

# Towards HPSG-based Concurrent Machine Translation via Oz

Liviu-Virgil Ciortuz  
LIFL, University of Lille I – France  
and  
University of Iasi – Romania

*Abstract*— We introduce **CHALLENGER 2**, a demonstrative project in the field of concurrent Natural Language Processing/Machine Translation. This paper gives an overview on the purposed goal for this project, the linguistic model involved, its computational support and current state of work. An interpreter and a compiler written in **DFKI Oz** for **DFL** – a very expressive (kernel) language for HPSGs implementation, built on top of a feature constraint system [5] using F-logic semantics [7] – are briefly presented and evaluated.

## 1. INTRODUCTION: PROJECT PRESENTATION

Automate Machine Translation (MT) is for some of us not a dream. It is (or it may be) just a task to be accomplished.<sup>1</sup> Doing it simply one have to choose

- the suitable linguistic model, and
- the supporting programming paradigm and language.

The **CHALLENGER 2** project has decided respectively for

- HPSG – the most used frame-based descriptive theory of language, and
- Oz – the first truly multi-paradigm programming language, underlied by a comprehensive theoretical model.<sup>2</sup>

Somehow hidden in between, making in fact the “fix-point” of the current state of work in our project is DFL, a

Liviu-Virgil Ciortuz is supported by a doctoral grant from the French Ministry of Foreign Affairs. Address: Université de Lille I, LIFL, Bât. M3. 59655, Villeneuve d’Ascq, Cedex. E-mail:ciortuz@lifl.fr.

This is a revised and extended version of the paper “Towards English into Romanian Translation via Oz” presented and published at the WOZ’95, the first International Workshop on Oz Programming, held at Martigny, Switzerland in November, 1995.

<sup>1</sup>The European Community conditions the acceptance of East European partners like Romania with (among other issues) the development of competitive MT systems from/into the Community languages. While a team of Computational Linguistics researchers at the Romanian Academy develops such a *real* project, **CHALLENGER 2** is a personal demonstrative project focusing on evaluating the benefits of new concurrent constraint-based multi-paradigm programming techniques in Natural Language Processing. It is the successor of **CHALLENGER 1** reported by [4].

<sup>2</sup>There are a number of Prolog-based systems doing HPSG implementation. The most important are ALE (Attribute Logic Engine) by Bob Carpenter and Gerald Penn at Carnegie-Mellon University and HPSG-PL by Fred Popowich, Snadi Kodric and Carl Vogel at Simon Fraser University of Canada[11].

Now, with the emergence of feature-constraint logic systems, it is interesting too see how new languages built on top of them, like Oz developed at DFKI - Saarbrücken, can support real NLP applications. In a message put on the Oz users list, Gert Smolka wrote: “I don’t know of any grammar formalism that has comparable abstraction power.” While he implements as a demo only a particular HPSG, we aim to do a more general work, namely interpreting and compiling HPSGs into Oz.

logic Data Frame-based Language for HPSGs implementation. DFL is built on top of a constraint system remaking the F-logic operational semantics [7]. It offers an expressive way to reason on frame data structures extending in some particular respects the Ait-Kaci’s  $\psi$ -terms [1] and Smolka and Treinen’ logic records [12]. An interpreter for DFL has already been written in Oz, and a compiler translating DFL logic programs to Oz is currently implemented at LIFL, France.

The second section of this paper presents the link between HPSGs and DFL, while the third one links DFL to Oz making (hope) some interesting points with respect to the above mentioned implementations. The fourth section gives a first simple example of automate translation from English into French and Romanian.

The purposed goal of this project is to do demonstrative concurrent Machine Translation from English into Romanian and French.

## 2. FROM HPSGs TO DFL

Head-driven Phrase Structure Grammars [10] deal with *signs*, feature-based constructs usually called frames. They express in a unitary form both linguistic data and relations between these data (“principles”). For instance, the simple HPSG grammar implemented by one of the DFKI Oz demonstrative programs can be visualized as the frame in Figure 2.<sup>3</sup>

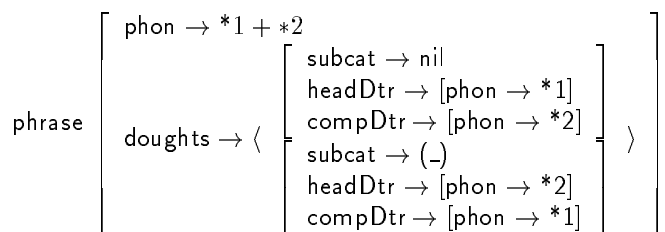


Fig. 1. A HPSG sign

OSF was the first constraint system conceived to reason on order-sorted feature constructs [2]. It conceptualizes LIFE, an extension of Prolog with (unification on)  $\psi$ -terms over an apriori given sort hierarchy [1]. Then OSF was followed by CFT, the base constraint system in Oz [3]. Making use of the terminology in [7], we can say that while  $\psi$ -

<sup>3</sup>The  $*n$  marks are tags,  $+$  denotes string concatenation, the  $\langle \rangle$  brackets enclose different values for multiple-valued features,  $\_$  is the anonymous tag, and  $()$  enclose (maybe empty) ordered sequences.

terms have (only) first-order typing features, logic records in CFT have (only) functional features.<sup>4</sup>

F-logic, a new foundation for frame-based and object-oriented languages [7] introduces F-terms, constructs with both functional and typing features. The so-called Well-Typing Condition Principle links function values to the corresponding types, possibly involving other principles like Type Inheritance, Argument Sub-Typing and Range Super-Typing. F-logic allows F-terms to contain multi-valued features and a higher-order syntax. So, object labels and features (with arguments, values and types) can be represented by Prolog terms.<sup>5</sup> F-logic completeness w.r.t. a first-order semantics has been proven. We argue that:

- F-terms can serve to adequately represent HPSGs, and
- Reasoning on them could be done quite nice in the functional object-oriented concurrent constraint language Oz.

For the first point, we would consider the simple task to write a program in F-logic doing parsing on the above given HPSG grammar. The program in Figure 2 does this work similarly to the previously referred DFKI Oz demonstrative program.<sup>6</sup> The first two clauses, C1 and C1', impose that rules' application results are phrases (i.e., two signs L and R should always combine within a sign), and then give the sign's feature structure, according to which one, either R or L, is the head of the sign. The next two clauses, C2 and C2', condition building a phrase by three principles satisfaction: Subcategorization principle<sup>7</sup>, Saturated complements<sup>8</sup>, and Head-feature principle<sup>9</sup>. Clauses C3 and C4 carry on lexical recognition and launch parsing. The intermediary parsing function (`parser1`, defined by C5 and C5') implies application of grammar rules (by using the `move` function, see clauses C6 and C6'). Finally, the clause C7 prepares for introduction of lexical entries.

The second point stated above makes the central subject of the next section.

DFL is a (constraint) logic language doing Prolog-like inferences on Horn clauses over F-terms. It allows one to reason on a partially known/specified sort hierarchy. The sort hierarchy can be concurrently completed during goal solving. A bidirectional suspend and resume mechanism relates the goal solver to the hierarchy completer. The feature constraint system underlying DFL was presented in [5].

<sup>4</sup>Variables in Oz allow one to reason on features as higher-order constructs.

<sup>5</sup>For efficiency and also sufficiency reasons, we use only Datalog terms.

<sup>6</sup>In this example, the F-logic syntax was enhanced with the well-known selection operator '.' in object-oriented languages. It is assumed left-associative. The '|' operator corresponds to list construction, and it is right associative. '|' and '+' have lower priority than '.', and '@' has greater priority than '.'.

<sup>7</sup>Head's subcategory is a 'cons' made of 1. the category of the complement daughter of the phrase and 2. the phrase subcategory.

<sup>8</sup>The subcategory of the complement daughter of the phrase should be empty.

<sup>9</sup>The phrase category is the category of its head.

```

C1:  L[rule@R ==>> phrase].
C1': L[rule@R ->> [phon -> L.phon+R.phon
                subcat -> nil
                headDtr -> R
                compDtr -> L],
     [phon -> L.phon+R.phon
     subcat -> (-)
     headDtr -> L
     com Dtr -> R]].

C2:  P:phrase:- P[headDtr -> [subcat -> P.compDtr.cat |
                             P.subcat]
                compDtr.subcat -> nil].
C2': P:phrase[cat -> P.headDtr.cat].
C2'': P:phrase[gender -> P.headDtr.gender].

C3:  Phon[parse -> Phon.map@f.parse1].
C4:  W[f -> F] :- F.word[phon -> (W)].

C5:  (F)[parse1 -> F].
C5': Fs [parse1 -> Fs.move.parse1].

C6:  F|G|Fr[move -> X|Fr] :- [F.rule@G ->> X].
C6': F|T [move -> F|T.move].

C7:  W:word[phon -> (P)
        cat -> C
        subcat -> S] :- W[C@P,S].

```

Fig. 2. A F-logic program for HPSG parsing

### 3. FROM DFL TO OZ

#### 3.1 Interpreting DFL in OZ

The CHALLENGER 2 interpreter for DFL/HPSGs is a concurrent resolution-based (i.e., Prolog-like) engine. In fact it could work for any pure logic language assuming that a suitable unification method on its simple expressions – terms and/or atoms – is rewritten via class derivation. The engine implementation is based mainly on the Procrastination Principle [9]. Full concurrent execution of a goal can be achieved by simply using parallel conditional case in the procedure doing clause selection for goal matching.

Technically speaking, we have to switch from the lazy list used for goal representation to a lazy tree. But this is a rather a conceptual changing, since concurrency in Oz makes it fully transparent to the programmer up to local substitution saving for each newly introduced goal sublist in the lazy list/tree.

The constraint concurrent capacity of Oz enables the DFL interpreter to work with partially specified (at syntax level) sort hierarchies and to complete them in parallel with goal solving, by applying the Well-Typing Conditioning principle. Suspension and resumption is synchronized in conjunction to sort hierarchy completion.

New “is-a” sort relationships can be concurrently entered in the hierarchy by

- reasoning on the DFL program representation, and (possibly)
- checking the entailment of corresponding conditions in the clause bodies.

```

(D1) X[happy] :- X:person[friend -> Y].
(D2) person[friend => person].
(D3) Z[F -> W] :- W[F:symmetric -> Z].
(D4) friend:symmetric.
(D5) albert:person.
(D6) albert[friend -> lucy].
    
```

Fig. 3. A simple DFL logic program

For example, given the simple DFL program in the Figure 3, the execution of the goal *lucy[happy]* will suspend until the completion procedure will “close” the program by adding the clause

```
(D7) lucy:person.
```

due to Type Inheritance and Well-Typing Conditioning between the (D2) and (D6) clauses.

Order-sorted unification to be implemented in the near future is expected to increase the DFL interpreter efficiency. It will be possible thus to extend Peter Van Roy’s idea for implementing  $\psi$ -term unification - using finite domain constraints [8] - to concurrent specification of domains.

The DFL interpreter highly exploits Oz concurrency. Statefull variables acting as unit substitutions were defined and then used for unification, thus dis-mounting monotonic constraint handling on Oz (stateless) variables. There are two versions of stateful variables implemented in the DFL interpreter: one involves object-orientation, the other generates variables as concurrent agents. Sequentialization control in this last version is under study. Frame unification is in progress at the time of writing this paper.

The Oz code in Figure 4 is a simplified version of the main function in the DFL interpreter.

Full object-oriented capacity of Oz will be highly addressed in the DFL into Oz compiler.

### 3.2 Compiling DFL into Oz

Due to objective limitations, the strength of this subsection is not on the HPSG/DFL into Oz compiling program. We have chosen instead to show how compiled DFL programs look like when translated into Oz programs. Let us consider for instance the translation of the priorly given DFL program into Oz, given in figures 5 and 6.

We can set up the following points guiding the compilation work:

i. Ground terms on *identification* ( $a[...]$ ) or *value/type* ( $-\dots \rightarrow v$ ) position in the DFL program translate into Oz classes. See for example the classes **Person** and **Albert** corresponding respectively to **person** and **albert**. They will be further addressed through message passing. This point applies also for **Top**, the highest class in the is-a hierarchy.

ii. Other ground terms which appear only in the (*super*)*class* position of is-a terms ( $-\dots : c$ ) translate into Oz atoms. This is the case of **symmetric** and **friend**.

```

fun{Query Prog Goal}
  case {ToEnd Goal} then True
  else
    HG = {Car Goal} FC = {GetFirstCl Goal}
    AS = {GetSub FC} OS OG
  in
    {Goal saveTo(OG)} {AS saveTo(OS)}
    {ForSomeB Prog}
    fun{$ ClauseL}
      {ClauseL renew} HC = {First {GetClause ClauseL}}
    in
      case {IUnify HG#(AS.list) HC#({GetSub ClauseL}.list)}
      then
        NewClauseL = {New Cl
          init({Tail {GetClause ClauseL}} {GetSub ClauseL})}
        NewGoal = {New LazyList init(NewClauseL Goal)}
      in
        case {Query Prog NewGoal} then True
        else {AS restoreFrom(OS)}
          {Goal restoreFrom(OG)} False
        end
        else {AS restoreFrom(OS)} False end
      end
    end
  end
end
    
```

Fig. 4. The engine of the DFL interpreter

iii. Is-a relationships in the DFL program correspond (in part) to class derivations in Oz. **Top** is derived from **UrObject**; the other compiled classes are derived from **Top**. These relationships together with the remaining others (like **friend:symmetric**) are stored in the global signature defined by the list **lsA**. The **{SubType X}** and **{TypeB O C}** functions return respectively the list of all descendents of **X**, and **True** (respectively **False**) if **O** is (is not) derived from **C** w.r.t. **lsA**.

iv. Definite ground constraints in DFL translate into Oz features in the case of functional constraints.<sup>10</sup> The “translated” features are unary functions. They will be applied to classes sending messages via a *Query&High* mechanism.

v. Higher-order functional features in DFL translate into **high** methods, one for each class, when required. See for example the **high** method in the **Top** class. A **high** method in a class **C** is a **case** construct selecting one function for each second-order DFL feature of the sort **C**. The higher-order protocol in DFL is “reflective” in the sense of [6]. The needed occur-check is implemented via the couple **checkIn/checkOut** methods in **Top**.

vi. Goal evaluation (or: entailment for existentially closed constraints in DFL) is managed by the **Query** function. If no homonymous feature can be found (via the inheritance mechanism of Oz), then explicit access to a higher-order function working for that feature is demanded. This is done by invoking the **High** function.

<sup>10</sup>Typing constraints in the DFL program are object of precompilation.

```

create Top from UrObject
% general attr + methods
attr check: nil
meth checkIn(X ?B)
  case {MemberB X @check} then
    <<checkOut(X)>> B = False
  else check <- X|@check B = True end
end
meth checkOut(X)
  check <- {FilterB @check fun {$ Y} X\=Y end}
end
meth high(HOF ?F)
% specific (compiled) methods
case {TypeB HOF symmetric} then
  F=fun {$ X}
    Y in
      case {ForSomeB {FilterB {SubType Top}
        IsInheritableB}
        fun {$ W}
          case {Query W HOF} == X then
            Y = W True
          else False end
        end}
      then Y
      else NoValue end
    end
  else F = NoValue end
end
end
end

```

Fig. 5. The Top class in the *simple* DFL program translated into Oz

Dynamic completion of the sort hierarchy is not (yet) supported in the compiler. The compiler design is under full development.

#### 4. AUTOMATE TRANSLATION: A FIRST EXAMPLE

We resume the program in Figure 2.

Before explaining how translation works, let's see the syntactic analysis result. Solving the goal<sup>11</sup> ?- "the girl is nice"[*parse* -> *Z*]. generates the goal lists in Figure 8. Due to objective space limitations, this list is selective. Failed steps are omitted. It should be also understood that using, for example, the clause C5' is followed by applying C1' and C2. (The last one is demanded by static type checking, see C1.

Note that when solving the G3 goal, the program is getting the *g(the, girl)* as representation<sup>12</sup> for "the girl", with the following sign description:

<sup>11</sup>Here and in the sequel, a string "a b c" will be assimilated with the list of literals 'a'|'b'|'c'|nil.

<sup>12</sup>The functional symbol *g*, with arity 2, is automatically associated by the DF interpreter to the clause C1' to identify – in the sense of F-logic, through Skolemization – the first value of the feature rule. Similarly, *h* is associated to the second value of the same feature.

```

create Person from Top
  feat happy:
    fun {$ X}
      Y = {Query X friend} in
        if Y \= NoValue then True
        else False fi
      end
    end
end
create Albert from Person
  feat friend: fun {$ X} Lucy end
end
create Lucy from Person
end
% the IsA signature
IsA = [Person#Top Albert#Person Lucy#Person
  symmetric#Top friend#symmetric]

```

Fig. 6. Classes in the *simple* DFL program translated into Oz

```

NoValue = {NewName}
fun {High Object Feature}
  F in {Object high(Feature F)} F
end
fun {CheckIn Object Feature}
  B in {Object checkIn(Feature B)}
  B == True
end
fun {Query Object Feature}
  case {MemberB Feature
    {Map {Record.toListInd Object} fun {$ Y#_} Y end}}
  then {Object.Feature Object}
  else
    G = {High Object Feature} in
      case G \= NoValue andthen {CheckIn Object Feature}
      then {G Object}
      else NoValue end
    end
  end
end

```

Fig. 7. The *Query&High* Protocol for DFL program execution

```

g(the, girl)[ phon    -> ('the,' girl')
              subcat  -> nil
              headDtr -> girl[ cat    -> noun
                              gender-> fem]
              compDtr -> the]

```

Similarly for "is nice", at G5, we obtain:

```

h(is, nice)[ phon    -> ('is,' nice')
             subcat  -> (-)
             headDtr -> is[cat -> verb]
             compDtr -> nice]

```

The subcat feature of *h(is, nice)* will be instantiated to (noun), cf. the subcategorization principle, C0, and C1.

And finally, for the whole sentence "the girl is nice", the structure is:

```

G0: 'the'|'girl'|'is'|'nice'|nil [parse -> Z].
(+C3) G1: 'the'|'girl'|'is'|'nice'|nil [map@f -> Z1],
      Z1[parse1 -> Z].
(+C4) G2: the|girl|is|nice|nil [parse1 -> Z].
(+C5') G3: the|girl|is|nice|nil [move -> Z2],
      Z2[parse1 -> Z].
(+C6) G4: g(the.girl)|is|nice|nil [parse1 -> Z].
(+C5') G5: g(the.girl)|is|nice|nil [move -> Z3],
      Z3[parse1 -> Z].
(+C6') G6: g(the.girl)|h(is,nice)|nil [parse1 -> Z].
(+C5') G7: g(the.girl)|h(is,nice)|nil [move -> Z4],
      Z4[parse1 -> Z].
(+C6) G8: g(g(the.girl),h(is,nice))|nil [parse1 -> Z].
(+C5) G9: □

```

Fig. 8. Solving parsing

```

g(g(the.girl),h(is,nice))[ phon    -> ('the','girl','is','nice')
                          subcat  -> nil
                          headDtr -> is[ cat    -> verb
                                      gender-> fem]
                          compDtr-> g(the.girl)]

```

The DF clauses designed to carry on translation are given in Figure 9. C8 translates sentences, while C9 and C10 do a similar job for verb phrases and respectively noun phrases.

```

C8: P:phrase[translate@L -> P.compDtr.translate@L +
      P.headDtr.translate@L] :-
      P[cat -> verb
        compDtr.cat -> noun].
C9: P:phrase[translate@L,Gen->P.headDtr.translate@L]
      P.compDtr.translate@L,Gen|nil] :-
      P[cat -> verb
        compDtr.cat -> adjective].
C10: P:phrase[translate@L -> P.headDtr.translate@L,det] :-
      P[cat -> noun
        compDtr -> the].

```

Fig. 9. Translation clauses

Now, asking for translation

```

?- "the girl is nice" [ translate@fr -> X
                      translate@ro -> Y].

```

the system will resume the precedent goal list, as shown in Figure 10.

```

G9: g(g(the.girl),h(is,nice))|nil [translate@fr -> X].
(+C8) G10: g(the.girl)[translate@fr -> X1],
      h(is,nice)[translate@fr -> X2],
      X1[append@X2 -> X].
(+C10,C9)G11: 'la'|'fille'|nil[append@'est'|'jolie'|nil -> X].
...

```

Fig. 10. Solving translation

So, the French translation is  $X = \text{"la fille est jolie"}$  and, similarly, the Romanian translation:  $Y = \text{"fata este frumoasa"}$ .

## 5. CONCLUSIONS

We implement a nucleus for a demonstrative NLP/MT system. The accent is for now swatched on the computational resource investigation process. Expressivity for both a descriptive language (DFL) for HPSG implementation and the supporting programming environment - Oz - is of the first interest.

In fact, the main interest we set down while starting this project is just to be an open-minded guest into the interesting *land* newly created by the Oz *language* and computational *model*. Otherwise said:

- try to evaluate from different (language and implementation) perspectives the expressivity of its logic records;
- learn to abstract knowledge using its higher-order functionality;
- wisely increase constraints on stateless constructs while highly inheriting on statefull objects;
- let us come over partial knowledge by concurrently demanded computations.

To achieve all these above mentioned goals in the ground of MT became for us a truly CHALLENGER task!

## REFERENCES

- [1] H. Ait-Kaci, A. Podelski, Towards a meaning of LIFE, *Journal of Logic Programming*, 1993:16:195-234.
- [2] H. Ait-Kaci, A. Podelski, G. Smolka, A feature constraint system for logic programming with entailment, *Theoretical Computer Science* 122, pp 263-283, 1994.
- [3] R. Backofen, A complete axiomatization of a theory with feature and sort constraints, *Journal of Logic Programming* 1995:24:37-71.
- [4] L.V. Ciortuz, A. Ignat, CHALLENGER: The first automate translation system from English into Romanian. In: *Proceedings of the 2nd Symposium on Developing and Application Systems*, Suceava, Romania, 1994, pp. 173-174.
- [5] L.V. Ciortuz, DF constraint system. In *Proceeding of the First International Workshop on Concurrent Constraint Programming*, Ph. Codognet (ed.), Venezia, 1995.
- [6] S. Constantini, G. Lazarone, A metalogic programming approach: language, semantics and applications, Technical Report, Universita di Milano, 1990.
- [7] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, *Journal of ACM*, Vol. 42, No. 4, July 1995, pp. 741-843.
- [8] T. Müller, K. Popow, C. Schulte, J. Würtz. Constraint Programming in Oz. In DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123, Saarbrücken, Germany, 1994.
- [9] D. Maier, D.S. Warren, *Computing with Logic*, Benjamin/Cummings, Menlo Park, CA 1988.
- [10] C. Pollard, I. Sag, *An information-based syntax and semantics*, vol. I, CSLI, 1987.
- [11] F. Popowich, C. Vogel, A logic-based implementation of Head-driven Phrase Structure Grammars. In *Natural Language Understanding and Logic Programming*, vol. III, C. Brown and G. Koch (eds.), Elsevier Science Publishers B.V., 1991.
- [12] G. Smolka, R. Treinen, Records for logic programming, *Journal of Logic Programming* 1994:18:229-258.