4QL Runner - Developer Description*

Łukasz Sobczyk

October 20, 2012

Introduction

4QL Runner is a reimplementation of Inter4QL [3] in Java, based on Drools rule system [8]. It has been described in the M.Sc. Thesis [2]. This paper, based on [2], describes some details of 4QL Runner useful for developers who want to use or extend 4QL Runner.

The paper is structured as follows. Section 1 lists tools used for developing the interpreter. Section 2 contains a brief description of 4QL language and discusses differences between 4QL Runner and Inter4QL. Section 3 describes the interpreter's architecture and contains some remarks on adjusting it to suit limited environments. Section 4 describes the interpreter's code and contains a brief introduction to technologies used (Drools and ANTLR). Section 5 contains remarks on 4QL Runners efficiency. Section 6 concludes with a list of ideas for future development of 4QL based technology.

1 Libraries Used

The code is based on ANTLR parser and Drools rule engine:

- antlr-3.4-complete [6]
- Drools-distribution-5.4.0.Final [8].

It was developed using Eclipse IDE [7] with the following plugins:

- Drools Eclipse IDE plugin 5.4.0.Final [8]
- Graphical Editing Framework GEF 3.7.2. [required by Drools plugin]
- ANTLR antlride-2.1.2 [6]
- Window builder 1.5.0 [9] [for GUI development].

*Supported by the Polish National Science Centre grant 2011/01/B/ST6/02769.

2 4QL

4QL, introduced in [1], is a rule language based on a four-valued logic. In principle, reasoning in 4QL is monotonic, but the mechanism of modules together with external literals allows one to express a variety of nonmonotonic rules.

The logic of 4QL consists of four values: *true, false, unknown* and *inconsistent*. Intuitions behind those values are the following:

- *unknown* describes facts for which no information is available; facts are considered unknown until information about them is provided
- *true* facts are facts stated to be true (and not stated to be false or inconsistent)
- *false* facts are facts stated to be false (and not stated to be false or inconsistent)
- *inconsistent* facts are facts stated to be both true and false at the same time.[1]

Modules group facts and rules. Modules can refer to facts from other module, but dependencies need to be acyclic. This creates layered structure. When querying data from other module it is possible to detect inconsistency or the lack of information and resolve them. This creates a possibility of reasoning characteristic for nonmonotonic systems, while keeping the base logic monotonic, allowing unrestricted negation and keeping PTIME data complexity.

Rules in 4QL have a strict syntax of implication where the conclusion is a (possibly negated) fact, and the condition is a disjunction of conjunctions of (possibly negated) facts. It can be expressed in the form used by interpreter:

```
headFact() :- cond1(),-cond2(),cond3() |
              cond1'(),cond2'() | cond1"().
```

The meaning of the above rule is:

$$\big((cond1() \wedge \sim cond2() \wedge cond3()) \vee (cond1'() \wedge cond2'()) \vee cond1''()\big) \rightarrow headFact()$$

Facts can be written as:

```
knownFact().
```

this translates to the rule:

$$[true] \rightarrow knownFact()$$

There can be only one rule having given fact as its head. (Negated fact can have another rule.)

[1] Because of monotonicity assumed inside od each module, no additional information can resolve this contradiction. Resolving inconsistencies requires a use of external literals.

Facts are elements of predefined relations. Variables can be used to create more general rules. References to facts from other modules are made by external literals consisting of a module name and a relation name, separated by dot. When referring to a fact from a different module it is possible to query about the specific value of a fact.This requires external literals with

"in {logic value set}"

syntax and behaves a bit different from normal fact reference (it can only be true or false and can detect inconsistency or unknown information). One can only use this statement when referring to other modules (and those references need to be acyclic). For example,

```
isSad(X) :- tookExam(X), -passedExam(X).
sensorError() :- sensorInput.montionSensor(freeRoad) in {incons}.
```

4QL Runner as its input takes module definition files describing type aliases, relation definitions, rules and facts:

```
module moduleName:
  domains:
    literal literalTypeAlias.
    integer integerTypeAlias.
  relations:
    relationName1().
    relationName2(literalTypeAlias, logic).
    relationName3(string).
  rules:
    relationName1() :- relationName2(X,true),
                       relationName2(X,false).
  facts:
    relationName2(lit1,true).
    relationName2(lit1,true).
    relationName2(lit2,unknown).
    -relationName2(lit2,true).
    -relationName3("some string").
end.
```

Each section is optional. Supported types are integer, real, literal, string, logic, date and dateTime. Literals are alphanumerics beginning with a lowercase letter (relation and module names should be literals). If an alphanumeric begins with a capital letter, it is considered to be a variable name. Logic is a four valued logic value (true, false, unknown, incons). DateTime format is year-month-day hour:minute [YYYY-MM-DD HH:mm]. Date is like DateTime but without the HH:mm part.

4QL Runner can also run in a "runRules" mode that assumes all statements are rules or facts and infers types, but this mode is for simple testing purposes only. In this case, the verification is very limited and dependencies are not

analyzed. This may be a cause of incorrect results (due to wrong computation order) or even of infinite loops.

The meaning of those rules is defined by calculating the (unique) well supported model of this set of rules (facts are considered rules with the empty body). The rule value is defined by following semantics:

$\wedge$	t	i	u	f
t	t	i	u	f
i	i	i	u	f
u	u	u	u	f
f	f	f	f	f

$\vee$	t	i	u	f
t	t	t	t	t
i	t	i	i	i
u	t	i	u	u
f	t	i	u	f

$\sim$	
i	i
t	f
f	t
u	u

$\rightarrow$	i	t	f	u
i	t	f	f	f
t	t	t	f	f
f	t	t	t	t
u	t	t	t	t

A model is a valuation of facts such that all rules take the value true. In particular, the valuation where all facts are valued inconsistent is a model for any set of rules. Conjunction of inconsistent facts is inconsistent, disjunction as well, and inconsistent implies inconsistent is valued true. It is required that models are well supported and contain as little inconsistency as possible.

Note that the semantics of implication can be understood as: inconsistent conditions result in inconsistent conclusions. True conditions imply true conclusions, but some other reasoning might still make them inconsistent.

In order to calculate this model efficiently a 4QL algorithm was developed [1]. It consists of three phases.

- Phase 1: Finding basic inconsistencies.

  Fact and its negation are considered separate literals in classic logic. A Herbrand model for the given rule set is computed. Facts that have both true and false literals in Herbrand model are considered inconsistent.

- Phase 2: Finding possibly true literals.

  Phase 1 is repeated, but without rules that have facts found in Phase 1 in its head. This makes sure all facts found in Phase 1 are not used in the reasoning process. The resulting Herbrand model defines fact values: if a literal is in the Herbrand model, its valuation is true,[2] otherwise its valuation is unknown. Note that no fact can be both true and false, as such literals have been identified in Phase 1 and are not used here.

[2]Observe that if a literal `-r()` is in the Herbrand model then it is assigned true, so `r()` is assigned false.

- Phase 3: Reasoning without inconsistency.

  The valuation resulting from Phase 2 might still not be a model for the given rule set. In Phase 3 rules that have inconsistent body, but true conclusions are found and valuation of conclusions are set to inconsistent. This may cause next rule bodies to become inconsistent. This process stops when all rules become true according to 4QL semantics. In the worst case, all facts will become inconsistent. The result of this phase is a valuation that is the unique well supported model for given set of rules.

The above procedure is performed for every module. Cross module references from the point of view of this computation are equivalent to constant values. The interpreter just needs to compute modules in order specified by their dependencies.

For more information see [1] and other papers available via [5].

2.1 4QL Runner and Inter4QL - Domain Scope Differences

When using rules with statements like:

1. $\bar{a}$ "in {unknown}"
2. magic module relation

with variables in $\bar{a}$ not referring to stored facts, 4QL Runner will raise an error, while Inter4QL will accept such rules. For example, in:

```
//lower.known(literal).
unknown(X) :- lower.known(X) in {unknown}.
```

the domain of variable `X` is the whole domain of literals. Inter4QL limits domains to all constants of given type that appear in modules. 4QL Runner does not make such a decision and requires the user to specify relations that will limit the scope of variables. For example:

```
interesting(X) :- lower.iHeardAbout(X).
unknown(X) :- interesting(X),lower.known(X) in {unknown}.
```

Similar phenomenon happens for some magic module relations:

```
bigNum(X) :- math.gt(X,100).
```

It needs some base in stored facts to be valid rule in 4QL Runner:

```
isNumber(99).
//...
isNumber(150).
bigNum(X) :- isNumber(X), math.gt(X,100).
```

NOTE: The source of the problem is not the size of the set of facts. The problem is that Drools engine requires a fact from memory to be matched. Both "in unknown" and magic module statements are implemented by filtering matched facts. Rules like:

```
myRange(X) :- math.gt(X,100), math.lt(X,150).
```

(with `X` being an integer) specify a finite set, but will still not be valid in 4QL Runner as `X` doesn't refer to any fact stored in database.

3 Architecture of 4QL Runner

The 4QL Runner's main task is to create a Drools session that corresponds to a given 4QL modules file. The process consists of a few steps:

Module File – [Parser] –>
fourQLmoduleDef collection – [verification] [DRLgen] –>
DRL file – [Drools compiler] –>
KnowledgeBase – [newStatefulKnowledgeSession()] –>
Session

This whole task is encapsulated in the RuleFileBrain class that is initialized by LoadRuleFile(filename) method and performs the above actions. RuleFileBrain has an interface that hides the Drools part, enables manipulation of stated facts and makes it possible to ask about brain's knowledge.

3.1 Running in Limited Environments

If the RuleFileBrain class was to be used in a limited environment, such as an embedded system or an agent system fragment, it might be beneficial to split the rule compilation and the rule execution parts. It this case ANTLR and Drools compiler would not be needed on the machine executing rules. Drools documentation mentions that the minimal required Drools runtime should not exceed a few hundred KB. In order to achieve this size, LoadRuleFile needs to be moved to a separate class and RuleFileBrain should be initialized with its output. Other option would be to serialize RuleFileBrain (unimplemented in the current version) so that it stores KnowledgeBase, facts and moduleDefinitions, then sends it to the limited system and deserializes there.

4 Interpreter's Code

The code base is divided into four packages:

- core - three core classes: Relation, Rule and fourQLmoduleDef; note that the Relation class can be used not only to define relations, but also facts or simple fact-based expressions in rules

- parser - module responsible for parsing 4QL files
- brain - basic interface for four-valued communication
- droolsBrain - main classes, RuleFileBrain class that can use 4QL reasoning and communicate using an interface defined in the brain package; this module also contains classes that implement 4QL rules in Drools DRL rules.

4.1 Core Package

- **Relation**
  (initially was meant for use in parser only, but turned out to be good enough to be used throughout the program).
  - It can be used to define a relation or a fact belonging to a given relation; its name and module should identify the relation, i.e., there should not be two different relations with the same name and module in the system.
  - It defines constants TRUE, FALSE, UNKNOWN and INCONS, that should be used when referring to truth values.
  - The field *value* for each relation definition should be set to TRUE (default value for new objects), for facts it specifies the fact value. Note that UNKNOWN facts are usually not represented explicitly. All facts not represented in the system are considered UNKNOWN, but sometimes it is useful to refer to them, for example when some fact becomes UNKNOWN and an object wants to inform other objects about it.
  - The field *arguments* specifies relation arguments; for relation definition only argument types are important, argument value field is ignored and isVariable should be set to FALSE; for facts the argument type defines how to interpret string in the value field (it should be a legal value according to module file standard types) unless is variable field is set to TRUE, then value is variable identifier.
  - Fields *hasIn* + *inMask* are used for describing
    "fact() in {false, incons}"
    type statements; for relations and facts should be left at the default FALSE, inMask=0; those fields are only used for creating a Rule with "in" statement with Java code rather than the parser, when hasIn is set to TRUE. inMask is a bit mask of the required values, where the n'th bit corresponds to the logical value represented by n, i.e. inMask |= 1<<UNKNOWN; adds UNKNOWN to inMask; clearing mask can be done by setting it to 0, there is also a helper function that adds a value to inMask taking its string representation.

- **Rule**
  This class is a simple container class for rule head and rule body represented as nested ArrayLists, it provides only toString method for printing rule description.

- **fourQLmoduleDef**
  This class describes a module. It is usually obtained by parsing modules file (but can be created by code if needed). It can verify a set of modules returning the list of errors found.

  The important fields are: name, domainList, relationList, ruleList and factList containing information from the module description. All other fields are set using this data during verification process. DAGpriority describes order for modules (from the last verify call) in which module dependencies would refer only to modules with lower number, it is dependent on a given set of modules, rather than a property of a module, but is stored here for convenience (a module is not expected to be verified in different sets of modules).

4.2 Parser Package

The parser is based on ANTLR [6]. ANTLR framework requires a file describing a grammar in the Extended BNF form, possibly with code fragments inserted into the grammar. It generates recursive descent-like parser that parses grammar and runs the given code when parsing the grammar part containing the code. It is a LL(*) parser that is fast and relatively easy to modify. For example:

```
module returns [fourQLmoduleDef module]:
{module = new fourQLmoduleDef();}
'module' name=Literal ':'
{ module.name = name.getText();}
('domains'':'
domainList[module])?
('relations'':'
relationList[module])?
('rules'':'
ruleList {module.addAllRules($ruleList.rules);})?
('facts'':'
factList {module.addAllFacts($factList.facts);})?
'end.';
```

is a fragment of a grammar that parses a single module definition. While parsing, the corresponding fourQLmoduleDef object is created and filled with parsed data.

The following convention is used:

- 'text' ale simple lexems

- productionName are productions (defined in other part of the grammar file)
- productions starting with a capital letter are lexems (defined in other part of the grammar file)
- *?+| describe regexp operations, respectively Kleene star, optional, repetition (at least one time), alternative; the default is concatenation, grammar fragments can be grouped using parentheses: (someEBNFexp)
- lexems provide the getText() method returning text fragment interpreted as a given lexem
- productions can be given arguments and can return objects. For example domainList production takes a fourQLmoduleDef object and adds parsed domain definitions to this module.

This approach makes it really fast and easy to create simple parsers like 4QL module file parser.

Also note that all productions have a Java parsing function generated. Any production can be used on its own, e.g., to read fact descriptions form given InputStream.

For more information see [6].

4.3 Brain Package

The brain layer defines basic communication interface for objects based on the four-valued logic. This interface is meant to provide communication between different systems not necessarily based on 4QL. This enables the creation of hybrid systems. For example, it is possible to wrap non 4QL data sources like SQL database using container implementing Brain interface and use it for 4QL reasoning as a fact source.

There are two interfaces provided:

- a4QLbrainI for class providing four-valued fact data
- a4QLSubsctiberI for class that requests to be notified when facts change.

A brain interface can be used to translate fact information form any source that provides information, be it sensor, SQL database, 4QL database well-supported model or any other object. It only needs to be able to provide four valued facts and implement the interface.

A brain can be asked about any or all fact values it knows. Other classes can also subscribe on a given set of facts to be notified when their values change. Brain should keep this information relatively up to date, but details might differ. For example sensor might update information once a given time interval or when something really important happens, rather than at its maximum sampling frequency. For the ease of extending a form of general query specified by a string of characters is provided (along with another one defining format used in the

query). Brain provides information on supported formats and can reject queries it doesn't understand.

This is a proposal for integrating 4QL into a larger system. This interface will probably change to suit needs of programs using 4QL as 4QL technology develops, but the current version is sufficient to implement prototype applications using 4QL reasoning.

4.4 RuleFileBrain Package

Rule file brain is the main class for this project. It can:

- read a 4QL module description file
- create Drools session
- calculate well-supported model
- provide integration interface for Brain layer.

Initial facts are read from file, but can be easily changed using brain's methods. When facts change a new session is created and a new model is calculated, then subscribers are notified of changes to facts they subscribed to.

In order to create Drools session that calculates the well-supported model RuleFileBrain depends on DRLgen class.

4.5 DRLgen

This class is the core of 4QL Runner. It prepares Drools rule file (DRL) that will compute the (unique) well supported model for a given rule and facts sets.

4.5.1 Drools

Drools Expert is a hybrid reasoning system [8]. Drools started out as a production rule system that enables forward chaining with rules using a RETE based algorithm. According to Drools documentation in version 5.x, backward chaining mechanisms were introduced, but 4QL Runner uses the forward chaining part.

Drools has a fact memory that stores objects that will be processed and a set of rules that try to match objects form the fact memory and execute an action based on the objects matched. Rules are created by writing a DRL text file. Drools compiler reads the file and creates a knowledge base object. The knowledge base object can then create knowledge session objects that have their own fact memory and can fire rules processing objects from their fact memory. Compiling rules is expensive, while creating new sessions is relatively cheap. This two step process makes it easy to create many sessions for given rule set.

A rule is written as:

```
rule "ruleName"
 //some options might appear here
when
 // condition part, called LHS (left hand side)
 $dev : Device(state==Device.State.broken)
then
 // action part, called RHS (right hand side)
 insert( new RepairOrder($dev));
end.
```

The LHS part is a statement that will try to match certain facts from facts memory. In this case it will try to find Device class objects with field state equal to "Device.State.broken" value. LHS might match multiple objects or check some more complicated conditions including the lack of certain objects in fact memory and queries that aggregate information in all facts meeting certain criteria. It is important to note here that "or" statements are understood as a short notation for two rules with the same RHS part. LHS statements may contain the eval() statement with any Java boolean expression, so it is important to note that LHS should not modify objects in fact memory. For more information on LHS see the Drools documentation [8].

For every LHS match, the RHS action is executed. (If there is rule with "or" with both alternative parts met RHS will be executed two times.) RHS is usually a Java code but some other dialects are also supported. RHS can refer to objects matched in LHS. Matched objects identifiers often start with $ symbol for readability, but it is not required. RHS can and often will modify fact memory. A few Drools methods for this are insert(), insertLogical(), retract(), update(). This will often affect other rules' LHS matches, so the order of rules firing is important.

All rules with matching LHS form an agenda. An agenda is a set of rules that are possible to fire at a given time. From the agenda one rule is chosen at a time. Rule choosing policy might be specified by the Drools user, but one can assume nondeterministic rule choosing policy. Once rule is chosen and fired, its RHS possibly influences the resulting agenda. A rule will fire only once for a given LHS match, but inserting new facts might still trigger the same rule with a new object matched. If a Drools session is started with fireAllRules() method, it will fire rules until the agenda becomes empty. The other option is fireUntilHalt() that will fire until RHS executing halt() method is fired. It is possible to loop Drools execution, for example with following rule:

```
rule "loop"
when
 MyObj(N)
then
 insert(new MyObj(N+1));
end.
```

Analyzing rules' behavior is the key in designing good Drools-based system. Drools provides some mechanisms to control rule firing. Rules can specify priorities. Rules with higher priority will be chosen to fire form agenda first (in Drools the priority is called "salience"). There are also other ways to control rules execution like agenda groups, no-loop option for preventing some simple loops, etc..

Fats in the Fact memory may be any Java objects. However, the following restriction applies:

> such objects should be Java Beans with correct equals, getHash methods and set/get methods for accessing fields.

Drools can also specify simple class descriptions inside DRL file and generate valid Java classes from it.

A fast execution of matching process is provided by the RETEOO algorithm. It is a version of the RETE algorithm adjusted for Java objects. The basic principle is sorting objects to sets matching given conditions and then joining those groups to form valid LHS matches. Both sorting and joining has a tree structure that can effectively sort elements or perform fast joins with filtering.

Form more information on Drools see Drools Expert documentation [8].

4.5.2 DRL for 4QL

This section describes how 4QL algorithm was translated to work with Drools. It describes the DRL file created when using 4QL Runner.

First, classes for facts are created. For each number of arguments a new class is created. For N argument facts the class has the following fields:

- name - a string in the form: "module_name.relation_name" identifying relation the fact belongs to
- value - the logical value of a fact, possible values ale defined by constants Relation.TRUE, Relation.FALSE, Relation.INCONS (UNKNOWN facts are not represented by objects)
- prio - the priority of the module the fact comes from, influencing the order of calculation, important for cross-module references; priorities of modules are set during the verification process and stored in DAGorder field of fourQLmoduleDef class
- lv - the well-supported model computation follows the 4QL algorithm [1] and requires three phases, each having a different set of facts; this field specifies the set of facts it belongs to; in general:
  - lv=0 - stated facts
  - lv=1 - facts used in the first phase of 4QL algorithm - finding inconsistencies

- lv=2 - true/false facts, corresponds to phase 2 of the 4QL algorithm (will be modified in Phase 3 of algorithm)
- lv=3 - all inconsistent facts (from Phase 1 and Phase 3)
- argI - with $I \in \{1..N\}$, a string representing the value on the Nth fact argument.

The stated facts are inserted with lv=0 and prio dependent on DAGpriority of the module they come from.

Next, a few layers of rules fire to compute the well-supported model. Rule priorities make sure that the computation of a given rule layer is finished before the next layer rules can fire. Rule layers are presented here in the order they fire.

The first set of rules creates lv=1 facts form stated lv=0 facts. (INCONS facts are represented as the pair of facts: both TRUE and FALSE at lv=1)

Next, the least Herbrand model for 4QL rule set is calculated. For each 4QL rule a DRL rule is created, that tries to match lv=1 fact sets given by rule's clauses and, if successful, adds the rule head as a new lv=1 fact.

Next, inconsistencies are found on lv=1. For each lv=1 fact that has both TRUE and FALSE version in lv=1, a corresponding INCONS fact is added to lv=3 (inconsistent) fact set.

The next step finds facts that are potentially true or false. It starts with logically inserting to lv=2 each lv=0 (stated) fact, that does not appear in lv=3 set (lv=3 contains inconsistent facts, lv=2 does not contain inconsistent facts).

Next, 4QL rules similar to lv=1 rules are run on lv=2. The difference is that those rules check if the conclusion is already on lv=3. If so, they do not logically insert conclusion. This corresponds to Phase 2 of the original algorithm and creates the set of potentially true/false facts with no inconsistent ones.

Next the set of 4QL rules propagates inconsistencies. Some facts in lv=2 are based only on inconsistent information, so they will be also considered inconsistent as in the Phase 3 of the original algorithm [1].

This layer is less straightforward. Note that the body of a rule is inconsistent iff:

1. none of its clauses is true (because it would make body true), and
2. there exists a clause with only true or inconsistent facts (it can't be all true since 1. holds, so it is inconsistent).

This makes those rules quite efficient for Drools even with many clauses. Rules only check "is conclusion not on lv=2" (equivalent to 1) and "is there any clause with facts matching both lv=(2 or 3) and val=(TRUE or FALSE or INCONS)", this kind of match criteria is efficient in Drools as this alternative is on a level of fact filtering rather than creating many LHS for given RHS.

Because all facts at lv>0 are added using insertLogical, this set of rules automatically retracts facts from lv=2 during lv=3 inconsistency propagation. When new fact is inserted on lv=3, the corresponding lv=2 rule condition stops being true and the fact is automatically retracted.

(This step agrees with the 4QL semantics, because of TMS interaction regarding cyclic dependencies described later.)

Each module generates such set of rules, that differ only on the priority of facts they process, those rules are fired in the order specified by DAGpriority fields of those modules. This makes sure that cross module references are calculated correctly.

After these computations, Relation objects are created according to facts on lv=2 and lv=3. There is also a query that extracts those relations from Drools for use in Java code.

4.6 Logical Cycles

In the current Drools implementation TMS (truth maintenance system) uses a simple reference count. This creates problems when rules with logical insert have cyclic dependencies. For example, consider:

```
when A() then logicalInsert(new B()) end.
when B() then logicalInsert(new C()) end.
when C() then logicalInsert(new B()) end.
```

When object `A` is inserted, it inserts object `B`, object `B` inserts object `C`, and `C` inserts `B`, but `B` is already in the fact memory, so its ref cnt is set to 2.

Now when `A` is removed `B`'s refcnt is reduced to 1, so `B` and `C` stay in the memory, even though no stated fact supports rules that inserted them.

Originally, logicalInserts were meant to allow dynamic updates of 4QL facts. Adding or deleting facts should require recalculation of only some part of the rule set, all other facts should be left untouched. Unfortunately no simple way of deleting such cycles with the current TMS semantics was found, so ruleFileBrain recalculates the whole model after each fact change. (The calculation is relatively fast, so it is not a big problem now, but it might become a problem with large enough fact sets; the best way to do it currently is to split computation to many RuleFileBrains and communicate them using subscriptions, if they are independent enough it will limit the scale of recomputation). A better way to deal with this issue in Drools might be found in the future.

When one analyzes the 4QL semantics whenever many rules (including facts) can have the same head, a bit similar phenomenon occurs.

For example, consider a set of rules:

```
b:-a;
c:-b | d;
d:-c;
```

with facts:

```
a = incons
b = true
```

during Phase 1 `a` is found inconsistent
during Phase 2 `b,c,d` are found true
during Phase 3 `b` is found inconsistent (but `c` and `d` are not inconsistent).

NOTE: the pure 4QL semantics does not allow multiple rules with the same head, and in the pure 4QL a fact is defined as a rule "fact() :- true.", so this describes a 4QL version used in interpreters where facts and rules are separate and fact declarations are source of facts separate from rules.

In Inter4QL and 4QL Runner `c` and `d` are true, even though there is no stated fact supporting them, they support each other enough for Phase 3 of inconsistency propagation not to consider them inconsistent. If the Drools TMS starts removing such cycles in its future versions, DRL rules will need to be changed in order to support such lv=2 subsets of facts that support each other, but "lost their ground" due to Phase 3 of inconsistency propagation.[3]

4.7 Magic Modules

Some purely rule-based calculations would be inefficient or really inconvenient. For example, a comparison of numbers is much more efficient when the standard computer arithmetics is used. Because Drools can evaluate any Boolean expression in Java as a part of LHS (condition part) in a rule, such possibility can be added to 4QL Runner. DRLgen makes it possible by the *magic module* mechanism.

To add a magic module one needs to write a class that implements MagicModule interface. It consists of three functions:

- getModuleName - the name of your magic module
- verifyCall - the function called by fourQLmoduleDef during the verification of a relation, returns true/false
- genCall - the function returning a LHS DRL rule fragment corresponding to the given relation (or throws an exception if provided Relation is not valid).

Next, one needs to register the new magic module in MagicModuleSet singleton used by fourQLmoduleDef.verify method and DRLgen class.

It is important to note that normal modules take precedence over magic modules - a magic module will be looked for after the verifier fails to find normal relation of given name. During the DRL generation, magic module LHS will be generated after normal fact matching LHS part. Magic module calls are intended as filters, they don't have defined types and need "normal" facts to bind variables to values.

The interpreter has MathMagicModule implementing math.lt(a,b) and math.gt(a,b) relations (that insert float comparing filters).

[3] Perhaps a version of the semantics where such facts are considered inconsistent might be found more useful.

5 Efficiency

Running rules for the first time takes about 3s for Drools compiler to process even simplest rules. This time decreases to 0.3s after a few recalculations by the same application instance. Drools compiler libraries are probably loaded by JVM, perhaps some JIT optimizations are made, maybe something other optimizations are performed here as well. Whatever the cause is, the first 4QL Runner's compilation of rules takes significantly more time.

For testing performance with larger data sets "binary string" facts were used. Simple rules of the form:

"bin(0). bin(1). binStr(A,B,...) :- bin(A),bin(B)...."

have been used to generate facts corresponding to binary strings of given length:

[binStr(0,1,0,1,1,1,1,0,...). etc.].

A 4QL program that generates 32768 (2^{15}) "binary string" facts takes about 3s to calculate the well-supported model (excluding compilation time). This calculation takes JVM about 200MB heap space. The test shows that the current 4QL Runner is not suited for massive data sets, but can handle non-trivial data sizes. It appears the the main problem is the memory use. This is caused by the decision to represent every value as a string, duplicating facts on many lv's and application of RETE algorithm.

Nevertheless, computation time and memory consumption indicate that 4QL Runner is sufficient for reasonably large databases.

6 Ideas for Future (What Can or Should Change in the Future)

- Brain communication API - the current API is the first proposal, there is no software using it, yet; it's highly probable that some part of it will change. In an unlikely event the interface will not change, string queries will certainly need some standards of communication defined; each new query type raises an issue of storing and processing facts in order to answer those queries efficiently.
- Better communication via Brain communication API - in order for this API to be useful, a code providing an abstraction layer for other technologies will be needed. Typical examples can be:
  - a fourQLBrain that maps SQL database content to facts
  - a Brain responsible for communicating sensor data, processing it and providing higher level conclusions
  - a Brain that interacts with a human user and expresses received information in terms of 4QL facts for further reasoning.

  It would also be useful to develop Brain API's communication towards having "gateway Brains" acting as tunnels, where API procedure calls for

facts would be transported, e.g., via a TCP/IP connection, enabling 4QL based system to be distributed across a network.

- Brain API system analysis tools - if such complex 4QL based systems are developed, tools to analyze their topology and dependencies might be required. Currently it is possible to make a cyclic dependency between 4QL Brains, resulting in endless cycle of fact value changes. Analyzing or at least monitoring of dependencies might be needed.

- More efficiency - the current way of implementing 4QL using Drools is far from perfect; it should be good enough for prototyping, but with bigger amount of data and possibility to generate not very useful facts during well-supported model calculation, the efficiency might be an issue in more practical applications. Due to time constraints under which the current 4QL Runner has been developed, the decision was to strictly follow the 4QL standard algorithm. With a better understanding of both Drools and 4QL, it might be possible to find an algorithm better suited for rule systems. Even with the current ideas, a closer integration with Drools and more advanced use of its internals might improve performance significantly. Also, one could consider another rule system or develop a new version of RETE, tailored for 4QL.

- More software - software using 4QL as its database/reasoning engine is desirable. 4QL Runner code base can be sufficient to prototype and test ideas; it would also be good to see what features of 4QL work well in applications, and what needs more work.

- Split of rule compilation and rule running - in agent systems it might be a good idea to compile rules on one machine and send them to a system with more limited resources and run the obtained system there. The current code almost allows for that, based on Drools serialization mechanisms and a simple code. However, depending on the specifics of a given limited environment such a mechanism may be trivial but in some cases it can be not that easy to actually design it.

- Better 4QL - there is 4QL+ already on its way [4], allowing for more complicated rules, lowering the requirements on module dependencies and generally adding new constructs to 4QL. With the Brain communication, 4QL and 4QL+ systems should still be able to communicate.

References

[1] J. Małuszyński and A. Szałas, *Living with Inconsistency and Taming Nonmonotonicity*, Datalog Reloaded, O. de Moor, G. Gottlob, T. Furche, A. Sellers, eds., LNCS 6702, 384-398, Springer-Verlag 2011.

[2] Ł. Sobczyk, *Implementacja interpretera 4QL w języku Java* (*Implementation of a 4QL Interpreter in Java*), M.Sc. Thesis, Department of Mathematics, Informatics and Meachanics, University of Warsaw, September 2012. In Polish.

[3] P. Spanily, *The Inter4QL Interpreter*, 2012 (based on M.Sc. Thesis, 2011), http://4ql.org/wp-content/uploads/2012/10/inter4ql.pdf.

[4] A. Szałas, *How an Agent Might Think*, To appear in Logic J. IGPL, 2013. Special issue edited by B. Dunin-Kplicz and R. Verbrugge.

[5] 4QL website, http://4ql.org

[6] ANTLR, http://antlr.org/

[7] Eclipse, http://www.eclipse.org/

[8] The JBoss Drools Team, *Drools Expert*,
http://www.jboss.org/drools/documentation
version 5.4.0.Final, 14-MAY-2012.

[9] Window Builder, http://www.eclipse.org/windowbuilder/