Functional RuleML:
From Horn Logic with Equality
to Lambda Calculus[*]

Harold Boley

Institute for Information Technology – e-Business,

National Research Council of Canada,

Fredericton, NB, E3B 9W4, Canada

Harold.Boley AT nrc-cnrc DOT gc DOT ca

October 15, 2005

Abstract

Functions are introduced to RuleML via orthogonal dimensions "constructor vs. user-defined", "single- vs. set-valued", "first- vs. higher-order". This enables functional-logic programming for the Semantic Web.

Keywords: RuleML, logic programming, functional programming, Horn logic with equality, interpretedness, valuedness, conditional equations, higher-order functions, lambda calculus

1 Introduction

Logic programming (LP) [GHR98] has been brought to the Semantic Web by RuleML,

whose relational rules are available as a modular system of XML Schema definitions [BBH+05]. Functional programming (FP) [BKPS03] is also playing an increasing Web role, with XSLT and XQuery [FRSV05] being prominent examples. We present here the design of Functional RuleML, developed via orthogonal notions and freely combinable with the previous Relational RuleML, including OO RuleML [Bol03]. This will also allow for FP/LP-integrated programming (FLP), including OO FLP, on the Semantic Web. Some background on FLP markup languages was given in [Bol00].

Since its beginning in 2000, with RFML [http://www.relfun.org/rfml] as one of its inputs, RuleML has permitted the markup of oriented (or directed) equations for defining the value(s) of a function applied to arguments, optionally conditional on a body as in Horn rules. Later, this was extended to logics with symmetric (or undirected) equality for the various sublanguages of RuleML, but the `Equal` element has still often exploited the left-to-right orientation of its (abridged) textual syntax.

It has been a RuleML issue that the constructor (`Ctor`) of a complex term (`Cterm`) is disjoined, as an XML element, from the user-defined function (`Fun`) of a call expression (`Nano`), although these can be unified by proceeding to a logic with equality. For example, while currently call patterns can contain `Cterm`s but not `Nano`s, obeying the "constructor discipline" [O'D85], the latter should also be permitted to legalize 'optimization' rules like `reverse(reverse(?L)) = ?L`.

This paper thus conceives both `Cterm`s and `Nano`s as expression (`<Expr>`) elements and distinguishes 'uninterpreted' (constructor) vs. 'interpreted' (user-defined) functions just via an XML attribute; another attribute likewise distinguishes the (single- vs. set-)valuedness of functions (section 2). We then proceed to the nesting of all of these (section 3). Next, for defining (interpreted) functions, unconditional (oriented) equations are introduced (section 4). These are then extended to conditional equations,

i.e. Horn logic implications with an equation as the head and possible equations in the body (section 5). Higher-order functions are finally added, both named ones such as `Compose` and λ-defined ones (section 6).

2 Interpretedness And Valuedness

The different notions of 'function' in LP and FP have been a continuing design issue:

LP: *Uninterpreted functions* **denote** unspecified values when applied to arguments, not using function definitions.

FP: *Interpreted functions* **compute** specified returned values when applied to arguments, using function definitions.

Uninterpreted function are also called 'constructors' since the values denoted by their application to arguments will be regarded as the syntactic data structure of these applications themselves.

For example, the function `first-born`: $Man \times Woman \rightarrow Human$ can be un-interpreted, so that `first-born(John, Mary)` just denotes the first-born child; or, interpreted, e.g. using definition `first-born(John, Mary) = Jory`, so the application returns `Jory`.

The distinction of uninterpreted vs. interpreted functions in RuleML 0.89 is marked up using different elements, `<Ctor>` vs. `<Fun>`. Proceeding to the increased generality of logic with equality (cf. section 1), this should be changed to a single element name, `<Fun>`, with different attribute values, `<Fun in="no">` vs. `<Fun in="yes">`, respectively: The use of a **Fun**ction's **in**terpreted attribute with values `"no"` vs. `"yes"` directly reflects uninterpreted vs. interpreted functions (those for which, **in** the rulebase, **no** definitions are expected vs. those for which they are). Functions' respective RuleML 0.89 [http://www.ruleml.org/0.89] applications with `Cterm` vs. `Nano` can then uniformly become **Expr**essions for either interpretedness.

The two versions of the example can thus be marked up as follows (where `"u"` stands

for "no" or "yes"):

```
<Expr>
  <Fun in="u">first-born</Fun>
  <Ind>John</Ind>
  <Ind>Mary</Ind>
</Expr>
```

In RuleML 0.89 as well as in RFML and its human-oriented Relfun syntax [Bol99] this distinction is made on the level of expressions, the latter using square brackets vs. round parentheses for applications. Making the distinction through an attribute in the `<Fun>` rather than `<Expr>` element will permit higher-order functions (cf. section 6) to return, and use as arguments, functions that include interpretedness markup.

A third value, "semi", is proposed for the **in**terpreted attribute: *Semi-interpreted functions* **compute** an application if a definition exists and **denote** unspecified values else (via the syntactic data structure of the application, which we now write with Relfun-like square brackets). For example, when "u" stands here for "semi", the

above application returns `Jory` if definition `first-born(John, Mary)` = `Jory` exists and denotes `first-born[John, Mary]` itself if no definition exists for it. Because of its neutrality, `in="semi"` is proposed as the default value.

In both XML and UML processing, functions (like relations in LP) are often *set-valued* (*non-deterministic*). This is accommodated by introducing a **val**ued attribute with values including "1" (deterministic: exactly one) and "0.." (set-valued: zero or more). Our `val` specifications can be viewed as transferring to functions, and generalizing, the cardinality restrictions for (binary) properties (i.e., unary functions) in description logic and the determinism declarations for (moded) relations in Mercury [SHC96].

For example, the set-valued function `children`: $Man \times Woman \rightarrow 2^{Human}$ can be **in**terpreted and set-**val**ued, using definition `children(John, Mary)` = {`Jory, Mahn`}, so that the application `children(John, Mary)` returns {`Jory`,

Mahn}.

The example is then marked up thus (other legal `val` values here would be `"0..3"`, `"1..2"`, and `"2"`):

```
<Expr>
  <Fun in="yes"
       val="0..">children</Fun>
  <Ind>John</Ind>
  <Ind>Mary</Ind>
</Expr>
```

Because of its highest generality, `val="0.."` is proposed as the default.

While uninterpreted functions usually correspond to `<Fun in="no" val="1">`, attribute combinations of `in="no"` with a `val` unequal to `"1"` will be useful when uninterpreted functions are later to be refined into interpreted set-valued functions (which along the way can lead to semi-interpreted ones).

Interpretedness and valuedness constitute orthogonal dimensions in our design space, and are also orthogonal to the dimensions of the subsequent sections, although space limitations prevent the discussion of all of their combinations in this paper.

3 Nestings

One of the advantages of interpreted functions as compared to relations is that the returned values of their applications permit nestings, avoiding flat relational conjunctions with shared logic variables.

For example, the function `age` can be defined for `Jory` as `age(Jory) = 12`, so the nesting `age(first-born(John, Mary))`, using the `first-born` definition of section 2, gives `age(Jory)`, then returns 12.

Alternatively, the function `age` can be defined for the uninterpreted `first-born` application as `age(first-born[John, Mary]) = 12`, so the nesting `age(first-born[John, Mary])` immediately returns 12.

Conversely, the function `age` can be left uninterpreted over the returned value of the `first-born` application, so the nesting `age[first-born(John, Mary)]` denotes `age[Jory]`.

Finally, both the functions `age` and

`first-born` can be left uninterpreted, so the nesting `age[first-born[John, Mary]]` just denotes itself.

The four versions of the example can now be marked up thus (where `"u"` and `"v"` can independently assume `"no"` or `"yes"`):

```
<Expr>
  <Fun in="u">age</Fun>
  <Expr>
    <Fun in="v">first-born</Fun>
    <Ind>John</Ind>
    <Ind>Mary</Ind>
  </Expr>
</Expr>
```

Nestings are permitted for set-valued functions, where an (interpreted or uninterpreted) outer function is automatically mapped over all elements of a set returned by an inner (interpreted) function.

For example, the element-valued function `age` can be extended for `Mahn` with `age(Mahn)` = 9, and nested, interpreted, over the set-valued interpreted function `children` of section 2: `age(children(John, Mary))` via `age({Jory, Mahn})` returns {12, 9}.

Similarly, `age` can be nested uninterpreted over the interpreted `children`: `age[children(John, Mary)]` via `age[{Jory, Mahn}]` returns {`age[Jory]`, `age[Mahn]`}.

The examples can be marked up thus (only `"u"` is left open for `"no"` or `"yes"`):

```
<Expr>
  <Fun in="u">age</Fun>
  <Expr>
    <Fun in="yes"
         val="0..">children</Fun>
    <Ind>John</Ind>
    <Ind>Mary</Ind>
  </Expr>
</Expr>
```

4 Unconditional Equations

In sections 2 and 3 we have employed expression-defining equations without giving their actual markup. Let us consider these in more detail here, starting with *un-*

conditional equations.

For this, we introduce a modified RuleML 0.89 `<Equal>` element, permitting both symmetric (or undirected) and oriented (or directed) equations via an `oriented` attribute with respective `"no"` and `"yes"` values. Since it is more general, `oriented="no"` is proposed as the default.

Because of the potential orientedness of equations, the RuleML 0.89 `<side>` role tag within the `<Equal>` type tag will be refined into `<lhs>` and `<rhs>` for an equation's **l**eft-**h**and **s**ide and **r**ight-**h**and **s**ide, respectively.

For example, the section 2 equation `first-born(John, Mary) = Jory` can now be marked up thus:

```
<Equal oriented="yes">
  <lhs>
    <Expr>
      <Fun in="yes">first-born</Fun>
      <Ind>John</Ind>
      <Ind>Mary</Ind>
    </Expr>
  </lhs>
```

```
  <rhs>
    <Ind>Jory</Ind>
  </rhs>
</Equal>
```

While the explicit `<lhs>` and `<rhs>` role tags emphasize the orientation, and are used as RDF properties when mapping this markup to RDF graphs, they can be omitted via stripe skipping [http://esw.w3.org/topic/StripeSkipping]: the `<lhs>` and `<rhs>` roles of `<Equal>`'s respective first and second subelements can still be uniquely recognized.

This, then, is the stripe-skipped example:

```
<Equal oriented="yes">
  <Expr>
    <Fun in="yes">first-born</Fun>
    <Ind>John</Ind>
    <Ind>Mary</Ind>
  </Expr>
  <Ind>Jory</Ind>
</Equal>
```

Equations can also have nested left-hand sides, where often the following restrictions apply: The `<Expr>` directly in the left-hand

side must use an interpreted function. Any `<Expr>` nested into it must use an uninterpreted function to fulfill the so-called "constructor discipline" [O'D85]; same for deeper nesting levels. If we want to obey it, we use `in="no"` within these nestings. An equation's right-hand side `<Expr>` can use uninterpreted or interpreted functions on any level of nesting, anyway.

For example, employing binary `subtract` and nullary `this-year` functions, the equation `age(first-born[John, Mary]) = subtract(this-year(),1993)` leads to this stripe-skipped 'disciplined' markup:

```
<Equal oriented="yes">
  <Expr>
    <Fun in="yes">age</Fun>
    <Expr>
      <Fun in="no">first-born</Fun>
      <Ind>John</Ind>
      <Ind>Mary</Ind>
    </Expr>
  </Expr>
  <Expr>
    <Fun in="yes">subtract</Fun>
```

```
    <Expr>
      <Fun in="yes">this-year</Fun>
    </Expr>
    <Data>1993</Data>
  </Expr>
</Equal>
```

5   Conditional Equations

Let us now proceed to oriented *conditional equations*, which use a (defining, oriented) `<Equal>` element as the conclusion of an `<Implies>` element, whose condition may employ other (testing, symmetric) equations. An equational condition may also bind auxiliary variables. While condition and conclusion can be marked up with explicit `<body>` and `<head>` roles, respectively, also allowing the conclusion as the first subelement, we will use a stripe-skipped markup where the condition must be the first subelement.

For example, using a unary `birth-year` function in the condition, and two ("?"-prefixed) variables, the conditional equation (written with a top-level "⇒")

?B = birth-year(?P) $\implies$

age(?P) = subtract(this-year(),?B)

employs an equational condition to test whether the `birth-year` of a person ?P is known, assigning it to ?B for use within the conclusion. This leads to the following stripe-skipped markup:

```
<Implies>
  <Equal oriented="no">
    <Var>B</Var>
    <Expr>
      <Fun in="yes">birth-year</Fun>
      <Var>P</Var>
    </Expr>
  </Equal>
  <Equal oriented="yes">
    <Expr>
      <Fun in="yes">age</Fun>
      <Var>P</Var>
    </Expr>
    <Expr>
      <Fun in="yes">subtract</Fun>
      <Expr>
        <Fun in="yes">this-year</Fun>
      </Expr>
```

```
      <Var>B</Var>
    </Expr>
  </Equal>
</Implies>
```

Within conditional equations, relational conditions can be used besides equational ones.

For example, using a binary `lessThanOrEqual` relation in the condition, the conditional equation

lessThanOrEqual(age(?P),15) $\implies$

discount(?P,?F) = 30

with a free variable ?F (flight) and a data constant 30 (percent), gives this markup:

```
<Implies>
  <Atom>
    <Rel>lessThanOrEqual</Rel>
    <Expr>
      <Fun in="yes">age</Fun>
      <Var>P</Var>
    </Expr>
    <Data>15</Data>
  </Atom>
  <Equal oriented="yes">
    <Expr>
```

```
    <Fun in="yes">discount</Fun>
    <Var>P</Var>
    <Var>F</Var>
  </Expr>
  <Data>30</Data>
  </Equal>
</Implies>
```

Notice the following interleaving of FP and LP (as characteristic for FLP): The function `discount` is defined using the relation `lessThanOrEqual` in the condition. The `<Atom>` element for the `lessThanOrEqual` relation itself contains a nested `<Expr>` element for the `age` function.

For conditional equations of Horn logic with equality in general [Pad88], the condition is a conjunction of `<Atom>` and `<Equal>` elements, as shown in appendix A.

6 Higher-Order Functions

Higher-order functions are characteristic for FP and thus should be supported by Functional RuleML. A *higher-order function* permits functions to be passed to it as (actual) parameters and to be returned from it as values.

Perhaps the most well-known higher-order function is `Compose`, taking two functions as parameters and returning as its value a function performing their sequential composition.

For example, the composition of the `age` and `first-born` functions of section 2 is performed by `Compose(age,first-born)`. Here is the markup for the interpreted and uninterpreted use of both of the parameter functions (where we use the default `in="semi"` for the higher-order function and let `"u"` and `"v"` independently assume `"no"` or `"yes"` for the first-order functions):

```
<Expr>
  <Fun>Compose</Fun>
  <Fun in="u">age</Fun>
  <Fun in="v">first-born</Fun>
</Expr>
```

The application of a parameterized `Compose` expression to arguments is equivalent to the nested application of its param-

eter functions.

For example, when interpreted with the definitions of section 2, `Compose(age,first-born)(John, Mary)` via age(first-born(John, Mary)) returns 12.

All four versions of this sample application can be marked up thus (with the usual `"u"` and `"v"`):

```
<Expr>
  <Expr>
    <Fun>Compose</Fun>
    <Fun in="u">age</Fun>
    <Fun in="v">first-born</Fun>
  </Expr>
  <Ind>John</Ind>
  <Ind>Mary</Ind>
</Expr>
```

Besides being applied in this way, a `Compose` expression can also be used as a parameter or returned value of another higher-order function.

To allow the general construction of anonymous functions, `Lambda` formulas from λ-calculus [Bar97] are introduced. A λ-*formula* quantifies variables that oc-cur free in a functional expression much like a ∀-*formula* does for a relational atom. So we can extend principles developed for explicit-quantifier markup in FOL RuleML [http://www.w3.org/Submission/FOL-RuleML], where quantifiers are allowed on all levels of rulebase elements.

For example, the function returned by `Compose(age,first-born)` can now be explicitly given as λ(`?X, ?Y`)age(first-born(`?X, ?Y`)). Here is the markup for its interpreted and uninterpreted use (with the usual `"u"` and `"v"`):

```
<Lambda>
  <Var>X</Var>
  <Var>Y</Var>
  <Expr>
    <Fun in="u">age</Fun>
    <Expr>
      <Fun in="v">first-born</Fun>
      <Var>X</Var>
      <Var>Y</Var>
    </Expr>
  </Expr>
</Expr>
```

11

```
</Lambda>
```

This `Lambda` formula can be applied as the `Compose` expression was above. The advantage of `Lambda` formulas is that they allow the direct λ-*abstraction* of arbitrary expressions, not just for (sequential or parallel) composition etc. An example is λ(?X, ?Y)plex(age(?X), xy, age(?Y), fxy, age(first-born(?X, ?Y))), whose markup should be obvious if we note that `plex` is the interpreted analog to RuleML's uninterpreted built-in function for n-ary com**plex**-term (e.g., tuple) construction.

By also abstracting the parameter functions, `age` and `first-born`, `Compose` can be defined generally via a `Lambda` formula as Compose(?F, ?G) = λ(?X, ?Y) ?F(?G(?X, ?Y)). Its markup can distinguish object (first-order) **Var**iables like ?X vs. function (higher-order) ones like ?F via attribute values ord="1" vs. ord="h".

7 Conclusions

The design of Functional RuleML as presented in this paper also benefits other sub-languages of RuleML, e.g. because of the more 'logical' complex terms. However, there are some open issues, two of which will be discussed below.

Certain constraints on the values of our attributes can apparently not be enforced with DTDs (cf. appendix A) or even (current) XSDs, e.g. in="no" on functions in call patterns in case we wanted to always enforce the constructor discipline (cf. section 4). However, a semantics-oriented validation tool will be required for future attributes anyway, e.g. for testing whether a rulebase is stratified. Thus we propose that such a static-analysis tool should be developed to make these fine-grained distinctions for all 'semantic' attributes.

The proposed defaults for some of our attributes may require further revisions. It might be argued that the default in="semi" for functions is a problem since equations could be invoked inadvertently for functions that are applied without an explicit in attribute. However, notice that the default oriented="no" for equations

permits to 'revert' any function call, using the same equation in both directions. Together, those defaults thus constitute a kind of 'vanilla' logic with equality, which can (only) be changed via our explicit attribute values.

While our logical design does not specify any evaluation strategy for nested expressions, we have preferred 'call-by-value' in implementations [Bol00]. A reference interpreter for Functional RuleML is planned as an extension of OO jDREW [BBH+05]; the first step has been taken by implementing oriented ground equality via an `EqualTable` data structure for equivalence classes [http://www.w3.org/2004/12/rules-ws/paper/49].

References

[Bar97] Henk Barendregt. The Impact of the Lambda Calculus in Logic and Computer Science. *The Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[BBH+05] Marcel Ball, Harold Boley, David Hirtle, Jing Mei, and Bruce Spencer. The OO jDREW Reference Implementation of RuleML. In *Proc. Rules and Rule Markup Languages for the Semantic Web (RuleML-2005)*. LNCS 3791, Springer-Verlag, November 2005.

[BKPS03] Paul A. Bailes, Colin J. M. Kemp, Ian Peake, and Sean Seefried. Why Functional Programming Really Matters. In *Applied Informatics*, pages 919–926, 2003.

[Bol99] Harold Boley. Functional-Logic Integration via Minimal Reciprocal Extensions. *Theoretical Computer Science*, 212:77–99, 1999.

[Bol00] Harold Boley. Markup Languages for Functional-Logic Programming. In *9th International Workshop on Functional*

and *Logic Programming, Benicassim, Spain*, pages 391–403. UPV University Press, Valencia, publication 2000/2039, September 2000.

[Bol03]   Harold Boley. Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms. In *Proc. Rules and Rule Markup Languages for the Semantic Web (RuleML-2003)*. LNCS 2876, Springer-Verlag, October 2003.

[FRSV05] Achille Fokoue, Kristoffer Rose, Jérôme Siméon, and Lionel Villard. Compiling XSLT 2.0 into XQuery 1.0. In *Proceedings of the Fourteenth International World Wide Web Conference*, pages 682–691, Chiba, Japan, May 2005. ACM Press.

[GHR98]   Dov Gabbay, Christopher Hogger, and J. A. Robinson, editors. *Handbook of Logic in Artificial*

Intelligence and Logic Programming, Volume 5: Logic Programming. Oxford University Press, Oxford, 1998.

[O'D85]   M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, Cambridge, Mass., 1985.

[Pad88]   P. Padawitz. *Computing in Horn Clause Theories*. EATCS Monographs on Theoretical Computer Science, Vol. 16. Springer, 1988.

[SHC96]   Z. Somogy, F. Henderson, and T. Conway. The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

A DTD for Functional RuleML

A DTD for our stripe-skipped version of Functional RuleML is given below. It mainly consists of declarations specifying a `Rulebase` (i.e., the **Assert**ion of an `And`) with zero or more `Implies`/`Atom`/`Equal` clauses. We introduce here for **Rel**ations **in**terpretedness distinctions analogous to those for **Fun**ctions, where the novel `<Rel in="no">` accommodates embedded propositons of model logics. An **Expr**ession, say `f[i]`, with an uninterpreted function, here `f`, can itself be used as the uninterpreted or interpreted function of another expression, e.g. `f[i][a]` or `f[i](a)`; to specify this distinction, such a 'function-naming' **Expr**ession also needs an **in**terpreted attribute. For DTD-technical reasons, only the two most important values are specified for the `val` attribute (similarly, only two `ord` values are given). The DTD also does not enforce context-dependent attribute values such as `<Equal oriented="no">` being normally used in conditions. Moreover, while the DTD does not prevent `Lambda` formulas to occur on the lhs of (both kinds of) equations, a static analyzer should confine them to the rhs of oriented equations. A more precise XSD is part of the emerging Functional RuleML 0.9 [http://www.ruleml.org/fun].

```
<!ENTITY % term          "(Data | Ind | Var | Expr)" >
<!ENTITY % ateq          "(Atom | Equal)" >
<!ENTITY % conclusion    "(%ateq;)" >
<!ENTITY % condition     "(And | %ateq;)" >


<!ELEMENT Rulebase        (Implies | %ateq;)* >


<!ELEMENT Implies        (%condition;, %conclusion;) >


<!ELEMENT And            (%ateq;)* >
```

```
<!ELEMENT Equal         (%term;, %term;) >

<!ELEMENT Atom          ((Rel | Expr | Lambda),
                         (%term; | Rel | Fun | Lambda)*) >

<!ELEMENT Expr          ((Fun | Expr | Lambda),
                         (%term; | Rel | Fun | Lambda)*) >

<!ELEMENT Lambda        ((%term;)+, %term;) >

<!ELEMENT Fun           (#PCDATA) >
<!ELEMENT Rel           (#PCDATA) >
<!ELEMENT Data          (#PCDATA) >
<!ELEMENT Ind           (#PCDATA) >
<!ELEMENT Var           (#PCDATA) >

<!ATTLIST Equal oriented (yes | no) "no" >
<!ATTLIST Expr in       (yes | no | semi) "semi" >
<!ATTLIST Rel in        (yes | no | semi) "semi" >
<!ATTLIST Fun in        (yes | no | semi) "semi" >
<!ATTLIST Fun val       (1 | 0..) "0.." >
<!ATTLIST Var ord       (1 | h) "h" >
```