert van der Torre\(^2\)

\(^1\) Dipartimento di Informatica - Università di Torino- Italy. E-mail: guido@di.unito.it
\(^2\) CWI Amsterdam and TU Delft - The Netherlands. E-mail: torre@cwi.nl

Abstract. In this paper, we propose a model of constitutive and regulative norms in a logical multiagent framework. We analyze the relationship between these two types of rules and explain similarities between them, using the metaphor of considering social entities - like normative systems, groups and organizations - as agents and of attributing them mental attitudes as well as an autonomous behavior. We argue that while constitutive norms expressing “counts-as” relations are modelled as the beliefs of social entities, regulative norms, like obligations, prohibitions and permissions, are modelled as their goals.
**Claims**

LPS gives a simple logic to

- reactive rules
- destructive updates

LPS is a simple alternative to

- modal logics of time, necessity and obligation.
Overview

Introduction

LPS examples

LPS in relation to computer science

**LPS implementation**

Foundations of LPS and related work
Overview

Introduction

LPS examples

LPS in relation to computer science

LPS implementation

Foundations of LPS and related work
Foundations of LPS and related work

Theorist

Abductive Logic Programming (ALP)

Related Frameworks

LPS can be extended

A theorist framework is a pair \( \langle P, A \rangle \) where

- \( P \) FOL theory.
- \( A \) sentences representing candidate assumptions.

Given a goal \( G \), the task is to generate some \( \Delta \subseteq A \) such that:

- \( P \cup \Delta \) logically implies \( G \) and
- \( P \cup \Delta \) is consistent.


Assumptions \( A \) can be default assumptions.
Abductive Logic Programming (ALP)

An abductive framework is a triple \( \langle P, I, A \rangle \) where

- \( P \) logic program.
- \( I \) integrity constraints.
- \( A \) atomic sentences.

Given a goal \( G \), the task is to generate some \( \Delta \subseteq A \) such that:

\[
P \cup \Delta \quad \text{solves } G
\]
\[
P \cup \Delta \quad \text{satisfies } I.
\]

There can be many such \( \Delta \).

It may be required to generate the best \( \Delta \) possible.

In applications to abduction \( \Delta \) explains observations \( G \).
In planning \( \Delta \) is a plan to achieve \( G \).
In default reasoning \( \Delta \) is a set of defeasible assumptions.
ALP alternative semantics

What does “$P \cup \Delta$ solves $G$” mean?

- $G$ is a theorem logically implied by the completion of $P \cup \Delta$?
- $G$ is true in the intended/standard model of $P \cup \Delta$?

What does “$P \cup \Delta$ satisfies $I$” mean?

- $I$ is consistent with $P \cup \Delta$?
- $I$ is consistent with the completion of $P \cup \Delta$?
- $I$ is true in the intended/standard model of $P \cup \Delta$?

In LPS there is no difference between $G$ and $I$:

$G$ and $I$ are both true in the intended/standard model of $P \cup \Delta$. 
LPS is an instance of a more general, simplified variant of ALP

An _abductive framework_ is a triple \((P, G, A)\) where

- \(P\) logic program.
- \(G\) sentences in FOL.
- \(A\) atomic sentences.

The _task_ is to generate some \(\Delta \subseteq A\) such that:

- \(G\) is true in the intended/standard model \(M\) of \(P \cup \Delta\).

If \(P\) is a set of Horn clauses,
then the intended/standard model \(M\) of \(P \cup \Delta\)
is the unique minimal model of \(P \cup \Delta\).

Advantages of the simplified semantics:

- \(G\) can be an arbitrary set of FOL sentences (+aggregation operators).
- \(\Delta\) can be infinite.

This is like the standard model of arithmetic (defined by Horn clauses) compared with first-order axioms for arithmetic.
LPS is an instance of the simplified variant of ALP

An LPS framework is a triple \( \langle P, G, A \rangle \) where

- \( P \) logic program.
- \( G \) reactive rules and constraints.
- \( A \) atomic sentences.

The task is to generate some \( \Delta \subseteq A \), such that:

- \( G \) is true in the intended/standard model \( M \) of \( P \cup F \cup \Delta \)
- where \( F \) is a set of frame axioms.

\( \Delta \) can be generated destructively without using \( F \).
But \( F \) is true in \( M \).

There can be many such \( \Delta \).
It is often desirable to generate the best \( \Delta \) possible.

In the current implementation,
the only way to indicate that one \( \Delta \) is better than another \( \Delta \)
is indirectly by ordering clauses.

Or by introducing extra conditions into clauses and rules.
Related frameworks


Combines classical (non-Herbrand) logical consequence for FOL with well-founded semantics for logic programs.

An _ALP framework_ is \( \langle P, I, A \rangle \) where given \( G \), the _task_ is to generate some \( \Delta \subseteq A \), such that:

- the completion of \( P \cup \Delta \) logically implies \( G \cup I \).

SCIFF extends the IFF proof procedure of Fung and Kowalski 1997.

Assumptions can be

- positive expectations of actions that ought to happen
- negative expectations of actions that ought not to happen.
METATEM: An Introduction

H. Barringer\textsuperscript{1}, M. Fisher\textsuperscript{2}, D. Gabbay\textsuperscript{3}, G. Gough\textsuperscript{1} and R. Owens\textsuperscript{4}

Consider a temporal sentence of the form:

\textbf{antecedent (about the past)} \Rightarrow \textbf{consequent (about the present and future)}

This can be interpreted as if the “antecedent (about the past)” is true then \textbf{do} the “consequent (about the present and future)". Adopting this imperative reading

future. Although we can adopt a declarative interpretation of the rules, for programming and execution purposes we take an imperative reading following a natural view of the way in which dynamic systems behave and operate, namely,

\textbf{on the basis of the past do the present and future.}
Computation in MetateM is model generation

3 Executing Temporal Logic

3.1 What Is Execution?

But, what does it mean to execute a formula, $\varphi$, of logic, $L$? In general, this means constructing a model, $M$, for $\varphi$, i.e.

$$M \models_L \varphi.$$ 

Typically, this construction takes place under some external constraints on $\varphi$, and many different models might satisfy $\varphi$. However, we note that:

- as $\varphi$ represents a declarative statement, then producing $M$ can be seen as execution in the declarative language $L$; and
- if $\varphi$ is a specification, then constructing $M$ can also be seen as prototyping an implementation of that specification.

MetaTem uses frame axioms to generate possible worlds simulates logic programs by reactive rules.
**Transaction logic programming – Bonner and Kifer 1993.**

*Transaction Execution Paths.* When a transaction is executed, the database may change, going from an initial state, $D_1$, to a final state, $D_n$. As shown in Section 2, the database may also pass through any number of intermediate states, $D_2, \ldots, D_{n-1}$, along the way. The sequence $\langle D_1, D_2, \ldots, D_{n-1}, D_n \rangle$ is the *execution path* of the transaction. It is said to have *length* $n$. The semantics of Transaction Logic is based on execution paths, or *paths* for short. As illustrated in Sections 2.6 and 7.7, paths

Unlike modal structures, truth in path structures is defined on paths, not states. For example, we would say that the path $\langle D, D + \{a\}, D + \{a, b\} \rangle$ satisfies the formula $a \cdot \text{ins} \otimes b \cdot \text{ins}$, since it represents an insertion of $a$ followed by an insertion of $b$. On the other hand, the path $\langle D, D + \{b\}, D + \{a, b\} \rangle$ does not satisfy this formula; instead, it satisfies the formula $b \cdot \text{ins} \otimes a \cdot \text{ins}$. This simple example illustrates a general idea in TR: the truth of a formula on a path corresponds to

Deductive databases as data structures.
The operational semantics performs destructive updates to a current state.
Reactive rules can be programmed/simulated.
**CHR** - Frühwirth, T., 2006, Constraint handling rules: the story so far.

### Declarative Semantics

**Simplification rule:** \[ H \leftrightarrow C \mid B \quad \forall x (C \rightarrow (H \leftrightarrow \exists y B)) \]

**Propagation rule:** \[ H \Rightarrow C \mid B \quad \forall x (C \rightarrow (H \rightarrow \exists y B)) \]

*Constraint Theory for Built-Ins*

- Head \( H \): non-empty conjunction of CHR constraints
- Guard \( C \): conjunction of built-in constraints
- Body \( B \): conjunction of CHR and built-in constraints (goal)

Soundness and Completeness based on logical equivalence of states in a computation.
CHR compared with LPS

Propagation rules can simulate production rules.

Linear logic semantics can simulate destructive updates.
Conclusion

LPS is a declarative, human-oriented language with an imperative interpretation.

Logic programs = beliefs
as in LogicBlox and Google (Yedalog)

Reactive rules = goals
as in production systems

Destructive change of state
as in the real world
Try it!
The End
Thank You
LPS (Logic-based Production Systems)
The current implementation: practical aspects

RuleML LPS Tutorial
London, July 12, 2017

Robert Kowalski and Fariba Sadri
Imperial College London

Miguel Calejo
Interprolog.com
Logical Production Systems Resources

• Main URL: http://lps.doc.ic.ac.uk
• Online playground:
  • http://lpsdemo.interprolog.com
  • Only this week: http://lpsdemo2.interprolog.com
• Source code:
  • https://bitbucket.org/lpsmasters/lps_corner
• For local installation: SWI Prolog
Pragmatic perspective

• LPS is a SWI Prolog superset, adding:
  • Fluent and event rules and declarations
  • Post conditions, integrity constraints
  • Reactive rules, external observations:

• LPS surface syntax is translated into Prolog code and facts, that are interpreted by the LPS runtime

• Executing a program is:
  • Simulating it over a period of time, feeding it with observations (events)
  • No user input yet (well... manual textual input)
LPS surface syntax recap.

- **Literals (sorted symbols)**
  - fluent at \( t_1 \), event from \( t_1 \) to \( t_2 \), (timeless) \( p \) (Prolog-like)
  - Internally: holds(fluent, \( t_1 \)), happens(event, \( t_1, t_2 \)), \( p \)

- **Composite events and intensional fluents**
  - macro from \( T_1 \) to \( T_n \) if \( e_1 \) from \( T_1 \) to \( T_2 \), \( e_2 \) from...
  - complexFluent at \( T \) if fluent1 at \( T \), \( p \), ...

- **Post-conditions: how actions affect fluents**
  - \( e \) initiates/terminates \( f \) if condition

- **Pre-conditions: prevent “bad” actions**
  - false someFluents, incompatibleActions.

- **Last but not least: reactive rules**
  - If antecedent then consequent.

- **The rest: Prolog code**

LPS syntax = Prolog + LPS constructs
First, a word about SWISH 💕 ...

A major innovation in logic programming!

- Open source collaborative Prolog IDE on a web site
- Notebooks for Prolog (as Jupyter does for Python)
- Self-contained, efficient; SWI-Prolog is the server
- Besides cute and friendly:
  - Just the visible tip of the powerful SWI Prolog IDE machinery
    - Extensible!
  - Solid **HTTP app server** built-in
  - **Term rendering** framework, DCGs + Javascript
  - Best of breed Javascript libraries

...a dream come true for developers of logic programming dialects...
Examples menu: First Steps with LPS, LPS Examples
• Open or write program on the left pane
  • If you Save..., you can later reopen with “Open Recent...”
  • Either Download or copy to your plain text editor, to be safe
• Errors reported in red – they must be fixed.
• Type goal on query panel on the bottom right
  • go. % logs events and states over time
  • go(Timeline). % nicer: shows fluent/event timeline
  • dump. % see transformed program (Prolog)
• Inspect the timeline... or see the log with fluent states and events

(*): you may also install your own local site instance in minutes
Making Life more interesting: 2D world

• Timelines and textual logs are not the best for Game of Life...
• Some fluents and events are best shown in 2D space..
  • ...at each cycle: LPS cycle states drawn over (real) time
• “annotation” \(d(LPS\text{–term, Properties})\):
  • Generates (sprites for frames of) a 2D animation
  • Declarative properties define display aspect of fluents, events... as well as ‘timeless’, non changing, background geometries
• **Game of Life** facelifted
• Let’s revisit **bubble sort, dad saving energy**...
• 2d **documentation**
Game of life, live

• Examples / LPS Examples menu
• Uncomment d/2 declarations

• 1 second per LPS cycle
  • 60, 30, 15 frames for longer simulations

• Events can be shown too
  • Fade in/out around the cycle transition

*Timeline still available (click top left corner of animation)*
About the implementation

• Engine history
  • Current main engine:
    • Sadri with student; later RK, MC
    • Started with XSB, moved to SWI in 2016
    • SWI specifics: delimited continuations variant; SWISH support
  • Other engines (Java):
    • cf. “bob the simplebot” sources from [LPS main page](#)

• Installation alternatives
  • SWI
    • Barebones version
    • SWISH: local, cloud
  • XSB
    • Barebones
    • InterProlog Studio
Source code tour from 30k feet - engine

https://bitbucket.org/lpsmasters/lps_corner/src

- interpreter.P - Core engine, internal syntax
  - go(File,Options), cycle(ReactiveRules,Goals)

- psyntax.P - Surface syntax processing:
  - Generates LPS internal syntax (Prolog) from LPS file
  - syntax2p/5, golps(File,Options).

- Some options:
  - make_test: records trace of present execution in .lpst (“correct test result”) file
  - dc: delimited continuations variant
  - cycle_hook(Predicate,Fluents,Actions): e.g. callbacks to another language
  - timeout(Seconds)

- Test suite:
  - interpreter: test_examples. *(verifies results for all examples/ *)
Source code tour from 30k feet—SWISH support

• Running LPS programs in the cloud
  • LPS engine preloaded
  • Temporary ‘user’ module; term_expansion/2

• Editor:
  • syntax coloring: extra info to feed SWI’s prolog_colouring.pl
  • error reporting: plug into SWI’s print_message/2 machinery
  • lps_corner/swish/user_module_file.pl

• Renderers for LPS executions
  • Timelines
    • http://visjs.org
    • lps_corner/swish/lps_timeline_renderer.pl
  • 2D world
    • http://paperjs.org
    • lps_corner/swish/lps_2d_renderer.pl
Developing software with LPS

_Preliminary LPS APIs, but lots of APIs to build upon_

- **Platforms**
  - Prolog app
    - import psyntax, use golps(..) predicate
    - Check state/1, happens/3
  - Java app
    - [LPS Engine](#) example (XSB): a simple synchronous LPS cycle handler
      - Shouldn’t be hard to implement [for SWI](#)
    - [Bot the Simplebot](#) : more extensive Java API, more limited (Java) LPS engine
  - Javascript app (TBD)
    - Embed in a [pengines](#) server (SWISH RPC infrastructure)
  - Python via [PYSSWIP](#)
- **LPS agents**
  - LPS cycle loop may be used for interfacing real world events
  - Games, IoT, smart (logical) contracts...
LPS (Logic-based Production System) Examples

Robert Kowalski and Fariba Sadri
Imperial College London
Miguel Calejo
Interprolog.com

Implementations of LPS

- LPS has been implemented in
  - Java
  - Sicstus Prolog
  - XSB Prolog
  - SWI Prolog and SWISH.

The SWISH implementation is the most reliable.

http://lpsdemo.interprolog.com/

Examples

1. Fire
2. Recurrent Fire
3. Map colouring
4. Bubble sort
5. Quicksort
6. Dining philosophers
7. Tower building
8. Tic-tac-toe
9. Turing
10. Prisoner’s dilemma
11. Trash

% Fire Example
% Declarations
maxTime(5).
fluent fire.
actions eliminate, escape.

% Initial state
initially fire.

% Goals: Reactive Rules
if fire at T1 then deal-with-fire from T1 to T2.

% Beliefs: Logic Programs
deal-with-fire from T1 to T2 if eliminate from T1 to T2.
deal-with-fire from T1 to T2 if escape from T1 to T2.

% Causal theory
eliminate terminates fire.
% Fire Example (run with go)
% Declarations:
maxTime(5).
fluents fire.
actions eliminate, escape.
% Initial state
initially fire.

% Goals: Reactive Rules
if fire at T1 then deal-with-fire from T1 to T2.

% Beliefs: Logic Programs
deal-with-fire from T1 to T2 if eliminate from T1 to T2.
deal-with-fire from T1 to T2 if escape from T1 to T2.

% Causal theory
eliminate terminates fire.

Fire example extended with recurrent fires

maxTime(10).
fluents fire, water.
actions eliminate, escape, ignite(_,), refill.
% Observations:
observe ignite(sofa) from 1 to 2.
observe ignite(bed) from 4 to 5.
observe refill from 7 to 8.
initially water.
% Time-independent information
flammable(sofa).
flammable(bed).

Of course we can also add the following reactive rule to ensure we always refill after using the water.

if not water at T1 then refill from T2 to T3.
**Actions and Events**

In LPS state transitions are achieved by:

- **Actions**: Under the control of the agent
- **Events**: External and observed and assimilated by the agent

In principle and in later implementations events can be observed dynamically.

Here, for simplicity, we add them to the program at the start.

Actions can be executed concurrently.

---

**Map Colouring example**

- Concurrent actions.

- Here simply combining a reactive rule and a causal theory with an action precondition solves this problem.

```plaintext
% The map colouring problem.
maxTime(5).
actions paint(_,_).
country(sweden).
country(norway).
country(finland).
country(russia).
colour(red).
colour(yellow).
colour(blue).
adjacent(sweden,norway).
adjacent(sweden,finland).
adjacent(norway,finland).
adjacent(norway,russia).
adjacent(finland,russia).

% Every country must be painted a colour.
if country(X) then colour(C), paint(X,C) from 1 to 2.

% Two adjacent countries cannot be painted the same colour.
false paint(X,C), adjacent(X,Y), paint(Y,C).

/* We can also write
if country(X) then colour(C), paint(X,C) from _T1 to _T2.
*/
```
% The map colouring problem.

maxTime(5).
actions paint(_, _).
country(az).
country(iz).
country(oz).
country(uz).
colour(red).
colour(yellow).
colour(blue).
adjacent(az, iz).
adjacent(az, oz).
adjacent(iz, oz).
adjacent(iz, uz).
adjacent(oz, uz).

if country(X) then colour(C), paint(X, C) from 1 to 2.
false paint(X, C), adjacent(X, Y), paint(Y, C).

% bubble sort with relational data structure.
maxTime(5).
fluents location(_, _).
actions swap(_,_,_,_).
initially location(d, 1), location(c, 2), location(b, 3), location(a, 4).
if location(X, N1) at T1, N2 is N1 + 1, location(Y, N2) at T1, Y@<X then swapped(X, N1, Y, N2) from T2 to T3.
swapped(X, N1, Y, N2) from T1 to T2 if location(X, N1) at T1, location(Y, N2) at T1, Y@<X, swap(X, N1, Y, N2) from T1 to T2.
swapped(X, N1, Y, N2) from T to T if location(X, N1) at T, location(Y, N2) at T, X@<Y.
swap(X, N1, Y, N2) initiates location(X, N2).
swap(X, N1, Y, N2) initiates location(Y, N1).
swap(X, N1, Y, N2) terminates location(X, N1).
swap(X, N1, Y, N2) terminates location(Y, N2).
false swap(X, N1, Y, N2), swap(Y, N2, Z, N3).

Bubble sort

Keep swapping adjacent elements that are out of order until the array is ordered.

d c b a

c d b a

c b d a

And so on ....

a b c d

LPS executes actions concurrently

Time 1

d c b a

Time 2
c d a b

Time 3
c a d b

Time 4
a c b d

Time 5
a b c d
Teleo-reactivity

- If later an object is moved, the same program will sort them again.
- observe swap(a,1,c,3) from 11 to 12.
- observe swap(b,2,c,3) from 15 to 16.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>b</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Time 11

Time 12

maxTime(20).

fluents location(_, _).

actions swap(_, _, _, _).

observe swap(a,1,c,3) from 11 to 12. %new

observe swap(b,2,c,3) from 15 to 16. %new

initially location(d, 1), location(c, 2), location(b, 3), location(a, 4).

if location(X, N1) at T1, N2 is N1 +1, location(Y, N2) at T1, Y@<X then

swapped(X, N1, Y, N2) from T2 to T3.

% swapped may not work if the order of the two clauses below is % reversed. Perhaps for good reasons.

% swapped(X, N1, Y, N2) from T1 to T2
if location(X, N1) at T1, location(Y, N2) at T1, Y@<X, swap(X, N1, Y, N2) from T1 to T2.

swapped(X, N1, Y, N2) from T to T
if location(X, N1) at T, location(Y, N2) at T, X@<Y.

swap(X, N1, Y, N2) initiates location(X, N2).

swap(X, N1, Y, N2) initiates location(Y, N1).

swap(X, N1, Y, N2) terminates location(X, N1).

swap(X, N1, Y, N2) terminates location(Y, N2).

false swap(X, N1, Y, N2), swap(Y, N2, Z, N3).

Classic Quicksort Example –
LPS can call Prolog

- LPS programs can include Prolog clauses.

maxTime(2).

events request(_).

actions announce(_).

observe request(sort([2,1,4,3]) from 0 to 1.

if request(sort(X)) from T1 to T2
then quicksort(X, Y), announce(sorted(Y)) from T2 to T3.

quicksort([X|Xs],Ys) :-
partition(Xs,X,Left,Right),
quicksort(Left,Ls),
quicksort(Right,Rs),
append(Ls,[X|Rs],Ys).

quicksort([],[]).

partition([X|Xs],Y,[X|Ls],Rs) :-
X <= Y, partition(Xs,Y,Ls,Rs).

partition([X|Xs],Y,[X|Ls],Rs) :-
X > Y, partition(Xs,Y,Ls,Rs).

partition([L],[L]).
Let us summarise the LPS language

In principle:

Anything you can do with Prolog you can also do with LPS.

In addition:

There is more.

### The LPS language

#### Declarations

- `maxTime(_)`
- `fluents` ...
- `actions` ...
- `events` ...

/* These declarations are used to do syntax checking of the rest of the program. They are used by the interpreter to decide what is an executable action, etc. */

#### Inputs

- `initially` ...
- `observe` ...
- `if .. then ..`

#### Reactive rules

- `.. if ..`
- `.. if ..`

#### Clauses

- `<event or action> initiates <fluent>` if ...
- `<event or action> terminates <fluent>` if ...
- `false ..`

#### Causal theory

### The Dining Philosophers

maxTime(7).
fluents available(_).
actions pickup(_,_), putdown(_, _).
initially available(fork1), available(fork2), available(fork3), available(fork4), available(fork5).

philosopher(p1).
philosopher(p2).
philosopher(p3).
philosopher(p4).
philosopher(p5).
adjacent(fork1, p1, fork2).
adjacent(fork2, p2, fork3).
adjacent(fork3, p3, fork4).
adjacent(fork4, p4, fork5).
adjacent(fork5, p5, fork1).
% dining philosophers

if philosopher(P)
then dine(P) from T1 to T2.

dine(P) from T1 to T2 if
adjacent(F1, P, F2),
pickup(P, F1) from T1 to T2,
putdown(P, F1) from T2 to T3,
putdown(P, F2) from T2 to T3.

pickup(P, F) terminates available(F).
putdown(P, F) initiates available(F).
false pickup(P, F), not available(F).
false pickup(P1, F), pickup(P2, F), P1 ≠ P2.

Tower Building Example

Initial State

Goal State

Behaviour
Tower Building Example

maxTime(12).
fluents location(_,_), requested.
actions move(_,_).
i initially requested, location(floor), location(b), location(e), location(a), location(c).
% observe move(a,e) from 4 to 5.
if requested then make_tower([a,b,c,floor]) from _T1 to _T2.
if requested then make_tower([f,e,d,floor]) from _T1 to _T2.

% clear(Block) at T if Block \not= floor, not location(_,Block) at T.
clear(floor) at _.
make_tower([Block,floor]) from T1 to T2 if
make_on(Block,floor) from T1 to T2.
make_tower([Block,Place|Places]) from T1 to T3 if
Place \not= floor, make_tower([Place|Places]) from T1 to T2,
make_on(Block,Place) from T2 to T3.
make_tower([floor]) from _T1 to _T2 if true.

make_on(Block,Place) from T1 to T4 if
not location(Block,Place) at T1,
make_clear(Place) from T1 to T2,
move(Block,Place) from T2 to T3,
move(_,Place) from T3 to T4.
make_on(Block,Place) from T to _T if
location(Block,Place) at T.
make_clear(Place) from _T to T if
clear(Place) at T.
make_clear(Block) from T1 to T2 if
location(Block1,Block) at T1,
make_on(Block1,floor) from T1 to T2.
move(Block,Place) initiates location(Block,Place).
move(_,_) terminates location(Block,Place).
false move(Block,P1), move(Block,P2), P1,P2
/**<examples>
  : godo(Timeline).
  */