DE GRUYTER OPEN Acta Univ. Sapientiae, Informatica **8**, 1 (2016) 41–62

DOI: 10.1515/ausi-2016-0003

Contralog: a Prolog conform forward-chaining environment and its application for dynamic programming and natural language parsing

Imre KILIÁN
University of Dunaújváros
Dunaújváros, Hungary
email: kilian.imre@uniduna.hu

Abstract. The backward-chaining inference strategy of Prolog is inefficient for a number of problems. The article proposes Contralog: a Prolog-conform, forward-chaining language and an inference engine that is implemented as a preprocessor-compiler to Prolog. The target model is Prolog, which ensures mutual switching from Contralog to Prolog and back. The Contralog compiler is implemented using Prolog's de facto standardized macro expansion capability. The article goes into details regarding the target model.

We introduce first a simple application example for Contralog. Then the next section shows how a recursive definition of some problems is executed by their Contralog definition automatically in a dynamic programming way. Two examples, the well-known matrix chain multiplication problem and the Warshall algorithm are shown here. After this, the inferential target model of Prolog/Contralog programs is introduced, and the possibility for implementing the ReALIS natural language parsing technology is described relying heavily on Contralog's forward chaining inference engine. Finally the article also discusses some practical questions of Contralog program development.

Computing Classification System 1998: Computing methodologies–Logic programming and answer set programming
Mathematics Subject Classification 2010: 68T27
Key words and phrases: logic programming, mechanical theorem proving, forward chaining, dynamic programming, natural language parsing

1 Pro-Contra-Log: a two way street for inference

Robinson's resolution principle proved to be strong enough to build a whole language, and quite a programming school upon it. The backward chaining resolution strategy, together with the left to right and top down rule firing strategies have provided the Prolog language with well known, easily perceivable operational semantics, which in addition, resemble the traditional sequential languages, with the extra flavor of backtracking added [2] .

The other direction however, the forward chaining strategy has never opened such a clear way to follow, though several attempts were made. Though their motivations must have been quite different, spreadsheets, the event driven working mechanism of modern graphical user interfaces, and some CASE tools follow seemingly similar ways.

The most important reason for Prolog's success, as a programming language, could be that its resolution strategy is somehow an extension of the old-fashioned and well-known procedure-call semantics. At the same time, as a theorem prover, we consider it rather weak. Beside many factors, the deepest reason of this might be, that its strategy to manage implications contradicts common human sense. Instead of deducing new facts from existing facts in the direct way, Prolog tries to prove certain facts by applying implications in the inverse direction, i.e. from consequence to conditions.

On the other hand, similarly to the "divide et impera" strategy of algorithmic thinking, Prolog's strategy may also be inefficient. Proven facts are not stored; therefore if a similar fact should be proved later, then the inferential operations are performed again, sometimes causing thereby a significant loss of performance.

Since Prolog, as a language appeared and has proven its strength to certain problems, the need of integrating it with other programming languages and other software tools, was always a burning issue. Similarily, the integration with another logical programming language, i.e. with a language that implements another subset or another inference strategy for the same subset of logic, remained only a theoretical possibility.

The present article proposes a *programmer controlled tight integration* of the two approaches. Tight integration means that the two languages are syntactically conform, their target models are compatible, and their software packages can be integrated with each other. The integration is called programmer controlled, because by means of declarations and/or modularization the programmer is able and is supposed to influence and control the actual code generation strategy for the different code segments.

The programming languages in the present proposal are Prolog and its counterpart: Contralog. They are syntactically compatible, and their declarative semantics are the same. Their procedural semantics however, since they implement different resolution strategies, are different, and they also implement different non-logical controls. Since Contralog's target model is Prolog, a translator program is implemented, which compiles Contralog modules to Prolog. The connection between the two segments is implemented through exports and imports, and it is triggered in a programmer defined way. The common Prolog target model makes the transition between the two segments very easy, and it also allows the easy use of Prolog built-in predicates from Contralog clauses.

Contralog, as an extension of Prolog, maps Horn-clauses to a Prolog code so, that an incremental compiler transforms Contralog rules to Prolog. The resulting code thereby can be executed in a normal Prolog runtime environment. This also ensures that Contralog and Prolog codes can be mixed and invoked from each other. This composite system is called Pro-Contra-Log, or simply PC-Log.

2 The Contralog target model

In the Contralog runtime-model everything works in the inverse way than we are familiar with:

- *Inference starts* not by goals, but *by facts.*
- If there is a rule with a condition matching the new fact, then the rest of the condition literals are also checked. If we could already have proven the conditions earlier, the rule is told to *fire*. In such case the conclusion-fact is told to have been inferred.
- The term: *recently proven conclusion* means a new resolvent fact. This is stored in the blackboard (in dynamic Prolog clauses), and we always continue inferencing using this fact.
- Inference stops when *goal statements are reached.* These don't have any consequence side, thus inference operations need not be continued.
- Upon reaching a goal, or when by any reason, inference cannot be continued along the actual chain, the system *backtracks and searches for an earlier open alternative* in the inference chain.

The Prolog-Contralog transition can be activated in the following ways:

- In the precondition side of Contralog rules, literal "{}/1" results an *immediate call* to a Prolog goal.

- *Imports* in Contralog are those facts which are taken from somewhere else. Such facts start an inference chain, therefore Contralog imports are mapped to Prolog exports of the corresponding firing rules.

- On the other hand, Contralog *exports* are mapped to firing predicates of freshly deduced facts. Those predicates are either imported from another Contralog module, thus they are defined there, or simply are imported from another Prolog module of the runtime environment. Contralog exports become Prolog imports (even though Prolog standard does not know this lingual construct).

The basic problem in forward chaining inference is that a rule might refer to more than a single condition. In case when not all of them can be satisfied, we must wait until the rest becomes true, and the rule can be fired only afterwards. We are solving this by storing the inferred facts in dynamic predicates. Furthermore, for each Contralog condition literal we construct a Prolog rule that checks if the other preconditions are already satisfied.

Let us regard the following Contralog rule, as a simple example!

```
a:-b, c, d.
```

If b or c or d conditions have already been satisfied, then the resulting facts are stored in the corresponding b/0, or c/0 or d/0 dynamic predicates. Besides those, we map each precondition literal to a `fire_NAME` and a `test_NAME` Prolog predicate.

The `fire_FACT` Prolog calls test, whether the rest of conditions (each but NAME) have already been inferred. If they have, the rule triggers the evaluation of the consequence part.

The `store_FACT` Prolog calls serve to store the recently inferred fact, and finally they call the firing predicate. The actual implementation takes into account the declarations for fact withdrawal, and performs the necessary actions. For predicates blocking the resolution chain, the call of firing the rule is missing.

In the case above, the following Prolog code is constructed:

```
fire_b:- assert(b), test_b.
fire_c:- assert(c), test_c.
fire_d:- assert(d), test_d.
test_b:- c, d, fire_a.
test_c:- b, d, fire_a.
test_d:- b, c, fire_a.
```

2.1 Backtracking

As it was already hinted, this target model uses *backtracking strategy* for searching, like Prolog itself does. Upon entering a new Contralog fact, the inference process produces newer and newer consequence facts. During this process there are some *choice-points*. If the straight line of inference described above cannot be followed further, we say, the evaluation has reached a *dead end*. In case of a dead end, the inference engine looks back in the evaluation stack, searches for the last open choice-point, takes the next alternative solution, and continues the interpretation from this point.

The following cases may produce a choice-point:

- If a condition literal is referred to by more than a single Contralog rule, then a Prolog definition is constructed consisting of the same number of Prolog `fire_` rules. Their ordering to follow is the textual ordering.
- If a condition literal is satisfied several times, we store the same number of dynamic facts – provided they are not in the scope of the non logical declarations described later.

Trying new open choice points is called *backtracking*. Finding dead-ends in the inference chain may happen in several ways:

- If any condition does not hold in the given time. This may be either the failure of a Contralog condition, or, using the {}/1 construct, any immediate Prolog call.
- If, upon reaching a Contralog goal, we force the system to backtrack by any Prolog means.

2.2 Facts and goals

Facts and goals play somewhat an opposite role in Contralog, than in Prolog. Facts are basic information sources, which start the data driven resolution, therefore they are compiled to goals.

`fact.` a Contralog fact.

`:-fire_fact.` a Prolog goal.

Many Prolog systems do not handle the presence of several logical goals correctly, i.e. in case one of them fails, they do not call the next one. Because of this the actual implementation constructs a single predicate (*goal/0*) that invokes the firing predicate of each fact in the current module in an alternative sequence:

`fact1.` a Contralog fact1.

`fact2.` a Contralog fact2.

The example above generates the following predicate:

```
goal:-fire_fact1.
goal:-fire_fact2.
```

2.3 Interface elements

Similarly to the overall opposite nature of Contralog, interfaces in the compiled Prolog code also play an opposite role. For the sake of explanation the interface is predicate-based, and exports but also imports(!) are allowed.

A Contralog export means a consequence part of a clause, which is shared with the outside world. Therefore the corresponding firing Prolog call must be implemented outside of the module and is thus imported. For convenience the generated predicate name is slightly different from the firing predicate. Contralog imports mean some input of more basic data, which, seen from the view of the target model, would be again considered to be calls from outside, which transfer the information, and launch the corresponding forward chaining inference process. This means, the corresponding `fire_` auxiliary predicate is implemented inside, and thus must be exported.

2.4 Non-logical means to control inference

In order to control the overwhelming amount of consequence facts, the pure logic language must be provided with means to control. Similarly to Prolog, for the sake of practical programming, Contralog introduces some non-logical means to influence the resolutional strategy. The language therefore implements certain declarational forms for this purpose.

In each target module a predicate (`clean/0`) is generated that cleans the module-specific part of the blackboard (the dynamically generated facts) completely.

A predicate P is told to *block the inference chain*, if, whenever any successful application of resolution steps yields a fact p, matching P, then the fact p does not cause to search for matching conditions. That is, though its immediate consequences are not evaluated, the new fact still remains available for further resolution. In such a case the new fact is stored in the blackboard, but the corresponding firing predicate is not invoked. Such an operation is performed for predicates denoted by the `:-lazy NAME/ARITY.` declaration.

A predicate P is told to *withdraw its earlier results*, if, whenever any successful application of resolution steps yields a fact p, matching P, then some, or all of earlier resolved p1,...,pn facts, all matching P, are excluded from further resolution. In Contralog the following withdrawal schemes are implemented.

- *Full withdrawal.* The predicate P withdraws all of its earlier results. The behavior is similar to that of a normal global variable, that is upon inferring certain facts, other facts with the same signature are deleted from the blackboard. To reach such effect the declaration `:-var NAME/ARITY` can be used.

- *Key controlled withdrawal.* Certain arguments of the predicate are told keys. The predicate P withdraws only those earlier results, whose keys are matching those of the recently inferred fact, p. (The behavior is similar to that of a table in a relational database, where unique keys are defined, i.e. elements with same keys are simply overwritten.) This can be reached by the `:-key(NAME(KEYVECTOR))` declaration, where NAME is the name of the predicate, KEYVECTOR is a list of argument patterns. Pattern "+" in KEYVECTOR denotes that the argument belongs to a (composite) key, pattern "–" denotes the opposite.

3 Contralog programming examples

3.1 Pythagoras' triads

Generating Pythagoras' triads can be among the first examples of beginners' Prolog courses. Let's see, how it works with Contralog. Peano's axiom is expressed by a predicate of two clauses, similarly to Prolog:

```
natural(1).
natural(X):- natural(X1), {X is X1+1}.
```

A Contralog recursion may also start an endless inference chain: stating the first fact launches the rule application that launches it again and again. This can be avoided by placing the recursive clause as the last among the referring clauses. This means when `natural/1` fires, each other firing procedure attached to `natural/1` is called before the rule above. Among these, those procedures which generate Pythagoras' triads are also fired. Firing `natural/1` for its own recursive rule occurs only when the generated triads do not satisfy Pythagoras' condition, or, when upon finding a perfect triad, the user asks for a new result, thus forcing the system to backtrack.

One feasible strategy to generate integer pairs of a quarter-plane is to visit them in a diagonal way. For one diagonal, the sum of its x and y coordinates is invariant. (For convenience we allow here the X=0 value too.) The Contralog predicate, implementing this, is the following:

```
natural2(0,SUM,SUM):- natural(SUM).
natural2(X,Y,SUM):-
  natural2(X1,Y1,SUM),
  {X1<SUM, X is X1+1, Y is Y1-1, X<Y}.
```

Predicate `natural/1` produces the invariant sum of both (X and Y) coordinates. This, with X=0 value, also gives the first integer pair. The sum is passed on through the third parameter of `natural2/3`. Given the actual pair, the recursive second clause takes care to produce the next pair. Endless inference is blocked by referring to `natural2/3` in another predicate: it is also a precondition in predicate `natural3/4` that generates integer triads. On the other hand the arithmetical tests in `natural/2` ensure that numbers remain not only in the given quarter-plane (X1<SUM), but also in an eighth-plane (X<Y).

Visiting integer triads happens slightly differently. The last two (Y and Z) coordinates are generated by `natural2/3`, but the first coordinate (X) is generated using the (0<X, X<Y) conditions, by a simple scanning of the (0;Y) integer range. Conditions (X>0,Y>0,Z>0) allow to search only in an eighth of the three-dimensional space, while (X<Y<Z) conditions part it to halves twice further on. A 32-fold reduction of the total searching space is a significant result in time, while we don't lose any characteristic results.

```
natural3(X,Y,Z,SUMXZ):-
  natural2(Y,Z,SUMXZ), {X is Y-1, X>0}.
natural3(X,Y,Z,SUMXZ):-
  natural3(X1,Y,Z,SUMXZ), {X is X1-1, X>0}.
```

Once we can generate integer triads, checking for Pythagoras' triads is an easy job by the following clause:

```
pyth3(X,Y,Z):-
  natural3(X,Y,Z,_),
  {SUM2 is X*X+Y*Y, SUM2 is Z*Z}.
```

There are several possibilities to make this program run:

- The most suitable way to start the program is to call `:-p3:goal.`, which is an automatically generated procedure in module `p3`. But if we leave the clause above in the present implicational form so, that no clause refers to `pyth3/3` in the condition side, then a reference is generated to its firing predicate, which remains unsatisfied, thus we get an undefined procedure Prolog error. If we declare `:-export([pyth3/3]).`, then, instead of a firing predicate, the Contralog translator generates a call to the Contralog-exported predicate `exp_pyth3/3`. For this we must provide a testing environment, a module to use our original module, and to define the missing predicate somehow like the following:

  ```
  exp_pyth3(X,Y,Z):-
    write([X-Y-Z]), nl.
  ```

- If we call thereby a simple goal, then it will generate triads, but for the first triad, matching Pythagoras' condition, the inference will stop. The generated results are displayed by the `exp_pyth3/3` predicate above. To get subsequent results the system must be forced to backtrack by the following way: `:-p3:goal, fail.`, or simply by pressing ";" in the SWI-Prolog environment.

- If we declare `:-lazy pyth/3.`, and we make the system backtrack, then it will generate Pythagoras' triads in and endless loop and it will collect the results in dynamic predicate `pyth3/3`, until some system limit is reached.

If we do all this, except the `lazy` declaration, then we get the first familiar result, and if – in the SWI-Prolog environment – we force the system to backtrack, than we may also get the rest of them.

```
?- p3:goal.
[3-4-5]
true ;
[6-8-10]
true ;
[5-12-13]
true ;
[9-12-15]
true
```

3.2 Dynamic programming: Optimization of matrix chain multiplication

Matrix chain multiplication is a typical example for dynamic programming [1]. Matrix multiplication is an associative, but not commutative operation. The actual parenthesization, however, influences the efficiency of the entire operation considerably.

When trying to find the optimal parenthesization, the classical "divide et impera" approach leads to an exponential time complexity, while the data driven dynamic solution remains polynomial. The reason of the combinatorial explosion is that "divide et impera" divides the problem two (or sometimes more) parts, which are presumed to be independent from each other. That is, a solution of one part cannot overlap with the solution of any other part. The approach may work even so, but in such cases it forgets the corresponding sub-results, and calculates them again and again multiple times. This causes the loss of efficiency.

The dynamic solution uses a triangle-matrix to store the intermediate results. One element, m(i,j) stores the optimal number of scalar multiplications necessary to compute the i to j (i<=j) subsection of the entire chain. For single element subchains the following supposition holds obviously.

$m(i, i) = 0$

For longer subchains (when j>i), the $m(i,j)$ optimum value can be calculated by breaking the chain at index k (i<=k<=j), and reducing the problem to the optimums of the left part (m(i,k)) and the right part (m(k+1,j)). Optimizing means to find the k index when the derived sum is the cheapest. This is described by the following equation:

$$m[i,j] = min_{i<=k<j}(m[i,k] + m[k+1,j] + row(i) * column(k) * column(j))$$

Here rows and columns denote the horizontal and vertical size of the i-th matrix in the chain. The algorithm uses another matrix $c(i,j)$, to store the index of optimal parenthesization in the i to j subchain. This is also called cutting matrix.

If we try to program the recursive equation above by a recursive computer program, then the solution will follow the "divide et impera" principle. This is also the case for Prolog predicates. We have already hinted that the Contralog solution turns the original backward-chaining interpretation of Prolog to the opposite: data items, as already proven facts, are stored in dynamic predicates, and implications are interpreted in their natural way: from precondition to conclusion. We expect Contralog to allow the programs to be as simple, as the equation above, while it can also manage all the technical details of forward-chaining and/or dynamic programming.

The $m(i,j)$ and $c(i,j)$ matrices are stored in a single Contralog predicate `matrix(I,J,M,C)`. Parameters I and J are matrix indices, while M and C are the values of the optimum and cutting matrices.

At the start of the algorithm, the $m(i,i) = 0$ initializations are performed by the following Contralog code.

```
size(6).
matrix(SIZE,SIZE,0,0):-
  size(SIZ), {SIZE is SIZ-1}.
matrix(I,I,X,C):-
  matrix(I0,I0,X,C), {I is I0-1, I>=0}.
```

The role of asserting the fact `size(6)` is to launch an inference burst out. This, in the first step asserts only the `matrix(0,0,0,0)` fact, but in the following steps all consecutive $m(i,i)$ values are generated in a cycle. Filling up the lowest and simplest layer (the $m(i,i) = 0$ values) itself is enough to start inference yielding the optimum for more complicated cases (longer matrix chains).

This is done by help of the following Contralog clause.

```
matrix(I,J,X,J1):-
  matrix(I,J1,Y,_), matrix(I1,J,Z,_),
  {I1 =:= J1+1, mxyz(I,I1,J,MULT),
  X is Y+Z+MULT,
    (matrix(I,J,X0,_)->X<X0;
    true)}.
```

The curly bracketed part of the clause is a direct Prolog call. We use this to perform simple arithmetic operations. The Prolog call `mxyz(I,K,J,M)` calculates the number of scalar multiplications necessary to multiply the result of the optimized (i,k) and (k+1,c) matrix subchain products so, that $M = row(i) * column(k) * column(j)$ holds.

The predicate says: if there are two consecutive subchains, then a scalar multiplication number can be calculated for their concatenated chain. Then, according to the principle of gradual approximation, if we have found a value for the concatenated chain that is cheaper than we stored until now, then the more expensive value is replaced by the cheaper one.

The replacement of old optimum value with the new, cheaper one is solved by Contralog's `:-key(matrix(+,+,-,-)).` declaration. This declares the first two parameters (matrix indices) as keys, i.e. for a given (i,j) index pair only a single clause is allowed. The actual retracting of the old clause and asserting of the new one is done by the generated Prolog code in the background.

The Contralog program can be started by calling `:-goal.` The program will fail, which means there is no goal in the program, and there is no other, untried way for inference either. The result is the generated content of the `matrix/4` predicate. To display this, we must implement a simple cyclic procedure. Taking the example in [1], we get the following result. The first triangle is that of optimums, the latter is the cutting matrix. For a (30x35, 35x15, 15x5, 5x10, 10x20, 20x25) matrix chain, the total number of necessary multiplications can be read in the top-leftmost corner.

```
[15125,10500,5375,3500,5000,0]
[11875,7125,2500,1000,0]
[9375,4375,750,0]
[7875,2625,0]
[15750,0]
[0]
[2,2,2,4,4,0]
```

```
[2,2,2,3,0]
[2,2,2,0]
[0,1,0]
[0,0]
[0]
```

When examining the actual order of `matrix/4` facts, we can observe another interesting feature of Contralog. Namely: the inference is eager, or depth-first, like Prolog itself. An implication fires immediately as soon as the last precondition item has arrived. Contralog implements therefore a bottom-up (data-driven) and depth-first strategy for discovering the resolution graph of the problem. In our case this means that the lowest layer ($m(i,i)$ elements) could not have been generated yet, when the first inference steps are already done. This is because the first rule application to calculate the optimum for a chain of two matrices, (the 3-rd `matrix/4` clause) is performed as soon as the first two consecutive matrix-subchains ($m(5,5)$ and $m(4,4)$) are already generated.

3.3 Dynamic programming: Floyd-Warshall algorithm

Similarly to matrix chain multiplication, the problem of shortest paths in weighted graphs can also be solved quicker by dynamic programming approach than by recoursive programming. According to this principle we first give the recoursive definition of the problem, and then we create the sequential program to deliver partial results in a bottom-up manner. In the following example we shall show that if we transcribe the recoursive definition to Horn-clauses – or to Contralog, then Contralog's execution mechanism automatically generates the results of dynamic programming.

In the recoursive definition of the problem d_{ij}^{k} denotes the shortest path between vertices i-j so, that intermediate vertices can be chosen only from the first k vertices of the graph (the actual ordering of vertices is irrelevant). The trivial alternative of the recoursion is case $k = 0$, when no intermediate vertices may be used. In such a case the shortest path is equivalent to the corresponding i-j item of the graph's weight matrix. Otherwise the shortest path using at longest the first k vertices, is equal to the shortest path using k-1 vertices, or it contains vertex k.

$d_{ij}^{0} = w_{ij}$

$d_{ij}^{k} = min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$

The Horn-clause (Contralog) transcription of the definition above is the following:

```
wm(0,I,J,W,SIZE):- w(LL),
 {length(LL,SIZE),between(1,SIZE,I),
 nth1(I,LL,L), between(1,SIZE,J),
 nth1(J,L,W)}.
wm(K,I,J,W,S):- wm(K1,I,J,K1IJ,S),
 wm(K1,I,K,K1IK,S),wm(K1,K,J,K1KJ,S),
 {K is K1+1, K1=<S},
 {W is min(K1IJ, K1IK+K1KJ)}.
```

Parameters K,I and J in definition `wm/5` are obvious. Here, W means the functions value d_{ij}^{k}, while SIZE is the size of the graph (the number of its vertices). This parameter is passed along in order not be calculated it again and again at each step. Definition `wm/5` is the three dimensional matrix itself that consists of dynamic facts. Instead of a sequential cycle, the Contralog execution it is built up by the forward chaining inference mechanism.

The clue of the algorithm lies in the second statement. The three `wm/5` conditions refer to the three d_{ij}^{k} values in the arguments of `min` function. Subsequent Prolog calls are calculating the index-relations, they stop the cycle or perform the actual calculation of minimal value.

The Contralog program, introduced above, implements the evaluation strategy of dynamic programming perfectly with one difference. Namely: partial results are not produced layer-by-layer, not in a breadth-first, but rather in a depth-first way.

Definition `wm/1` contains the weight matrix in a single fact. The first statement of three `wm/5` produces the first element of the 0-th layer first, and upon backtracking it produces also the rest. After the production of each element the second statement of three `wm/5` also starts, and if there are elements in the previous layer for which the preconditions are met, then it also produces an element from the next layer. This may also produce an element of the second next layer and so on

Depth-first execution means, that one corner of `wm/5`, the three dimensional matrix is built up as high is possible, and only in case of no further steps, we backtrack and try to build the lower layers forward. But any element is inserted in a lower layer, building up the higher layers is immediately tried.

4 Applying Contralog for ReALIS natural language parsing

The basic principles of ReALIS (Reciprocal And Lifelong Interpretation System) research project, targeting natural language processing techniques, is described in many conference articles, and even in a book [6, 4]. From the aspect of a Contralog natural language parser, its most important principles are the following:

- *Total lexicality*: each kind of lingual information is stored in the lexicon (in the vocabulary) [5]. Lexical items basically follow the feature structured method, whereas each item may demand certain other lingual context, and it also may offer certain services. Parsing, according to this approach, means the exact discovering and unveiling the offer-demand relationship. This also means: there is no special repository for lingual rules; lexical items describe all relevant information to perform parsing and/or generating natural language texts.

- *Modal logical framework*: interpreters are modelled in the world, and worlds are modelled in the brain of interpreters. The world-model in the background is a hierarchical structure of world-contexts. The root-world corresponds to the objective, outside world, that contains objects, and also contains subjects, i.e. agents, being able to interpret sentences and to store their own world-model. The content of interpreters internal world-model may also contradict the root world model. On the other hand interpreters store an own internal world structure – different modal contexts are stored in a different worldlets (for seen, heard, read, believed information, etc.) The internal world model of interpreters is empty when born, and the content of their world model gradually increases during their life – not necessary monotonically, certain information pieces may also be mistaken, and can later be corrected or even erased.

- *Discourse representation theory*, integrated in the interpreter's world model. Parsing is not restricted to a single sentence, but is extended to all the sentences of a *discourse* (several sentences in the same thematic or situational context). These sentences are usually also in a certain relationship with each other (e.g. antecedent, consequent, argument, etc.). These are called *rhetoric relations*. In addition to simply taking over similar structures of earlier discourse-representation schools [7], we improve them slightly. First, they are not necessarily independent, so we are using a minimal/canonical set of rhetoric relations. Second they are not necessarily objective hence they are not stored in the objective root world, but in the interpreter's subjective world model.

According to ReALIS, instead of calculating a parse tree, the primary goal of parsing is to produce the following four mathematical relations:

- α: the *entity anchoring relation* is an equivalence relation that connects equivalent entities in a discourse.
- κ: the *cursor cluster* that describes global contextual information (Here, Now, Ego, There, Then, You, etc.). Some of its elements (place, time, influence) act as cursors; i.e. in a real discourse they are automatically advanced sentence by sentence.
- λ: the *level relation* that relates rhetorical contexts and/or modal logic relationships [7]. The level structure shows a self-embedding nature, and it also specifies the availability scope of discourse object references.
- σ: the *eventuality relation*, that maps lexical items to logical expression fragments, and finally maps sentences and/or larger textual units to complete logical expressions.

Total lexicality means that beyond mere textual elements (words and/or morphemes), lexical items also contain their eventuality function (a logical expression), instructions for entity anchoring, and the rhetorical and/or modal anchoring instructions. Furthermore a very important piece of a lexical item is its *offer-demand relationship*. This is used to describe morphological and/or syntactical bindings, along with the strength and the direction of the bindings. According to these principles, parsing is nothing more than a sort of domino-game; each syntactical element has certain offers, and may demand certain other elements in the neighborhood. The calculation of the relations mentioned above is done by the underlying unification-based means of finding matching demands and offers.

The mentioned parsing strategy of ReALIS is suitable for any languages, but in practice we recommend it especially for languages, like Hungarian, with variable or free word order. The application of traditional parsing methods for free word order languages produce inefficient results because of the frequent need of backtracking.

4.1 Prolog target models

The most obvious and usual target model for designing Prolog programs is the *relational target model.* According to this, the program calculates the <input,output> relation, which, if we program carefully, enables calculating the relationship in both directions. That is, for a parsing project, the same program can calculate the parse tree for a sentence, and may also calculate the sentence for a given parse tree at the same time.

Relational target model, on the other hand, performs a depth-first search on the resolution graph that depicts the inference process. The efficiency of backtracking programs can be strongly reduced by the frequency and the depth of backtracking, and we guess, natural language parsing may well involve deep and frequent backtracking.

Instead of the relational target model we propose the *inferentional target model* for natural language parsing. Instead of the <input,output> relation, the inferentional model tries to calculate the input $\rightarrow$ output implication. That is, the program must be reformulated so that it can prove that output data in some sense is the logical consequence of input data.

Both for relational and inferentional target models a key expression is *non-determinism.* That means: Prolog programs in general may deliver more than a single equivalent result. The set of results are propagating further on, while other constraints in the calling programs may filter out certain non-applicable results, and in a fortunate situation the end-result is already definite.

Prolog programs, in general, implement a *deductive inference model.* Deductive inference means that, for a given set of basic facts, the set of logical consequences is calculated. From this point of view the direction of inference (forward or backward chaining) is completely irrelevant.

The other strategy is called *abductive inference model.* For abductive inference we are aware of the consequences and the basic rules for inference, and we are searching for the possible facts, based on which the consequences can be inferred.

Though Prolog programs are basically implementing deduction, with a very easy extension we can also apply them for abduction. In case of abduction usually not the entire set, but only a subset of possible facts is unknown. To implement abduction in Prolog, instead of programming fixed set of facts, an implementation of *backtrackable assert operation* is to be programmed, that asserts each possible value for a given fact in a backtrackable way, and at last it retracts the fact completely. This, for the first run asserts certain facts. These may be deleted (and/or reasserted with other values) upon backtracking. The simplest implementation of backtrackable asserts is the following:

```
assertb(FACT):-
  (assert(FACT);
  retract(FACT), fail).
```

Speaking about inference, its *strategy* can be crucial. We have already mentioned: in a number of cases Prolog's *backward-chaining* is not efficient enough. In case of *forward-chaining*, inference rule application burst-outs (inference chains) are started in a data-driven way, upon the arrival of certain facts. They may however arrive at any time, in an asynchronous way; delayed or even in a changed order. One inference step is performed when each precondition holds, and the corresponding facts are accessible. Although it is possible to prune the branches of the inference tree, consequences are produced in their entire richness, but if any of them matches any goal, then the program stops.

One obvious advantage of forward chaining is that already proven facts are stored in the blackboard, and they may be referred later on, for any times.

4.2 Inferential target model for parsing

If we apply the inferential target model to lingual parsing, the input sentence must be stored in a series of facts. Program clauses can be derived from the offer-demand relation of lexical elements, and certain general goals are the constructs to stop (or to start?) the inference.

As a drawback, the model cannot be used for generating text.

When performing ReALIS parsing, it is practical to define the following cuts (layers) over the entire inference graph.

1. The *layer of morphological analysis.* The character stream on the input channel is packed to words by the lexical parser. The stream of words is parsed by a morphological parser. Although ReALIS also has a solution for morphology [5], for convenience we propose to use a commercial solution. The result of morphological parsing is a Contralog sequence of facts. To focus on syntactical parsing, in the following example we omit to deal with the problem of morphological analysis. Let's see the facts resulting from the following Hungarian sentence: ”Petra vágyik arra a magas német úszóbajnokra” (”Petra desires-3SG that the tall German

swimming champion-SUB") [6].

```
word(petra,1,1,noun('Petra',proper,nom,sing-3)).
word(petra,1,2,verb('vágy',[],decl, pres, sing-3)).
word(petra,1,3,noun('az',pro(point),sub,sing-3)).
word(petra,1,4,art(def,cons)).
word(petra,1,5,adj('magas')).
word(petra,1,6,adj('német')).
word(petra,1,7,adj('úszó')).
word(petra,1,7,noun('bajnok',common,sub,sing-3)).
```

The arguments of the facts above are the following: 1. a discourse identifier 2. sentence index in the discourse 3. word index in the sentence 4. a Prolog structure describing the result of morphological analysis.

2. The *layer of grammatical dependency relations*. We are calculating the regent-argument (bidirectional) and adjunct-argument- (one directional) relations. The example below shows the regent-argument description of verb "vágyik" (desires). Hungarian verb "vágyik" demands a nominative argument (the subject), and a sublative argument (the object). The strength of the former binding is -7, of the latter is +7. The former annotation (-7) means, the subject may appear well before the verb in a loose distance. The other means the opposite: the object may appear after the verb, and arbitrary other words may appear between them. Both arguments should form a generalized quantifier determinant structure (proper noun, determinate article, adjective, etc.).

```
regArg2(ID,S,XV,verb('vágy',[],MODE,VTIME,AGR),
      XS,noun(SUBJ,SKIND,nom,AGR),-7,
      XO,noun(OBJ,OKIND,sub,OAGR),7):-
  verb(ID,S,XV,'vgy',[],MODE,VTIME,AGR),
  gqdet(ID,S,XS,SUBJ,SKIND,nom,AGR),order(XV,XS,-7,nei),
  gqdet(ID,S,XO,OBJ,OKIND,sub,OAGR),order(XV,XO,7,nei).
```

3. The layer of *eventuality relations*. In this layer the logical form of regent-adjunct structures is built up in a way, that their arguments logical form is supposed to be built up before. In the example below the predicate `regArg2` builds up the grammatical structure (the regent-argument relation), while `sigma3` prepares the overall logical expression as a result.

(`=../2` is a technical Prolog call that transforms time referent structure to grammatical time notation.)

```
sigma3(ID,S,XV,TIME,SUB,OB,CLAUSE):-
    regArg2(ID,S,XV,verb('vágy',[],MODE,VTIME,_AGR),
                 XS,SUBJ,_PRS,XO,OBJ,_PRO),
    TIME =.. [VTIME,_],
    sigma3(ID,S,XS,TIME,SUB,CLAUSE,
               (desire(TIME,SUB,OB):-CONS)),
    sigma3(ID,S,XO,TIME,OB,CONS).
```

As the logical form of the sentence above, we may get the following clause. (The double implication can be transformed easily into a conjunction on the precondition side)

```
CLAUSE=((desire(pres(T),SUB,OB) :- swim(T, OB),german(T, OB),
        tall(T, OB),champion(T, OB)) :- name(T,SUB,'Petra'))
```

4. The *layer of rhetorical and modal relations*. In this layer, rhetorical and modal logical relations are built up as described by the λ function. Although there have been theoretical investigations completed [8], up to the writing of the article we don't have any concrete results to demonstrate this.

5 Program development with Contralog

Contralog is available as a preprocessor for SWI-Prolog [9] that works on the basis of its macro extension mechanism (`term_expansion/2`) . This is unfortunately only de facto standard, therefore the seamless operation with other Prolog dialects is not guaranteed.

When developing Contralog programs, the Prolog module, defining the above mentioned macro expansion predicate, must be consulted first. (`clog.pl`). It is not enough only to load `clog.pl` in the application module, because in the time of reading the first (module head) declaration of the module, the Contralog pretranslator must already be active. The module declaration itself is handled otherwise by Prolog, even its export list is understood as Prolog exports.

To declare Contralog export, `:- export(EXPLIST).` declaration should be used.

In general, Contralog program clauses are to be placed in a Prolog program. To switch between normal Prolog clauses, and preprocessed Contralog clauses, the following two new directives can be used.

`:- contra.` that starts the processing of Contralog code

`:- pro.` that switches back to normal Prolog

Beside the Contralog to Prolog translator, at the moment there is no other development tool available. It is a bit clumsy, for example, to debug Contralog programs. Since there is no debugger, debugging is possible only by help of the Prolog debugger, and it may work only for those, who are familiar with the target model.

6 Summary and future work

We have defined Contralog, which is a Prolog-conform language, but instead of backward chaining, it uses forward-chaining inference. The language itself is built-up on the same syntax: Horn clauses, only its declaration forms are different from those in Prolog.

For this end we have developed a Prolog target model to perform forward chaining inference. This enables to translate Contralog clauses directly to Prolog, and execute them by the Prolog software environment.

The Contralog to Prolog translator has been implemented by using Prolog's macro extension mechanism. This approach, and the Prolog target model itself enables various possibilities to integrate forward and backward chaining inference in a programmer-controlled way.

To demonstrate these, the article introduces several examples: after the simplest examples two dynamic programming problem and ReALIS natural language parsing mechanism is described.

Future work may include extended Contralog program development possibilities and tools.

The natural language parsing mechanism has been tested only for a handful of sentences by translating the lingual information to Prolog (Contralog) manually. Any future work must aim to collect lingual information, the mechanical translation of these to the Contralog target model, and the development of the overall ReALIS parsing environment.

Acknowledgements

The author is grateful for the Hungarian Research Fund TÁMOP 4.2.2.C-11/1/KONV-2012-0005 (Well-being in the informational society) for the support of his research activities regarding natural language technologies, and also for the Hungarian Research Fund TÁMOP 4.2.2.D. to cover the Mathinfo2015 conference participation expenses. Special thanks to colleagues: Gábor Alberti and Márton Károly for their kind remarks and contribution.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (3rd edition), The MIT Press, 2009. ⇒50, 52

[2] W. F. Clockshin, C. S. Mellish, *Programming in Prolog*, Springer Verlag Berlin, Heiderlberg, New York, 1994. ⇒42

[3] G. Alberti, *ReALIS: Interpretators in the world, worlds in the interpretator* (in Hungarian: ReALIS. Interpretálók a világban, világok az interpretálóban). Akadémiai Kiadó, Budapest, 2011. ⇒55, 59

[4] G. Alberti, *ReALIS. An interpretation system which is reciprocal and lifelong.* Akadémiai Kiadó, Budapest, 2011. ⇒55

[5] G. Alberti, K. Balogh, J. Kleiber, A. Viszket, Total lexicalism and GASGrammars: A direct way to semantics *Proc. CICLing2003*, NLCS 2588, pp. 37–48, (ed Gelbukh, A.) Springer Verlag, Berlin 2003. ⇒55, 58

[6] G. Alberti, I. Kilián, Bipolar influence-chain families instead of lists of argument frames – the sigma function of ReALIS (In Hungarian: Vonzatkeretlisták helyett polaritásos hatáslánccsaládok) *Proc. MSzNyVII. Hungarian Conference of Computer Linguistics*, pp. 113–126, (ed: A. Tanács, V. Vincze) VII. Magyar Számítógépes Nyelvészeti Konferencia, MSzNy pp. 113-126, SzTE Informatikai Tanszékcsoport, Szeged. 2010. ⇒55, 59

[7] H. Kamp, J. van Genabith, U. Reyle, Discourse representation theory. *Handbook of Philosophical Logic*, Vol. 15., 125394. Springer Verlag, Berlin, 2011. ⇒55, 56

[8] M. Károly, Interpretation and modality - towards the implementation of ReALIS' λ-function. (In Hungarian: Interpretáció és modalitás avagy a ReALIS λ-fūggvényének implementációja felé.) (ed. V. Vincze, A. Tanács) VIII. Magyar Számítógépes Nyelvészeti Konferencia, MSzNy 284–296. SzTE Informatikai Tanszékcsoport, Szeged. 2011. ⇒60

[9] J. Wielemaker: An overview of the SWI-Prolog programming environment, *Proc. 13-th International Workshop on Logic Programming Environments*, ed: F. Mesnard, A. Serebenik, Katholieke Universiteit Leuven, Belgium, 2003. 1–16 pp. ⇒60

Received: February 9, 2016 • Revised: April 16, 2016