Published on *Multilingual Online Translation* (http://www.molto-project.eu)

D6.3 Assistant for solving word problems

Contract No.:	FP7-ICT-247914
Project full title:	MOLTO - Multilingual Online Translation
Deliverable:	Assistant for solving word problems
Security (distribution level):	Public
Contractual date of delivery:	December 2012
Actual date of delivery:	May 2013
Type:	Prototype
Status & version:	Final
Author(s):	Jordi Saludes
Task responsible:	Jordi Saludes
Other contributors:	

Abstract

We will introduce a prototype for dealing with simple arithmetical problems involving concepts of the physical world (word problems). The first software component allows an author to state a word problem by writing sentences in several languages and converting it into Prolog code. The second component takes this code and presents the problem in the student's language. Then it provides step-by-step assistance in natural language into writing equations that correctly model the given problem.

1. Introduction

This software deliverable is a prototype of a word problem solver, namely a system that interactively poses a word problem, (in many languages), then constructs a solution and a reasoning context for it. The overall architecture is based on the usage of third-party, open-source software components to provide the reasoning infrastructure for the system and are not distributed in this deliverable.

This document describes:

- how to install the prototype,
- how to create word problems involving simple arithmetic;
- how to assisting a student into finding the equations related to it.

The first component is a Scala library (http://www.scala-lang.org) to be used inside the Scala Interpreter shell (in a Read-Evaluate-Print Loop), while the second component is a dialog system which runs in the command line. Both components were developed within the framework of the MOLTO project.

2. Installation

The source code for this deliverable can be downloaded from the MOLTO svn repository by:

```
svn co svn://molto-project.eu/mgl/wproblems
```

It will appear into the `wproblems` directory. But prior to building the system for the natural language interpretation of the word problems, you need to install the external components that are handling the computational aspects of the system.

Software requirements

Third-party software components provide the following functionalities in the system:

- SWI prolog; (in our architectures, (1) **SWI-Prolog version 6.2.2 for i386-darwin11.3.0**, (2) **SWI-Prolog (Multi-threaded, 64 bits, Version 6.2.6**). Set the environment variable `SWIPL_LIBDIR` to the path to the swi-prolog library. The prototype employs Prolog as domain reasoner for certain schemata of word problems.
- Scala; (in our architectures, (1) **2.9.2**, (2)**Scala version 2.10.1 (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_43**) - Scala is a general purpose programming language designed to express common programming patterns using both object-oriented and functional paradigms. Conciseness is a key feature of Scala (scalable language). The prototype employs Scala for constructing, from natural language input, a Prolog program that can solve the problem.

In addition, the system requires the availability of the jpl library for accessing Prolog from Java code. It is installed by the prolog installer, check that the `jpl.jar` exists and write down the path (It will be needed for the configuration step below).

Project-related software components:

- GF; (in our architecture, version **version 3.4**). GF is used to provide the natural language parsing and generation in the Dialog system.

- gf-java to use the GF web services from Java; (distributed under `lib`, version **gf-java-0.8.1.jar**).

Install all the components as directed. Now configure by passing `SWIPL_LIBDIR` for the path to the swi-prolog library and `JPL_LIBDIR` for the path to the directory containing `jpl.jar`. In our case:

```
./configure SWIPL_LIBDIR=/opt/local/lib/swipl-6.2.2/lib/i386-darwin11.3.0/ JPL_LIBDIR=/opt/local/lib/swipl
```

and then build the system

```
make
```

3. Quick examples of usage

Having finished the installation step, we are now ready to use the system[1].

In this document *word problem* means a mathematical problem requiring writing the equations describing all the relevant information needed to get the solution. We we'll split the solution of such a problem into:

Modeling:
: Finding out the equations describing all the relevant information needed to get the solution. This requires the student to use common sense reasonong abot the world and the ability to write these relations in the mathematical language.

Solving:
: Determining the solution by handling the equations in a pure formal way.

There are a lot of applications for helping students with **solving** step but only a few for the **modeling** step. We will present here a prototype for addressing this step for problems requiring just elementary arithmetics.

The system allows two modes of usage, for authors (teachers) and for students:

- creation of a word problem
- tutorial dialog for solving a word problem.

The first application runs inside the Scala REPL, and consists in a library implementing the class `Problem` with resources for constructing problems from natural language sentences. Problems are saved as Prolog clauses with comments used to reconstruct the originating sentences. The second application is a Scala executable that loads a saved problem and engages the student in a nalural lenguage dialog conducting to have the problem correctly modeled.

Both applications use a Prolog database to reason about the problem. The basic difference is that for the author tool, the system constructs the model automatically in order to check if the problem is *consistent* (it does not contain contradictions) and *complete* (it has enough information to give a single solution), while for the student tool, the model construction is driven by the sentences proposed by the student. The system leads the student through several discovering steps (see next section) and checks that the proposed sentences are correct and relevant.

Writing a word problem

Invoke the author tool by:

```
./create
```

Create a new problem to be saved into file **fruit.pl**

```
scala: val p = new Problem("fruit.pl")
p: wp.Problem = Problem with 0 statements
```

We could use the `Statement` class to add new statements to the problem with the += operator. However, it is more convenient to define a *statement factory* for entering them in natural language (denoted by its 3-letter ISO code):

```
scala: val en = new StatementFactory("Eng")
```

We can now use a predictive parser to enter a new sentence into the problem:

```
scala: p += en.read
Eng: John has seven fruit .
```

Notice the final period. We can keep track of how many statements our problem has by:

```
scala: p
res1: wp.Problem = Problem with 1 statements
```

Let us add some more facts:

```
scala: p += en.read
Eng: John has two apples , some oranges and three bananas .
scala: p += en.read
Eng: how many oranges does John have ?
```

To take a look to the internal representation of the problem, use `print`:

```
scala: p.print
```

We can check if the problem is *consistent* (it does not contain contradictory statements) or *complete* (it has a single solution) by using the methods `consistent` and `complete`:

```
scala: p.complete
res3: Boolean = true
```

Remember to save the problem:

```
scala: p.save()
Saved to 'fruit.pl'
```

and now we can exit:

```
:q
```

Solving the problem

We can now try to solve our problem, by calling `model` with the file containing the problem:

```
 ./model fruit.pl
```

It shows us the statement of the problem:

```
John has seven fruit .
John has two apples , some oranges and three bananas .
how many oranges does John have ?
```

and displays the prompt:

```
 ?
```

We can always press return at the prompt (or type `help`) for the system to suggest the proper action:

```
you must assign a variable to the oranges that John has .
```

But we do not know how to assign variables. Let us ask for an example:

```
? give me an example
let $x$ denote the animals that Mary has
```

Using this template we can now compose a definition for the variable *x*:

```
? let x denote the oranges that Mary has
you must assign a variable to the oranges that John has .
```

I forgot that we were dealing with John's fruit, not Mary's:

```
? let x denote the oranges that John has
it is right .
```

Press again return for the next suggestion:

```
you must split the fruit that John has .
```

This means that we have to specify how John's fruit are split in different classes:

```
 ? the fruit that John has are the apples that John has and the bananas that John has
 you must consider oranges .
```

Yes, there are oranges too. Let us correct it:

```
 ? the fruit that John has are the apples that John has , the bananas that John has and the oranges that Joh
 it is right .
```

Good. Next suggestion:

```
you must write an equation which says that the fruit that John has are the bananas that John has ,
the oranges that John has and the apples that John has .
```

What about this?

```
? y plus 2 plus 3 is equal to 7
it doesn't follow .
```

This means that the proposed equation can not be deduced from the statement of the problem. Let us see what is wrong with the variable *y*:

```
? tell me about y
nothing is known about it .
```

Perhaps we used a different variable to denote the amount of oranges:

```
? tell me about the oranges that John has
the oranges that John has are $x$ oranges .
```

So we used *x* for it. Just to confirm it:

```
  ? tell me about x
  $x$ denotes the oranges that John has .
```

We rewrite the equation using *x*:

```
? x plus 2 plus 3 is equal to 7
it is right .
```

Now the problem is correctly modeled. The next action will give us the solution:

```
the oranges that John has are two oranges .
```

Going multilingual

To run the same problem but in Spanish, add the 3-letter-ISO code of the language as second argument:

```
./model examples/fruit.pl spa
...
Juan tiene siete frutas .
Juan tiene dos manzanas , algunas naranjas y tres plátanos .
¿ cuantas naranjas tiene Juan ?
```

Asking for help:

```
?
```

```
debes asignar una variable a las naranjas que Juan tiene .
```

Asking for an example:

```
? dame un ejemplo
denota las cartas que María tiene por $z$
```

---

1. The system will start/stop the GF-java service for you, but if you run into trouble you can check the state of the service by: `bin/wpserver status` and stop it by: `bin/wpserver stop`. ↩

4. Reasoning aspects of word problems

Word problem schemata

The current prototype allows to state word problems of the following form:

```
   John|Mary has|owns  one|two|...|seven|some fruit|apples|oranges|bananas|animals|rabbits|cows
```

in the languages: English, Catalan, Swedish and Spanish.

Amounts

The building block for the reasoning is the **amount**: a relaxed version of a **set** in which one does not have access to the composing elements, but can know the number of elements in it.

An *amount* is constructed by:

- Giving the cardinal and the **class** of its elements (i. e. **three oranges**). Notice that the cardinal may be undefined (i. e. **some oranges**);
- The `own` predicate binding an individual and a class (i. e. **the apples that John has**);
- Disjoint unions of these constructions (**three apples and two oranges**)

Propositions

Available sentences express the equality between two **amounts** (i. e. **The fruit that John has are two apples and some oranges**)

The modeling process implies transforming a set of propositions into another set in which the numerical interpretation is evident.

Setting the problem model

We consider two grammars to express these facts:

- The *plain* language is for direct communication with the user;
- The *core* language is for the reasoner to work with.

This is how we express the amount **John apples** in *plain* (Prolog concrete):

```
own(john, apple)
```

while in *core*:

```
p(X, apple, own(john,X))
```

The latter is more suited to reasoning with it.

Another step into *normalizing* (making it *core*) an amount is to disaggregate sums. In this way a statement like **John has three apples and six bananas** is converted into: **John has three apples** and **John has six bananas**.

Another case is to convert questions as **how many apples does Mary have?** which are represented in *plain* as:

```
find(own(mary,apple))
```

into the *core* expression:

```
find(X, apple, own(mary,X))
```

A set of statements in *core language* is what is needed to process a word problem. This is what the `create` tool saves: A Prolog file consisting of:

- A GF abstract tree for the plain sentence of a problem. This is written as a Prolog comment.
- Core statements in Prolog format that correspond to the plain expression.

As an example, this is a complete problem in *core* Prolog clauses. The comments contain the GF abstract tree corresponding to the original *plain* expression:

```
% abs:fromProp (E1owns john (gen Fruit n7))
% Eng:John has seven fruit .
-(p(_1, fruit, own(john, _1)), *(7, unit(fruit))).
% abs:fromProp (E1owns john (aplus (ConsAmount (gen Apple n2) (BaseAmount (some Orange) (gen Banana n3)))))
% Eng:John has two apples , some oranges and three bananas .
-(p(_5, apple, own(john, _5)), *(2, unit(apple))).
-(p(_6, banana, own(john, _6)), *(3, unit(banana))).
-(p(_7, orange, own(john, _7)), some(orange)).
% abs:fromQuestion (Q1owns john Orange)
% Eng:how many oranges does John have ?
find(_19, orange, own(john, _19)).
```

Workflow for modeling a problem

When the `model` tool is started on a word problem file, the system uses the GF abstract lines to display the statement of the problem in the selected language. Now the student must go through a sequence of steps to have the problem correctly modeled:

1. **Assigning variables**. At the beginnig the student must choose variables to designate unknowns that are relevant to the problem. This includes the target unknowns (they appear as arguments of `find` clauses) and expressions like **some apples**.
2. **Discovering relations**. In this step the student has to combine information from different statements into new relations. For example, decomposing the fruits that John has into the apples and bananas that John has.
3. **Stating equations**. In the next step, the student converts the relations uncovered in the previous step into numerical equations. This steps finishes when there are enough equations to determine the unknowns of the problem. The system checks that the student's equations are consistent equations and are entailed by the problem information.
4. **Final**. At the last step, the system displays the solution for the unknowns of the problem and exits.

5. Current limitations and future work

The current prototype is a proof of concept aiming at demonstrating that the semantics of word problems can be handled given a formalization of the specific domain, a decision procedure on the resulting model, and a natural language application that allows to express and semantically interpret the facts describing the specific world instance.

In this work we have considered problems of a specific kind but we maintain that, in the e-Learning scenario, which is our target area of application, word problems can be classified according to schemes which can be formalized along the lines shown here. In many problems, understanding of natural language formulation is translated to facts in the knowledge base where two seemingly independent facts are put in a relation and become a new assumption for solving the problem (**if A is an animal tamer, then A is not afraid of animals**. **if F is the father of S, then F is older than S**. **Every orange is a fruit**). Construction of the correct assumption can be done in an exhaustive way only under a finite world assumption (what is known is what it is explicitly stated).

Future work

Replacing the Prolog engine by a *proof assistant*. These systems delivers **proofs** of propositions in a theory. By using a theory supporting a kind of word problems and forcing the problem author to express the problem as a a valid theorem in this theory would have the benefit of uncover hidden assumptions on the problem statement.

Also, these systems being more expressive than Prolog clauses, and supporting complex *tactics* for automatic proving could benefit the maintenance of the system.

On the student side, the discovering of new facts is converting into asserting propositions that can be transparently proved by tactics or presented to the student to deal with them: This would lead to re-using the same problem in different educations levels according to what is assumed and what is proved by the student.

Source URL: http://www.molto-project.eu/wiki/living-deliverables/d63-assistant-solving-word-problems