RuleML as a Declarative Language for Inputs, Outputs, and Background Knowledge in Inductive Programming

Harold Boley, UNB, Faculty of Computer Science

Dagstuhl Seminar 15442
Approaches and Applications of Inductive Programming
October 25-30, 2015
Schloss Dagstuhl – Leibniz Center for Informatics, Germany
Abstract

RuleML is a system of families of languages for Web rules connecting related efforts such as: SWRL, SWSL, and RIF (W3C); SBVR, PRR, API4KP, and OntoIOp (OMG); Common Logic (ISO); and LegalRuleML (OASIS).

Inductive Programming can use RuleML to represent inputs, outputs, and background knowledge, since:
a) it combines relational/logical and equational/functional representations, e.g. Prolog-like relations and functions defined as oriented (conditional) equations;
b) its canonical XML format allows modular (Relax NG-schema) validation of (challenge/benchmark/…-)library entries, each w.r.t. the most precise language;
c) RuleML/XML also allows (XSLT) transformation, e.g. to favorite presentation syntaxes of library users and to other XML-based standards;
d) its engines permit deduction/querying on induced programs/rulebases;
e) its canonical syntax, validation methods, interoperation techniques, as well as, e.g., operational and model-theoretic semantics can support the evaluation of Inductive Programming systems.
RuleML Basics

• RuleML includes the three declarative language paradigms of Functional, Logic, and Functional-Logic Programming
• The system of RuleML's Web rules is defined and validated through schema languages (mainly, Relax NG) initially developed for Web documents and data in XML and later transferred to other formats such as JSON
• Rulebases in RuleML/XML are translated with XML-aware transformation languages (mainly, XSLT) for normalization, presentation, and interoperation
• Semantic styles describe the meaning of RuleML languages such as Datalog and Hornlog (e.g., with Herbrand models)
• Engines permit the execution/querying of programs/rulebases (e.g., the Relfun, OO jDREW, Prova, and PSOATransRun engines)
RuleML System of Families of Languages

Deliberation
- HOL
- FOL

Consumer

Reaction
- KR
- ECAP
- CEP
  - Trigger (EA)
  - Production (CA)

Derivation
- Fact
- Query
  - Hornlog
  - Datalog

Relations:
- subClassOf
- Overlaps
- Syntactic specialization of
Functional Programs: Inputs (1)

**Input** – Number Series: 1, 3, 5, ...

Representing sequence as a set of **ground equations** as in IFP

(cf. [Igor2](#))

Igor2:

\[
\text{eq Plustwo}((s \ 0) \ \text{nil}) = s^3 \ 0
\]

\[
\text{eq Plustwo}((s^3 \ 0) \ (s \ 0) \ \text{nil}) = s^5 \ 0
\]

\[
\text{eq Plustwo}((s^5 \ 0) \ (s^3 \ 0) \ (s \ 0) \ \text{nil}) = s^7 \ 0
\]

RuleML/Relfun: *

\[
\text{plusTwoF}([s[0]]) :\& \ s^3[0].
\]

\[
\text{plusTwoF}([s^3[0], s[0]]) :\& \ s^5[0].
\]

\[
\text{plusTwoF}([s^5[0], s^3[0], s[0]]) :\& \ s^7[0].
\]

* In Relfun and Prolog presentation and RuleML serialization, \(^j\) is meta-syntax.
Functional Programs: Inputs (2)

RuleML/Reلفun (presentation syntax):

\[
\text{plusTwoF([s[0]]) :& s^3[0].} \quad \% \text{plusTwoF([s[0]]) } \rightarrow \text{s}[s[s[0]]] \\
\text{plusTwoF([s^3[0], s[0]]) :& s^5[0].} \\
\text{plusTwoF([s^5[0], s^3[0], s[0]]) :& s^7[0].}
\]

RuleML/XML (serialization syntax):

\[
<\text{Equal oriented}="yes"> \\
<\text{Expr}> \\
<\text{Fun per}="value">\text{plusTwoF}</\text{Fun}> \\
<\text{Plex}> \\
<\text{Expr}><\text{Fun per}="copy"><\text{s}</\text{Fun}><\text{Data}>0</\text{Data}></\text{Expr}> \\
<\text{Plex}> \\
</\text{Expr}> \\
<\text{Expr}><\text{Fun per}="copy"><\text{s}^3</\text{Fun}><\text{Data}>0</\text{Data}></\text{Expr}> \\
</\text{Equal}> \\
<\text{Equal oriented}="yes"> \ldots </\text{Equal}> \\
<\text{Equal oriented}="yes"> \ldots </\text{Equal}>
\]
Functional Programs: Outputs (1)

**Output** – General Rule: \( f(1) = 1, \quad f(n + 1) = f(n) + 2 \)

Representing sequence as a set of *functional equations*

Haskell:

```haskell
data Peano = 0 | S Peano
f (S 0) = (S 0)
f (S n) = add (f n) (S (S 0))
```

RuleML/Relfun:

```ml
f(s[0]) :& s[0].
f(s[N]) :& add(f(N), s[s[0]]).
```

* Number series map natural numbers \( n = 1, 2, 3, \ldots \) into integers. Guard/condition for \( n \neq 0 \) could be included here and later on.
Functional Programs: Outputs (2)

RuleML/Relfun:

\[
\begin{align*}
  f(s[0]) & : & s[0], \\
  f(s[N]) & : & add(f(N), s[s[0]]). \quad *
\end{align*}
\]

RuleML/XML:

\[
\begin{align*}
  & <\text{Equal oriented}="\text{yes}" > \\
  & \quad <\text{Expr}> \\
  & \quad \quad <\text{Fun per}="\text{value}" >f<\/Fun> \\
  & \quad \quad <\text{Expr}> <\text{Fun per}="\text{copy}" >s<\/Fun> <\text{Data}>0<\/Data> \</Expr> \\
  & \quad \quad \</Expr> \\
  & \quad <\text{Expr}> <\text{Fun per}="\text{copy}" >s<\/Fun> <\text{Data}>0<\/Data> \</Expr> \\
  & \quad \</Equal> \\
  & \quad <\text{Equal oriented}="\text{yes}" > * \\
  & \quad <\text{Expr}> \\
  & \quad \quad <\text{Fun per}="\text{value}" >f<\/Fun> \\
  & \quad \quad <\text{Expr}> <\text{Fun per}="\text{copy}" >s<\/Fun> <\text{Var}>N<\/Var> \</Expr> \\
  & \quad \quad \</Expr> \\
  & \quad <\text{Expr}> <\text{Fun per}="\text{value}" >add<\/Fun> \\
  & \quad <\text{Expr}> \\
  & \quad \quad <\text{Fun per}="\text{value}" >f<\/Fun> \\
  & \quad \quad <\text{Var}>N<\/Var> \\
  & \quad \quad \</Expr> \\
  & \quad <\text{Expr}> <\text{Fun per}="\text{copy}" >s^2<\/Fun> <\text{Data}>0<\/Data> \</Expr> \\
  & \quad \</Expr> \\
  & \</Equal>
\]

* Guard for \( n \neq 0 \) would result in conditional equations such as \( f(s[N]) : \neq (N,0) \ & add(f(N), s[s[0]]). \)
Functional Programs: Background Knowledge (1)

**Background Knowledge** – General Rules

Representing arithmetic etc. as a set of **functional equations**

Haskell:

```haskell
add 0 n = n
add (S m) n = add m (S n)
```

... 

RuleML/Relfun:

```plaintext
add(0, N) :& N.
add(s[M], N) :& add(M, s[N]).
```

...
RuleML/Relfun:

add(0, N) &: N.
add(s[M], N) &: add(M, s[N]).

RuleML/XML:

<Equal oriented="yes">
  <Expr>
    <Fun per="value">add</Fun>
    <Data>0</Data>
    <Var>N</Var>
  </Expr>
</Equal>
<Equal oriented="yes">
  <Expr>
    <Fun per="value">add</Fun>
    <Expr><Fun per="copy">s</Fun><Var>M</Var></Expr>
    <Var>N</Var>
  </Expr>
</Equal>

Functional Programs: Background Knowledge (2)
Relational (Logic) Programs: Inputs (1)

**Input** Number Series: 1, 3, 5, ...

Representing sequence as a set of **ground atoms** as in ILP

**Prolog:**

```
plusTwoR([s(0)], s^3(0)).  % plusTwoR([s(0)], s(s(s(0))))).  
plusTwoR([s^3(0), s(0)], s^5(0)).  
plusTwoR([s^5(0), s^3(0), s(0)], s^7(0)).
```

**RuleML/Relfun:**

```
plusTwoR([s[0]], s^3[0]).  % plusTwoR([s[0]], s[s[s[0]]])).  
plusTwoR([s^3[0], s[0]], s^5[0]).  
plusTwoR([s^5[0], s^3[0], s[0]], s^7[0]).
```
Relational Programs: Inputs (2)

RuleML/Relfun:

plusTwoR([s[0]], s^3[0]).
plusTwoR([s^3[0], s[0]], s^5[0]).
plusTwoR([s^5[0], s^3[0], s[0]], s^7[0]).

RuleML/XML:

<Atom>
  <Rel>plusTwoR</Rel>
  <Plex>
    <Expr><Fun per="copy">s</Fun><Data>0</Data></Expr>
  </Plex>
  <Expr><Fun per="copy">s^3</Fun><Data>0</Data></Expr>
</Atom>
<Atom> . . . </Atom>
<Atom> . . . </Atom>
Relational Programs: Outputs (1)

**Output** – General Rule: \( r(1, 1), \quad r(n + 1, a + 2) \iff r(n, a) \)

Representing sequence as a set of *relational clauses*

Prolog (successor arithmetic):

\[
\begin{align*}
r(s(0), s(0)). \\
r(s(N), s(s(A))) & : - r(N, A).
\end{align*}
\]

RuleML/Relfun (successor arithmetic):

\[
\begin{align*}
r(s[0], s[0]). \\
r(s[N], s[s[A]]) & : - r(N, A).
\end{align*}
\]
Relational Programs: Outputs (2)

RuleML/Relfun:

\[ r(s[0], s[0]). \]
\[ r(s[N], s[s[A]]) :- r(N, A). \]

RuleML/XML:

\[
<Atom>
  <Rel>r</Rel>
  <Expr><Fun per="copy">s</Fun><Data>0</Data></Expr>
  <Expr><Fun per="copy">s</Fun><Data>0</Data></Expr>
</Atom>

<Implies>
  <Atom><Rel>r</Rel><Var>N</Var><Var>A</Var></Atom>
  <Atom>
    <Rel>r</Rel>
    <Expr><Fun per="copy">s</Fun><Var>N</Var></Expr>
    <Expr><Fun per="copy">s^2</Fun><Var>A</Var></Expr>
  </Atom>
</Implies>
Relational-Functional (Functional-Logic) Programs

Output – General Rule: \( rf(1, 1), \ rf(m, b) \iff rf(m - 1, a) \land b = a + 2 \) *

Representing sequence as a set of relational-functional clauses

Prolog (decimal arithmetic):

\[
\begin{align*}
rf(1, 1). \\
rf(M, B) :\text{-} N & \text{ is } M - 1, \ rf(N, A), \ B \text{ is } A + 2. * \\
\end{align*}
\]

RuleML/Relfun (successor arithmetic):

\[
\begin{align*}
rf(s[0], s[0]). \\
rf(s[N], B) :\text{-} rf(N, A), \ B & = \text{add}(A, s[s[0]]). \\
\end{align*}
\]

* Guard/condition for \( m > 1 \) could be included here.
Top-Level Terminology for Functions from (I)FP, Relations from (I)LP, and Their Combinations from (I)FLP *

- (I)FP: Function
- (I)LP: Relation (or Predicate)
- (I)FLP

* Slides 15-24 are adapted from http://cs.unb.ca/~boley/FLP/cs6905FLP.pdf.
Amalgamation/Integration Preview

- Functional-Logic **Amalgamation**: Function and relation calls can be combined in the same definition where appropriate.

- Functional-Logic **Integration**: Functions and relations can inherit each others’ expressiveness; e.g., in FLP certain functions – even when mapping from ground lists to ground lists – can be more easily defined using intermediate non-ground lists* (generally, partial data structures), as pioneered by relation definitions in LP.

- Partial data structures may be dynamically generated with fresh variables that make operation calls succeed (paradigm: zip or pairlists function).

* FLP operation: ground non-ground ground

Variable-containing terms
Functional-Relational Call Amalgamation: Quicksort Function as Background Knowledge

Directed, Conditional Equations:

qsort([]) :& [].
qsort([X|Y]) :-
    partition(X,Y,Sm,Gr) &
    cat(qsort(Sm),tup(X|qsort(Gr))).

Logic Rules & Fact:
partition(X,[Y|Z],[Y|Sm],Gr) :-
    <(Y,X), partition(X,Z,Sm,Gr).
partition(X,[Y|Z],Sm,[Y|Gr]) :-
    <(X,Y), partition(X,Z,Sm,Gr).
partition(X,[X|Z],Sm,Gr) :-
    partition(X,Z,Sm,Gr).
partition(X,[],[],[]).

Auxiliary Function (append or catenate):
cat([],L) :& L.
cat([H|R],L) :& tup(H|cat(R,L)).
Higher-Order Operations Defined: Quicksort Parameterized by Comparison Relation

Functional and relational arguments plus values. User-defined comparison relations Cr. **Restriction** to **named** functions and relations (no \( \lambda \)-expressions), as they are dominant in practice and more easily integrated (avoids \( \lambda \)/logic-variable distinction and higher-order unification): apply-reducible to 1\(^{st} \) order.

\[ \text{qsort}[\text{Cr}](\{X | Y\}) :\]
\[ \text{partition}[\text{Cr}](X,Y,\text{Sm},\text{Gr}) \& \]
\[ \text{cat}({\text{qsort}[\text{Cr}]}(\text{Sm}),\text{tup}(X | {\text{qsort}[\text{Cr}]}(\text{Gr})))]. \]

\[ \text{partition}[\text{Cr}](X,[Y | Z],[Y | \text{Sm}],\text{Gr}) :\]
\[ \text{Cr}(Y,X), \text{partition}[\text{Cr}](X,Z,\text{Sm},\text{Gr}). \]

\[ \ldots \]

\[ \text{before}([X1,Y1],[X2,Y2]) :\] string<(X1,X2). \]
Higher-Order Operations Called: Quicksort Parameterized by Comparison Relation

Cr bound to <:
rfi-p> qsort[<](3,1,4,2,3)
[1,2,3,4]

Cr bound to before:
rfi-p> qsort[before](([d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6])
([a,Y2],[d,Y1],[l,Y3],[s,Y6])
Y4=Y3
Y5=Y2

Prompt symbol of online relfun interpreter with prolog-like syntax
Logic Variables and Non-Ground Terms: pairlists Operation

Function calls can – like relation calls – use (free) logic variables as actual *arguments* and, additionally, return them as *values*. Likewise, *non-ground terms*, which contain logic variables, are permitted. Processing is based on unification:

Call with R creates inner Y1,Y2, ..., used as 2\textsuperscript{nd} pair elements

\[
\text{pairlists}([],[]) :& [].
\]

\[
\text{pairlists}([X|L],[Y|M]) :& \text{tup}([X,Y]\mid \text{pairlists}(L,M)).
\]

\[\text{rfi-p} > \text{pairlists}([d,a,l,l,a,s],R)\]

\[
[[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]]
\]

\[R=[Y1,Y2,Y3,Y4,Y5,Y6]\]

Non-ground pair list term (‘partial data structure’) containing six logic variables

Flat list of these logic variables
Function Calls Nested in Operation Calls: numbered Operation

Call-by-value nestings allow (built-in and user-defined) functions to be nested into other such functions or relations. Built-in function + nested here into user-defined relation numbered

numbered([],N).

numbered([[X,N]|R],N) :- numbered(R,+(N,1)).

rfi-p> numbered([[a,Y2],[d,Y1],[l,Y3],[s,Y6]],1)
true
Y2=1, Y1=2, Y3=3, Y6=4
Integrated Functional-Logic Programming Using Intermediate Non-Ground Terms: 
**serialise** Operation – IFLP Challenge

Transform a list of symbols into the list of their lexicographic serial rank numbers.

**Input Examples:**
\([a,d] \rightarrow [1,2], \quad [d,a,s] \rightarrow [2,1,3], \quad [d,a,l,l,a,s] \rightarrow [2,1,3,3,1,4]\)

**Desired Specific Output for, e.g., 3\(^{rd}\) Example:**
```
rfi-p> numbered(qsort[before](pairlists([d,a,l,l,a,s],R)),1) & R
[2,1,3,3,1,4], \quad R=[2,1,3,3,1,4]
```

**Desired General Output by Abstraction** \([d,a,l,l,a,s] = L:\)
```
serialise(L) :- numbered(qsort[before](pairlists(L,R)),1) & R.
```
Derivation of the \texttt{serialise} Solution

\[
\text{rfi-p> pairlists([d,a,l,l,a,s],R)}
\]
\[
[[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]]
\]
\[
R=[Y1,Y2,Y3,Y4,Y5,Y6]
\]

Intermediate non-ground pair list (unsorted)

\[
\text{rfi-p> qsort[before]}([[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6])}
\]
\[
[[a,Y2],[d,Y1],[l,Y3],[s,Y6]]
\]
\[
Y4=Y3
\]
\[
Y5=Y2
\]

Intermediate non-ground pair list (sorted, w/o ‘duplicates’)

\[
\text{rfi-p> numbered([a,Y2],[d,Y1],[l,Y3],[s,Y6]],1)}
\]
\[
\text{true}
\]
\[
Y2=1, \ Y1=2, \ Y3=3, \ Y6=4
\]

Bindings of inner variables produced

\[
\text{serialise([d,a,l,l,a,s]) :-}
\]
\[
\text{numbered(qsort[before](pairlists([d,a,l,l,a,s],R))),1)}
\]
\[
\& \ R
\]
\[
[2,1,3,3,1,4]
\]

Bindings used for result-list instantiation
Online Execution of the `serialise` FLP Specification: `serialise([d,a,l,l,a,s,t,e,x,a,s,u,s,a])`

**RELFUN Interface Page**
(http://www.dfki.uni-kl.de/~vega/relfun-cgi/cgi-bin/refun-cgi)

**Database:** PROLOG Syntax

- `t1() &: serialise([d,a,l,l,a,s]).`
- `t2() &: serialise([d,a,l,l,a,s,t,e,x,a,s,u,s,a]).`

- `serialise(L) :-
   numbered(qsort(before)(pairlists(L,R)),1) & R.`

- `pairlists([],[]) &: [].
pairlists([X|L],[Y|M]) &: tup([X,Y]|pairlists(L,M)).`

- `numbered([],N).
numbered([[X,N]|R],N) :- numbered(R,+(N,1)).`

- `qsort[Cr]([]): & [].
qsort[Cr](X|Y) :-
   partition[Cr](X,Y,Sm,Gr) &
   cat(qsort[Cr](Sm),tup(X|qsort[Cr](Gr))).`

- `partition[Cr](X,Y|Z,[Y|Sm],[Y|Gr]) :-
  Cr(Y,X), partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,Y|Z,[Y|Sm],[Y|Gr]) :-
  Cr(X,Y), partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[X|Z],Sm,Gr) :-
  partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[],[],[]).`

- `before([X1,Y1],[X2,Y2]) :- string<(X1,X2).`

- `cat([],L) &: L.
cat([H|R],L) &: tup(H|cat(R,L)).`

**Query (batch):**

- `t1()`
- `t2()`

**Result:**
relfun
rfi-p> t1()
[2,1,3,3,1,4]
rfi-p>
rfi-p> t2()
[2,1,4,4,1,5,6,3,8,1,5,7,5,1]

**Query (batch):**
trace pairlists numbered qsort[Cr]

**Copy & paste ready**

**Try again with tracer**
XML Serializations of the `serialise` FLP Specification: From (Relfun-Parsed) RFML/XML to RuleML/XML

**RFML**
(http://deliberation.ruleml.org/1.03/exa/Holog/nafhologeq/serialise.rfml):

...<ft>
  <pattop>
    <con>serialise</con>
    <var>l</var>
  </pattop>
  <callop>
    <con>numbered</con>
    <callop>
      <struc>
        <con>qsort</con>
        <con>before</con>
      </struc>
      <callop>
        <con>pairlists</con>
        <var>l</var>
        <var>r</var>
      </callop>
    </callop>
  </callop>
  <con>1</con>
</ft>
...

**RuleML**
(http://deliberation.ruleml.org/1.03/exa/Holog/nafhologeq/serialise.ruleml):

...<Implies>
  <Atom>
    <Rel>numbered</Rel>
    <Expr>
      <Expr per="value">
        <Fun per="copy">qsort</Fun>
        <Const>before</Const>
      </Expr>
      <Expr>
        <Fun per="value">pairlists</Fun>
        <Var>l</Var>
        <Var>r</Var>
      </Expr>
    </Expr>
  </Atom>
  <Data>1</Data>
</Implies>
...

**Translation:**

[http://ruleml.org/fun/1.0/](http://ruleml.org/fun/1.0/)

RFML ‘foot’ part becomes RuleML equation right-hand side in the ‘then’ part
Conclusion and Future Work

- RuleML can represent inputs, outputs, and background knowledge for Inductive Functional, Logic, and Functional-Logic Programming
- Precise requirements for these three declarative programming paradigms should lead to an initial characterization of specific Inductive Programming languages from the RuleML system, each
  - referring to an existing semantics (e.g., the Herbrand model semantics for Horn logic), or
  - adapting a semantics (e.g., Herbrand models with oriented equality or/and with object identifiers and slots)
- This enables the future specification of Inductive Programming RuleML languages in Relax NG (extending EBNF),
  - automated by MYNG for highly modular schema customization from desired language features