Remarks on Dependent Types in Syntax

Hans Leiß
leiss@cis.uni-muenchen.de

Retired from: Universität München
Centrum für Informations- und Sprachverarbeitung

5th Grammatical Framework Summer School
Riga, August 14–25, 2017

Where can dependent categories in syntax be useful?

Earlier exercise in GF:

I. Expl. Use dependent arguments to distinguish Sg- and Pl-NPs in abstract syntax. Appl.: *(S -> NP.sg + VP.reciprocal)
`(the children | *the child) like(|*s) each other.`

New experiment in GF:

II. Use categories dependending on arguments that reduce the number of modification possibilities

\+ Deliver much less parse trees in order to obtain a reasonable test suite and less input possibilities for semantic rules

III. Use categories depending on arguments restricting the subject position of verbs and nouns (nominal vs. sentential subject)

But: GF does not use dependent arguments at parse time. So, in analogy to `V,V2,VS`, introduce `NV,SV, NV2,SV2, NVS,SVS` ?

I. Number restrictions in abstract syntax

Using dependent categories in abstract syntax, we can implement

- semantically necessary number restrictions on verb arguments,
- reciprocal pronouns as operations reducing the arity of verbs and imposing a plurality restriction on other arguments
- a difference between mass- and count-nouns on the type level.

⟨*DepReciprocals.gf*⟩≡

```
abstract DepReciprocals = {
  cat Number ; NP Number ;     -- NPs depending on number
  fun sg, pl, any : Number ;   -- any: only in verbs

      John, Mary : NP sg ;
      And : (n,m : Number) -> NP n -> NP m -> NP pl ;
      Or  : (n : Number)   -> NP n -> NP n -> NP n ;
```

Construct NPs from count/mass nouns and suitable determiners:

⟨*DepReciprocals.gf*⟩+≡

```
cat Kind ; CN Kind ;                    -- count vs. mass CNs
fun
  count, mass : Kind ;
  Gold : CN mass ; Man : CN count ; -- example CNs
  UseCNm : CN mass -> NP sg ;    -- NPs without det
  UseCNc : CN count -> NP pl ;

cat Det Kind Number ;                   -- determiners dep.on
fun                                     -- CN-kind and number
  DetCN : (k:Kind) -> (n:Number) ->
    Det k n -> CN k -> NP n ;
  Much  : Det mass sg ;                 -- pl not for mass CNs
  Every : Det count sg ; All,Many : Det count pl ;
  TheSg : (k:Kind) -> Det k sg ;
  ThePl : Det count pl ;
```

Dependent verb types can restrict number for verb arguments:

⟨*DepReciprocals.gf* ⟩+≡

```
cat V1 Number ;  -- 1-ary verbs with argument in n:Numb
    V2 Number Number ;  V3 Number Number Number ;
    VP Number ;
    VPSlash Number Number ;
    S ;
fun
  agree1 : V1 pl ;          -- subject must be pl
  walk1  : V1 any ;         -- subject may be sg or pl
  agree2 : V2 any any ;     -- to agree with sb.
  like2  : V2 any any ;
  mix2   : V2 any pl ;      -- object must be pl
  introduce3 : V3 any any any ;
```

Rem. `any` imposes no restriction. [There is no NP of type `NP any`.]

The number restrictions have to be

- inherited from categories `Vi` to `VPSlash`,`VP`,
- observed when combining an NP with a VP. (`Pred*`, `Compl*`)

⟨*DepReciprocals.gf*⟩+≡

```
UseV1  : (n:Number) -> V1 n -> VP n ;
PredVP : (n:Number) -> NP n -> VP n -> S ; -- agree!
PredVPany : (n:Number) -> NP n -> VP any -> S ;

SlashV2a : (n,m:Number) -> (V2 n m) -> VPSlash n m ;
ComplSlash: (n,m:Number) ->
                  VPSlash n m -> NP m -> VP n ;
ComplSlashAny: (n,m:Number) ->
                  VPSlash n any -> NP m -> VP n ;
ComplV3 : (n,m,l:Number) ->
                  V3 n m any -> NP l -> VPSlash n m ;
```

From predicates of arity $n + 1$, an object reciprocal pronoun builds a predicate of arity n in which the subject (or another object) position must be in plural: *x likes y* $\mapsto$ *(x and y) like each other*

⟨*DepReciprocals.gf*⟩+≡

```
      Reci2any : VPSlash any any -> VP pl ;
                              -- to like each other : VP pl
      Reci2pl  : VPSlash any pl -> VP pl ;
      Reci3 : V3 any any any -> VPSlash any pl ;
         -- to introduce to each other : VPSlash any pl
  }
```

A concrete implementation[1] then generates only sentences where number distinctions, count/mass distinctions, and reciprocals obey the constraints:

[1] `DepReciprocalsEng.gf`, to be added to `GF/contrib` on github

⟨*DepReciprocals.expl*⟩≡

```
DepReciprocals> gr -number=4 -tr | l
PredVPany pl (DetCN count pl Many Man)
          (ComplSlashAny any sg (SlashV2a any any like2)
                                (DetCN mass sg Much Gold))
PredVPany sg John
       (ComplSlashAny any pl (SlashV2a any any agree2)
                             (DetCN count pl ThePl Man))
PredVP pl (And sg sg (DetCN count sg Every Man) Mary)
       (Reci2any (SlashV2a any any like2))
PredVPany sg Mary (ComplSlash any pl (Reci3 introduce3)
                                (DetCN count pl Many Man))


many men like much gold
John agrees with the men
every man and Mary like each other
Mary introduces many men to each other
```

II. Ambiguities with modification rules

Modification rules `Mod: C->M->C` can be applied in different order.
Expl: `(Adv|Adj|Rel)CN, (Adv|Count|Rel)NP, (AdA|Adv|Sent)AP`

⟨*Parsing example, gf-3.8*⟩≡

```
i LangGer.gf
> p "der kleine Junge hier , der schläft , atmet"
...
(PredVP
  (RelNP (DetCN (DetQuant DefArt NumSg)
                (AdvCN
                  (AdjCN (PositA small_A) (UseN boy_N))
                  here_Adv))
         (UseRCl ... (RelVP IdRP (UseV sleep_V))))
  (UseV breathe_V))
```

Gives 24 abstract trees, depending on the history of modifications in AP+CN+ADV+RCl.

Are there that many different meanings?? Almost certainly NO!

- For `LangGer`, one can prove that some different histories of modifications always result in the same concrete record: `ap+cn+adv` as `(ap+cn)+adv` or `ap+(cn+adv)`.
- This also holds for modifications at different levels:

  a) for modification by `RS` at `CN` or `NP`-level:

  ⟨*equal linearization records*⟩≡

```
DetCN det (RelCN cn rs) =
RelNP (DetCN det cn) rs
```

  b) for modification by `Adv` at `VPSlash` or `VP`:

  ⟨*equal linearization records*⟩+≡

```
(AdvVP (ComplSlash vpsl np) adv) =
(ComplSlash (AdvVPSlash vpsl adv) np)
```

This seems to be language independent, i.e. we have "spurious ambiguities by design".

Simplified implementation of binary modifications as in the RGL, with categories as non-dependent types:

⟨*Binmod.gf*⟩≡

```
abstract Binmod = { -- types are *not* depending
cat S; VP; NP; CN; AP; RS; V2;   -- on arguments
fun AdjCN : AP -> CN -> CN ;
    RelCN : CN -> RS -> CN ;
    DetCN : CN -> NP ;
    Compl : V2 -> NP -> VP ;
    Pred : NP -> VP -> S ;
    a : AP ; v : V2 ; r : RS ; n : CN ;
}
```

Iterated postmodification `cn+rs+rs` might be `cn+adv+rs` in reality.

⟨*BinmodGer.gf*⟩≡

```
concrete BinmodGer of Binmod = open Prelude in {
lincat S, VP, NP, CN, AP, RS, V2 = { s:Str } ;
lin AdjCN ap cn = { s = ap.s ++ cn.s } ;
    RelCN cn rs = { s = cn.s ++ rs.s } ;
    DetCN cn    = { s = cn.s } ; -- simplified
    Compl v2 np = { s = v2.s ++ np.s } ;
    Pred np vp  = { s = np.s ++ vp.s } ;
    a = { s = "a" } ;
    r = { s = "r" } ;
    v = { s = "v" } ;
    n = { s = "n" } ;
}
```

This gives 6 trees for a sentence of the form *der kleine Junge hier, der schläft , träumt von dem großen Fisch im Meer*

⟨*6 different parse trees*⟩≡

```
Binmod> p -cat=S "a n r r v a n r"
Pred (DetCN (AdjCN a (RelCN (RelCN n r) r)))
     (Compl v (DetCN (AdjCN a (RelCN n r))))
Pred (DetCN (AdjCN a (RelCN (RelCN n r) r)))
     (Compl v (DetCN (RelCN (AdjCN a n) r)))
Pred (DetCN (RelCN (AdjCN a (RelCN n r)) r))
     (Compl v (DetCN (AdjCN a (RelCN n r))))
Pred (DetCN (RelCN (AdjCN a (RelCN n r)) r))
     (Compl v (DetCN (RelCN (AdjCN a n) r)))
Pred (DetCN (RelCN (RelCN (AdjCN a n) r) r))
     (Compl v (DetCN (AdjCN a (RelCN n r))))
Pred (DetCN (RelCN (RelCN (AdjCN a n) r) r))
     (Compl v (DetCN (RelCN (AdjCN a n) r)))
```

Can't we avoid this (without paying a price in semantics)?

- Use dependent categories to reduce the number of parse trees

  ⟨*Basic idea*⟩≡

```
cat Md ;              -- modification dimension
fun e, m : Md ;       -- unmodified, modified
cat CN Md Md Md ; -- CN Adj? Adv? Rel?
```

  and then collect modifiers in each dimension in order:
  `(A1 .. (An-1 An)..) N (Adv1 ...) (Rel1 ...)`
  Eliminate trees caused by artificial bracketings `((A1 A2) A3)`.
  |trees| drops from exponential to linear in sentence length.

  Human impression: long sentences have *few* readings, $O(n)$
  What's wrong if grammar+lexicon lead to millions of readings?
  Just the missing intonation and semantic restrictions?

We need slight changes in the abstract grammar to distinguish unmodified from modified CNs (in types, not in record fields).

⟨*BinmodDep.gf*⟩≡

```
abstract BinmodDep = {
cat S; VP; NP; AP; RS; V2; A; R;
    Kind; CN Kind Kind;  -- type CN *depends* on 2 kinds
fun e,m : Kind;
    AdjCN : AP -> CN e e -> CN m e; -- AP-modified only
    RelCN : CN e e -> RS -> CN e m; -- RS-modified only
    AdjCnRel : AP -> CN e e -> RS -> CN m m ;
    DetCN : (x,y:Kind) -> CN x y -> NP ;
    Compl : V2 -> NP -> VP ;
    Pred :  NP -> VP -> S ;
    a : A ; v : V2 ; r : R ; n : CN e e ; -- unmodified
    AdjAP : A -> AP -> AP ; Adj : A -> AP ;
    RelRS : R -> RS -> RS ; Rel : R -> RS ;
}
```

⟨*BinmodDepGer.gf*⟩≡

```
concrete BinmodDepGer of BinmodDep = open Prelude in {
lincat S, VP, NP, CN, AP, RS, V2, A, R, Kind = { s:Str }
lin e, m = {s=[]} ;
    AdjCN ap cn = { s = ap.s ++ cn.s } ;
    RelCN cn rs = { s = cn.s ++ rs.s } ;
    AdjCnRel ap cn rs = { s = ap.s ++ cn.s ++ rs.s } ;
    DetCN _ _ cn = { s = cn.s } ;  -- simplified: no det
    Compl v2 np = { s = v2.s ++ np.s } ;
    Pred np vp  = { s = np.s ++ vp.s } ;
    a = { s = "a" } ;    r = { s = "r" } ;
    v = { s = "v" } ;    n = { s = "n" } ;
    AdjAP a ap = { s = a.s ++ ap.s } ;
    Adj a = { s = a.s } ;
    RelRS rel rs = { s = rel.s ++ rs.s } ;
    Rel rel = { s = rel.s } ;
}
```

In this way, the same modifications lead to only *one* analysis:

⟨`BinmodDepGer` *gives only 1 tree*⟩≡

```
BinmodDep> p -cat=S "a n r r v a n r"
Pred (DetCN m m (AdjCnRel (Adj a) n (RelRS r (Rel r))))
     (Compl v (DetCN m m (AdjCnRel (Adj a) n (Rel r))))
```

Dependent types here help to increase the number of categories without adding more category names.

Hope:

- Slight change of the grammar leads to much less trees
- May be useful to obtain a small, human-checkable number of trees as parser outputs,

  even if the "real" RGL is not changed for semantic reasons

Remark `gf-3.8`: parse problems with split categories (definition??)

⟨*Expl: a) input of split strings?*⟩≡

```
Lang> p -cat=NP "der kleine Junge hier , der schläft"
The sentence is not complete
Lang> p -cat=NP "hier"
AdvNP (UsePron ?1) here_Adv
... (340 trees as NP!) ...
DetCN ?1 (AdvCN ?2 here_Adv)
Lang> p -cat=NP ", der schläft ,"
... (25 readings as NP) ...
DetCN ?1 (RelCN ?2 (UseRCl ..(RelVP IdRP (UseV sleep_V)))
```

⟨*Expl: b) incomplete linearization*⟩≡

```
Lang> l (RelNP (DetCN .. (AdvCN (AdjCN (PositA small_A) (
                  (UseRCl .. (RelVP IdRP (UseV sleep_V))))
der kleine Junge
```

Special categories `GluedNP` make cross-lingual testing difficult

III. Imposing restrictions on the subject position

Problem: GF makes distinctions according to object types of verbs,

- `V2` = verbs with NP- or PP-object,
- `VS` = verbs with S-object

but GF does not make a similar distinction according to the subject position of verbs:

- No restriction for 0-ary verbs:

  ⟨*Expl: the house rains*⟩≡

  ```
  Lang> p -cat=Cl "das Haus regnet"
  PredVP (DetCN (DetQuant DefArt NumSg) (UseN house_N))
         (UseV rain_V0)
  ```

- No difference between NP- vs. S-subject:

  ```
  John:NP believes:VS (that it rains):S
  (that it rains):S surprised:?V2 John:NP
  ```

We'd need more verb (and VP) classes.

Goal: What can we do by using dependent categories?

- Add arguments for the subject type: `V np`, `V cl`, `VP np`, ...
- Add linearizations of clauses where a sentential subject/object is moved and leaves a corrolate in place:
  Expl. *It surprised John, that it rained*
- Do this by extending Aarne Ranta's "Types and Records for Predication" (EACL 20140), cf.`gf/lib/src/experimental/`

So far: partial implementation as `DepLang.gf`, `DepLangGer.gf`, independent of Ranta's `Pred*`-grammar.

⟨*From abstract grammar DepLang.gf*⟩≡

```
-- Argument lists
cat Args ;
fun O : Args ;                    -- empty list
    np, cl : Args -> Args ; -- list extensions

-- Independent and dependent categories
cat S ; PN ; NP ;
    V (xs:Args) ; VP (xs:Args) ;
    Cl (xs:Args) ; CN (xs:Args) ;
```

⟨*from abstract grammar DepLang.gf*⟩≡

```
fun
-- Inititial formation of VP(objs) from V(subj objs)
  UseOV: Temp -> Pol -> V 0 -> Cl 0 ;
  UseNV: (xs:Args) -> Temp -> Pol -> V (np xs) -> VP xs ;
  UseSV: (xs:Args) -> Temp -> Pol -> V (cl xs) -> VP xs ;

-- Complementation with nominal or clausal object
  ComplNP: (xs:Args) -> VP (np xs) -> NP -> VP xs ;
  ComplCl: (xs:Args) -> VP (cl xs) -> Cl 0 -> VP xs ;

-- Predication with nominal or clausal subject
   --  Expls: (NP | that S) is strange : Cl 0
  NpPred : (xs:Args) -> NP -> VP xs -> Cl xs ;
  ClPred : (xs:Args) -> Cl 0 -> VP xs -> Cl xs ;
```

⟨*form abstract grammar DepLang.gf*⟩≡

```
fun
  beer, book, boy : CN 0 ;
  brother : CN (np 0) ;        -- ResGer.N2 with c2=Gen
  belief : CN (cl 0) ;         -- ResGer.NS

  rain : V 0 ;                 -- 0-ary verb
  sleep : V (np 0) ;           -- V1 with np-subject
  read : V (np (np 0)) ;       -- V2 with np-object
  wait : V (np (np 0)) ;       -- V2 with pp-object
  believe : V (np (cl 0)) ;    -- VS: John believes that S.
                               -- With new types:
  surprise : V (cl (np 0)) ; -- That S, surprises John.
  cause : V (cl (cl 0)) ;      -- That S1, caused S2.
```

GF-Problem: dependent categories `V xs` have to use the *same* linearization type `lin V = ...` for all `xs`. (go for the "worst case")

⟨*From resource file* `DepArityGer.gf`⟩≡

```
resource DepArityGer = open ResGer,CatGer in {
  param
    SForm = That | Conj ; -- types of sentential arg
    ArgTy = In1 Case | In2 SForm | None ; -- arg type
  oper -- admit at most 3 args
    SubjTy : Type = { subj : ArgTy } ;
    ArgsTy : Type = { cmpl1 : ArgTy ; cmpl2 : ArgTy } ;
    NoArgs : ArgsTy = { cmpl1 = None ; cmpl2 = None } ;
    subject = overload {
      subject: Prep->SubjTy = \p -> {subj = In1 ..p..} ;
      subject: SForm->SubjTy = \sf -> {subj = In2 sf} } ;
    args = overload {
      args : SForm -> ArgsTy = \sf -> {cmpl1 = In2 sf;
                                       cmpl2 = None} ;
      ... } }
```

The concrete `DepLangGer.gf` uses `DepArityGer` and `ResGer`, `TenseGer`, `ParadigmsGer` from the GF rescource grammar library.

⟨*from concrete grammar* `DepLangGer.gf`⟩≡

```
  lincat
    CN = ResGer.Noun ** ArgsTy ;
    NP = { s : Case => Str ; a : Agr } ;
    V  = { s : Tense => Polarity => Agr => Str*Str*Str ;
           c1 : OCase ;            -- case/prep(=VerbGer.V.c2)
           c2 : OCase ;            -- case/prep of object 2
         } ** SubjTy ** ArgsTy ; -- added subj,compl-types
    VP = { s : Agr => Str * Str * Str ;
           c1,c2 : OCase ;           -- object case/prep
           obj1,obj2 : Agr => Str ;  -- nominal objects
           ext1,ext2 : Str ;         -- sentential objects
           adv1,adv2 : Str ;
         } ** SubjTy ** ArgsTy ;    -- subject type,
                                    -- arity reduction
```

To implement the lexemes, use the RGL with `L=LexiconGer`, `P=ParadigmsGer` and impose restrictions on `subj`,`compl`-fields:

⟨*from DepLangGer.gf*⟩≡

```
oper
  appV: L.V -> Tense=>Polarity=>Agr=>Str*Str*Str = ...
  V0, V1, V2, VS, SV2 = V ; -- as the lincat above
  -- dflt=filler for 'non-existing' arguments
  mkV0 : L.V -> V0 = \v -> v ** NoSubj ** NoArgs **
    { s = appV v ; c1 = dflt ; c2 = dflt }  ;
  ...
  mkSV2 : L.V2 -> SV2 = \v -> lin SV2 (
      v ** (subject That) ** (args P.accPrep) **
      { s = appV v ; c1 = P.accPrep ; c2 = dflt } ) ;
lin
  rain = mkV0 (P.regV "regnen") ;
  ...
  surprise = mkSV2 (P.mkV2 (P.regV "wundern") P.accusativ
```

To implement the predication rule with clausal subject, i.e.

⟨*from DepGrammar.cf*⟩≡

```
fun ClPred : (xs:Args) -> Cl O -> VP xs -> Cl xs ;
```

GF does not allow one to use the arguments `xs` in the lin-function; but we can distinguish different cases of `subj`,`compl`-fields:

⟨*from DepGrammarGer.gf*⟩≡

```
lin ClPred _ cl vp = { s = clPred cl vp } ;
oper clPred : Cl -> VP -> Ord => Str = \cl,vp ->
    let a : Agr = Ag Neutr Sg P3 ;
        v : Str*Str*Str = vp.s!a ;
        thatS : Str = "dass" ++ cl.s!Sub
    in case <vp.subj,vp.cmpl1> of {
      <In2 That,In1 _> => \\o => case o of {
        Sub => "es" ++ v.p1 ++ v.p2 ++ vp.obj1!a ++ v.p3
               ++ "," ++ thatS ; -- moved sent.subject
        Decl => thatS ++ "," ++ ... ++ vp.obj1!a ++ v.p3
        Inv => ... } ;
      ...
```

⟨*Sentential subject, extracted in subordinate sentences:*⟩≡

```
DepLang> p "dass es regnet , wundert Johann nicht"
DeclS (ClPred 0 (UseOV Pres ?3 rain)
               (ComplNP 0 (UseSV (np 0) Pres ?9 surprise)
                          (UsePN john)))
DepLang>  p "es wundert Johann nicht , dass es regnet"
SubordS (ClPred 0 (UseOV Pres ?3 rain)
               (ComplNP 0 (UseSV ...) (UsePN john)))
```

Metavariables `?3,?9` are caused by my poor linearization of `Temp`.

⟨*Sentential object: extraction and corrolate*⟩≡

```
DepLang> gr -cat=S -tr | l
SubordS (NpPred 0 (UsePN john)
             (ComplCl 0 (UseNV (cl 0) Past Pos believe)
                        (UseOV Pres Neg rain)))
```

```
[weil] Johann es glaubte , dass es nicht regnet
```

Many of the possible movements have not yet been implemented.

This approach still has a problem: in our predication rules

⟨*from DepLang.gf*⟩≡

```
NpPred : (xs:Args) -> NP -> VP xs -> Cl xs ;
ClPred : (xs:Args) -> Cl 0 -> VP xs -> Cl xs ;
```

the type `(VP xs)` does not encode the admitted subject type.
Hence, we still may combine a `vp:(VP xs)` with a wrong subject:

⟨*Sentential subject combined with a verb expecting a nominal subject*⟩≡

```
DepLang> Expression:
ClPred 0 (UseOV Pres Pos rain)
       (ComplCl 0 (UseNV (cl 0) Pres Neg believe)
          (NpPred 0 (UsePN john) (UseNV 0 Pres Neg sleep)))
Type:        Cl 0
```

To avoid this, the VP-type needs also an argument for the subject position, and the rules need to be similar to complementation rules.

Conclusion

I. Having a number and mass/count distinction at the type level can be done by dependent types, but seems to raise questions of how clear-cut the distinctions can be made.

II. It seems not too hard to reduce the spurious ambiguities caused by the application ordering of binary modification rules. (This can be done using dependent categories, but also by splitting existing non-dependent categories into several ones, i.e. unmodified vs. modified in certain dimensions.)

III. Languages make a distinction of verbs according to the type of subjects that is not made in GF yet; the demo implementation uses too many record fields, which increase the grammar size.

A better implementation and a combiniation with A. Ranta's `gf/lib/src/experimental/Pred*` should be made.

But: In GF, restrictions implemented as arguments of categories apply only *after* parse trees have been built.