Programming with Sem.Reasoner

Content

1 Introduction

Sem.Reasoner has APIs fro programming for all modules on different levels of abstraction. This tutorial should give developers an immediate start to integrate it as part of his own applications. It is organized in form of different tasks which should be solved with Sem.Reasoner capabilities.

2 Package Overview

For Sem.Reasoner there exists a package called api which contains the api classes for all different modules (see fig 1).

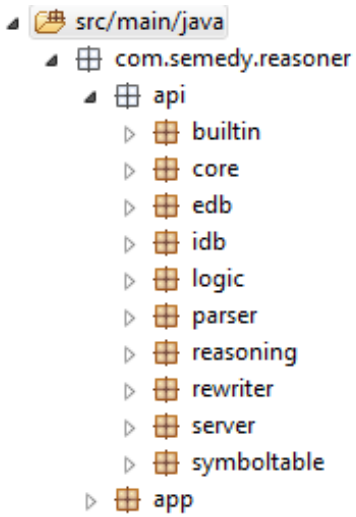

Figure 1. api package

The package builtin contains all apis for the different built-in types and handling of built-ins. The core package provides access to classes central to every application. Access to the extensional database is given in the extensional database package, access to the intensional database (containing the rules) is given in the intensional database package. Definition of logic classes like Literal, Rule is given in the logic package. For parsing OO-logic apis may be found in the parser package. Reasoning functionality is given in the reasoner class. Access and management of rewriters is provided in the rewriter package. Symbol table apis may be found in the symboltable package. edb, idb, parser, reasoning, rewriter, symboltable are all low level apis which are used in the core package, but are not necessary for our first steps in this tutorial. In our tutorial we only need to know the DeductiveDatabase API, the Configuration API which are both part of the core package, the definition of the logic components in the logic package and the built-in APIs which we find in the built-in package.

3 Deductive Database

This class will be the central class for our development. It provides a high level api for handling fact and rules, for querying, for loading files and for backup and restore. Changes in the deductive database can only be applied during transactions. This means that every such a change must be started with a transaction begin and finished with a transaction commit. Transactions can be rolled back with transaction rollback. The reasoner supports two types of transactions: snapshot transaction and long transactions.

Snapshot transactions provide the following functionality:

- Multiple transactions can be executed at the same time.
- One user thread can have at most one open transaction at the same time.
- Facts and rules which are added during a transaction can only be seen by the thread running this transaction (the transaction is isolated).
- Queries executed within a transaction only see the database state of their transaction (committed database state plus transaction-local modifications).
- The modifications in a snapshot transaction are store in RAM and are transferred to the deductive database during the commit operation

Long transaction are different:

- Only one long transaction can run simultaneously on the whole deductive database
- Queries inside this transaction do not see the modifications inside the transaction
- The modifications are stored in the deductive database (are not kept in RAM)
- Snapshot transaction can run in parallel with the long transaction

A deductive database is basically a client to the core system. Every deductive database can add, remove facts and rules which is not visible for the others (during a transaction).

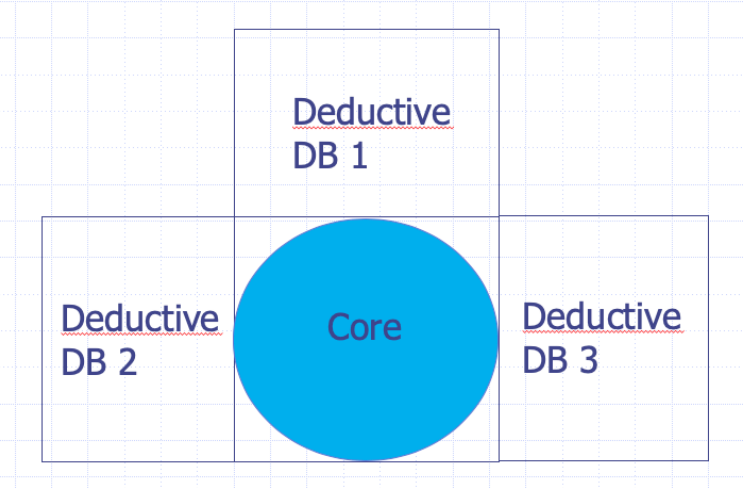

Figure 1. Deductive Database

3.1 A first program

```
public static void main(String[] args) throws EDBException, IOException, SemReasonerExcep-
tion, InterruptedException {

        try (Core core = new Core()) { // create a new core
            InterruptFlag interruptflag = new InterruptFlag(); // for interrupting a query
            DeductiveDatabase database = new DeductiveDatabase(core); // a client for the core

            String ontology =

           "/* ontology */"
                            + "woman::person. // class hierarchy"
                            + "man::person."
                            + "person[hasProperties: {father,mother,daughter,son}].// signatures"
                            + "father[hasRange:man]."
                            + "monter[hasRange: woman]."
                            + "daughter[hasRange: woman]."
                            + "son[hasRange: man].\r\n"

                            + "/* rules consisting of a rule head and a rule body */"
                            + "r1: ?X[son: ?Y] :- ?Y:man, ?Y[father: ?X]."
                            + "r2: ?X[son: ?Y] :- ?Y:man, ?Y[mother: ?X]."
                            + "r3: ?X[daughter: ?Y] :- ?Y:woman, ?Y[father: ?X]."
                            + "r4: ?X[daughter: ?Y] :- ?Y:woman, ?Y[mother: ?X]."

                            + "/* facts */"
                            + "Abraham:man."
                            + "Sarah:woman."
                            + "Isaac:man."
                            + "Isaac[father: Abraham, mother: Sarah]."
                            + "Ishmael:man."
                            + "Ishmael[father: Abraham, mother: Hagar]."
                            + "Jacob:man."
                            + "Jacob[father: Isaac, mother: Rebekah]."
                            + "Esau:man.\r\n"
                            + "Esau[father: Isaac, mother: Rebekah].";
            // add rules and facts to database
            try {
                database.transactionBegin();
                database.addRulesAndFacts(ontology);
                database.transactionCommit();
            } catch (Exception e) {
                database.transactionRollback();
            }

            // pose query
            try (ResultEnumerator it = database.query(interruptflag,
                 "?-  ?X:woman, ?X[son: ?Y], ?Y[father: Abraham]. ", -1, null)) {
```

```
                ResultBuffer buf = it.getBuffer();
                while (it.hasMoreElements()) { // print results
                    it.nextElement();
                    for (int i = 0; i < buf.length(); i++) {
                        System.out.print(buf.get(i).toString());
                        if (i < buf.length() - 1)
                            System.out.print(", ");
                    }
                    System.out.println();
                }
            }
        }
    }
```

3.2 Skeleton for our excercises

The following piece for code provides a skeleton for out applications and could be copied into our Java IDE.

```
public static void main(String[] args) throws EDBException, IOException, SemReason-
erException, InterruptedException {

        try (Core core = new Core()) { // create a new core
            // for interrupting a query
            InterruptFlag interruptflag = new InterruptFlag();
            // a client for the core
            DeductiveDatabase database = new DeductiveDatabase(core);
            String ontology = “”; // define some facts and rules here

            // add rules and facts to database
            try {
                database. (); // open a transaction
                database.;    // add rules and facts
                database.;    // commit transaction
            } catch (Exception e) {
                database.;    // rollback transaction in case of a failure
            }

            // pose query and print results
           try (ResultEnumerator it =
                    database.query(interruptflag, "?-query().", -1, null)) {
                ResultBuffer buf = it.getBuffer();
                while (it.hasMoreElements()) { // print results
                    it.nextElement();
                    for (int i = 0; i < buf.length(); i++) {
                        System.out.print(buf.get(i).toString());
                        if (i < buf.length() - 1)
                            System.out.print(", ");
                    }
                    System.out.println();
                }
            }
        }
    }
```

3.3 Exercise 1: Add Facts and Rules and Query

In our first exercise we fill in the gaps in our skeleton. We want to add some data to the deductive database and query for them afterwards. Use the data and query from the OO-logic documentation 2.2.1.

3.4 Exercise 2: Simple Command-line Interpreter

Now we will develop a simple command line interpreter. The interpreter gets to arguments during launch. The first argument is a file path to a text file containing OO-logic. The second argument is an OO-logic query. A file containing OO-logic as text is loaded by the loadPredicateFiles method in the deductive database class. Please fill in the gaps.

```
public Exercise2() {
    }

    // print the results
    static void printResults(ResultEnumerator it) throws {
            ResultBuffer buf = it.getBuffer();
            while (it.hasMoreElements()) { // print results
                it.nextElement();
                for (int i = 0; i < buf.length(); i++) {
                    System.out.print(buf.get(i).toString());
                    if (i < buf.length() - 1)
                        System.out.print(", ");
                }
                System.out.println();
            }
     }

    public static void main(String[] args) throws Exception {

        try (Core core = new Core()) { // create a new core

            // create a deductive database client
            DeductiveDatabase ddb = new ;

            // path of OO-logic file to be loaded
            ddb.load

            // query
            try (ResultEnumerator enm=ddb.query(new InterruptFlag(), "", -1, null)) {
                printResults(enm);
            }

        }
    }
```

3.5 Exercise 3: Command-line Interpreter

Extend the simple command-line interpreter of our previous exercise so that an arbitrary number of files can be loaded and after launch in a loop queries are read from the console and the results are printed to console.

3.6 Exercise 4: Use of Built-ins

Use a built-in in your query in the command-line interpreter. For instance we want to produce the text “Sahra is Isaacs mother” as a result of a query.

3.7 Exercise 5: Handling of Jsons

Repeat exercise 1-3 with Jsons. For that purpose use JsonDeductiveDatabase instead of DeductiveDatabase. You will find JsonDeductiveDatabase also in package Core.

4 Development of Built-ins

Though Sem.Reasoner already provides a large set of built-ins usually every project need additional ones. Sem.Reasoner provides the following classes of built-ins:

- filter built-ins
    - all arguments are bound to concrete values
    - examples are comparisons like ?X < ?Y
- functional built-ins
    - usually all but one arguments are bound
    - last argument provides the result of the function
    - creates exactly one result for the given arguments
    - example is concat, concat(?S,”hallo”,?T) appends hallo to the string ?S
    - all functionality of the Java packages String and Math are available as functional built-ins
- relational built-ins
    - built-in may create more than one result for its input
    - example is SQRT, _sqrt(4,?X) creates two results: -2 and 2
- connector built-ins
    - are used to access date in external systems or from external services
- aggregation built-ins
    - collect a number of values and produce a result out of those values
    - examples are count, mean etc.
- sensor built-ins
    - are used in conjunction with stream-reasoning
    - they feed in new values into the reasoning process
    - used e.g. for sensors
- action built-ins
    - action built-ins are the only built-ins which occur in the head of a rule
    - they are used to trigger actions dependent on the conditions in the rule body
    - such actions can be: send an e-mail, write some data to an api

Sem.Reasoner can easily be extended by such built-ins. In the following we will learn how to do that.

4.1 Exercise 6: develop a filter built-in

In this exercise we want to develop a built-in that checks whether a text is a correct URL. A filter built-in has input values only, no output values, so all arguments must be ground. First of all we have to develop the filter built-in for itself. Here is a built-in which checks whether the input is a string.

```
public class IsString implements FilterBuiltin {
    protected static java.util.logging.Logger _log = Logger.getLogger("Builtins");
    protected SymbolTable _symbolTable;

    public IsString() {
    }

    public void setProperties(Properties prp) {
    }

    public void setSymbolTable(SymbolTable symbolTable) {
        _symbolTable = symbolTable;
    }

    @Override
    public boolean isEvaluable(BitSet grounds, BitSet variableInstantiations, Ob-
ject[] args) throws BuiltinException {
        return grounds.get(0);
    }

    @Override
    public void init(Object[] args) throws BuiltinException, InterruptedException {
    }

    @Override
    public void evaluationFinished() throws BuiltinException, InterruptedException {
    }

    @Override
    public long getWeight(BitSet grounds, BitSet variableInstantiations, Object[]
args) {
        return 1;
    }

    @Override
    public BitSet objectGeneratingArguments(BitSet grounds, BitSet variableInstantia-
tions, Object[] args) throws BuiltinException {
        BitSet result = new BitSet();
        return result;
    }

    @Override
    public String getName() {
        return "_isString";
    }

    @Override
    public int getArity() {
        return 1;
    }

    @Override
    public BuiltinType getType() {
        return BuiltinType.FILTER;
```

```
    }

    @Override
    public boolean isInternal() {
        return false;
    }

    @Override
    public String getDescription() {
        return "is the argument a string";
    }

    @Override
    public String[] getArgumentDescriptions() {
        return new String[] {"calendar"};
    }

    @Override
    public boolean readsSymbolTable() {
        return true;
    }

    @Override
    public boolean writesSymbolTable() {
        return false;
    }

    @Override
    public boolean isTrue(long[] input) throws InterruptedException, IOException ..{
        try {
            Object s0 = _symbolTable.getValue(input[0]);
            return s0 instanceof String;
         } catch (SymboltableException e) {
            throw new ReasoningException(e.getMessage());
        }
   }

    @Override
    public Builtin cloneBuiltin(){
        return new IsString();
    }

    public void clear() {
    }


}
```

Use this as a blue print for your built-in checking for correct URLs.

As soon as we have such a built-in it must be registered to be usable within OO-logic. This is done with the following statement in case of our IsString built-in:

```
core.getBuiltinProvider().registerBuiltin(new IsString());
```

From now on the built-in is available in OO-logic. For instance we can use it in the following query:

```
?- ?X:father, _isString(?X).
```

4.2 Exercise 7: develop a functional built-in

Now we want to develop a functional built-in. A functional built-in creates exact one result for each combination of valid input values. In many cases it has one output value. We want to develop a simple mathematical function *plus*, which is able to add two numbers and return the result. A special feature will be, that this built-in allows any two of its three arguments to be bound, i.e. it also computes the right value if for instance the first argument is not bound: _plus(?X,3,12) delivers ?X = 9.

So in a first step the skeleton for a functional built-in has to be created.

As mentioned the built-in can be evaluated as soon as two of the three arguments are bound. This is expressed in isEvaluable method:

```
public boolean isEvaluable(BitSet grounds, BitSet variableInstantiations, Object[] args)
throws {
        return variableInstantiations.cardinality()==2;
}
```

We get the arguments in the method evaluate as an array of long values. These codes encode the real values. We have to decode them using the symbol table. Then we consider all possible cases (bindings of variables):

```
public boolean evaluate(long[] input, BitSet grounds) throws InterruptedException, IOExcep-
tion, SemReasonerException {
        Integer[] args = new Integer[3];
        for(int i = 0; i < args.length; i++) {
            if (grounds.get(i)) {
                Object arg = symbols.getValue(input[0]);
                if (!(arg instanceof Integer))
                    return false;
                args[i] = (Integer)arg;
            }
        }
        // first two arguments are given
        if (grounds.get(0) && grounds.get(1)) {
            if (grounds.get(2)) {
                // check for a correct result if last argument is given
                if (args[2] == args[0]+args[1])
                    return true;
            } else {
                // compute last argument
                args[2] = args[0]+args[1];
                return true;
            }
        }
        // all other cases
        .....

        return false;
    }
```

Again the built-in has to be registered and then we can test it with the following query:

```
?- _plus(4,?X,6).
```

4.3 Exercise 8: develop a relational built-in

A relational built-in may create several results for each combination of valid input values. We will develop the built-in sqrt which computes the square root of a number. This built-in has one input argument which has to be bound and one output value.
Don't forget to define the name (starting with underscore) and the number of arguments at the appropriate places.

The built-in produces several results which are sent back into the reasoning process. For this purpose the method gets a BuiltinReceiver which allows to send the results.

```
    public void evaluate(long[] input, BitSet grounds, BuiltinReceiver receiver)
throws InterruptedException, IOException, SemReasonerException {
        Object number;
        try {
            // decode the input value
            number = _symbolTable.getValue(input[0]);
            Double result = 0.0;
            // sqrt of the input value
            if (number instanceof Double)
                result = Math.sqrt((Double) number);
             else if (number instanceof Long)
                result = Math.sqrt((Long) number);
             else if (number instanceof Integer)
                 result = Math.sqrt((Integer) number);
             else
                 return;
            // there are two results: result, -result
            // encode result
            long code1 = _symbolTable.encode(result);
            long code2 = _symbolTable.encode(-result);
            input[1] = code1;
            // send the first result away
            receiver.send(input);
            // send the second result away
            input[1] = code2;
            receiver.send(input);
        } catch (SymboltableException e) {
            throw new ReasoningException(e.getMessage());
        }
    }
```

Again the built-in has to be registered and then we can test it with the following query:

```
?- _sqrt(4,?X,6).
```