Can We Add Subtyping to GF?

Hans Leiß
leiss@cis.uni-muenchen.de

Universität München
Centrum für Informations- und Sprachverarbeitung

4th Grammatical Framework Summer School
Gozo, July 13–24, 2015

We focus on subtypes in abstract *syntax*, i.e. subtypes in abstract resource grammars rather than application grammars.

GF admits subtypes in concrete grammars, but not in abstract ones.

- Subtypes in concrete grammar: record subtyping
- Subtypes in abstract grammar: do they make much sense?

Content:

- Notions of subtyping, and their use in GF
- Examples with dependent types
- Examples of possible use of subtypes in abstract grammar
- Issues of implementation of subtypes in abstract grammar
- Known complexity results with respect to subtyping

Basic idea of subtypes

The general idea (in object-oriented programming, for example) is:

- if $b : B$ and $B \leq A$, then $b : A$

If function types $A \to C$ consist of *total* functions only, this implies

- if $f : A \to C$ and $b : B \leq A$, then f applies to b, and $f(b) : C$.

In particular, when $C =$ `bool`:

> *objects of a subtype $B \leq A$ can have all properties that objects of type can A have, and possibly some more.*

If the functions are syntactic constructions, this means:

> *expressions of a category $B \leq A$ can be used in every construction where those of category A can be used.*

Subtyping for basic objects: subsumption

In a CFG, expression categories X are interpreted as string sets $D_X = L(X)$, so $B \leq A \leq \mathit{string}$ means $L(B) \subseteq L(A) \subseteq D_{\mathit{string}}$.

Hence: in a CFG,

$$A \to B_1 \cdots B_n \mid C_1 \cdots C_k$$

amounts to subtype assumptions

$$A \geq B_1 \cdots B_n \quad \text{and} \quad A \geq C_1 \cdots C_k$$

For example, $NP \to Pron$ resp. $Pron \leq NP$ means: a pronoun can occur whereever a noun phrase can occur.

But: *Pron* $\leq$ *NP* isn't really true in German:

- Possessive NP^{gen}-attributes in *Det N* NP^{gen} must not be *Pron*'s:

  *(alle) die Werke (des Autors | Goethes | *seiner)*

  Pron's have special possessive forms as determiners:

  (alle) seine Werke

- Another possessive construction applies (better?) to all NPs:

  die Werke ((?)von dem Autor | von Goethe | von ihm)

So, *Pron* $\not\leq$ *NP*: (Maybe $NP_{P3} \leq Pron_{P3}$.) Likewise $N^{gen}_{pl} \not\leq NP^{gen}_{pl}$:

*ein Tag (des Glücks | *Glücks); die Freude (der Fische | *Fische),*

A more precise possessive rule were: $NP \rightarrow Det\ N\ (NP\text{-}Pron\text{-}N)^{gen}$

Coercive subtyping

Subtyping on base domains, like $bool \leq int$ in programming, is uncommon in GF (if it exists at all).

What occurs very often is subtyping via a coercion function,

$$B \leq A \iff D_B \subseteq_c D_A, \text{ meaning } c : D_B \to D_A.$$

Often, c is injective or even a constructor, such as

```
UseN : N -> CN ;
UsePron :  Pron -> NP;
ImpVP : VP -> Imp ;
```

An advantage(?) is that different coersions can cause $B \leq A$, as in

⟨*from Ranta 2014*⟩≡

```
fun Decl : Cl -> S ; Quest : Cl -> S
```

Subtyping for records: "component omitting" coercion

A record type $\rho = \{\, i : \tau_i \mid i \in I \,\}$, where $I \subseteq \mathit{Lab}$ is a finite set of labels, is interpreted as the set of all dependent functions

$$D_\rho = \{\, f : I \to \bigcup_{i \in I} D_{\tau_i} \mid f(i) \in D_{\tau_i}, 1 \leq i \leq n \,\}.$$

Write $f \in D_\rho$ as $\{\, i = a_i \mid i \in I, a_i \in D_{\tau_i} \,\}$ where $a_i = f(i)$.

For record types, subtypes may have additional fields and fields with smaller types

$$\frac{J \supseteq I, \quad \tau_i \leq \sigma_i \text{ for all } i \in I}{B := \{\, j : \tau_j \mid j \in J \,\} \leq \{\, i : \sigma_i \mid i \in I \,\} =: A}\,(\mathit{rec} \leq)$$

Here the interpretation D_B is subsumed by D_A via a coercion c

$$B \leq A \iff \{\, c(f) \mid f \in D_B \,\} \subseteq D_A \iff: D_B \subseteq_c D_A$$

where c coerces f by $c_{B,A}(f)(i) = c_{\tau_i,\sigma_i}(f \restriction_I (i))$ for $i \in I$.

Objects of a subtype contain more and more detailed “information”.

⟨*Examples from CatEng.gf*⟩≡

```
NP = {s : NPCase => Str ; a : Agr} ;
Pron = {s : NPCase => Str ; sp : Case => Str ; a : Agr} ;

Ord = { s : Case => Str } ;
Num  = {s : Case => Str ; n : Number ; hasCard : Bool} ;
Card = {s : Case => Str ; n : Number} ;

Subj = {s : Str} ;
Prep = {s : Str; isPre : Bool} ;
```

As record types, this gives

`Pron < NP`,   `Num < Card < Ord`,   `Prep < Subj`.

But, of course, GF does *not* use prepositions as subjunctions etc.

Subtyping for functions: $f : (A \to B) \leq (A' \to B')$ maps all elements of $A \geq A'$ to values of $B \leq B'$

If $(A \to B) = \{ f \mid \forall a : A, f(a) : B \}$ is the set of *total* functions, then $\to$ is *contravariant* in its argument and *covariant* in the target:

$$\frac{A' \leq A \qquad B \leq B'}{(A \to B) \leq (A' \to B')} \, (\to \leq)$$

Expl: In the RGL of GF, `V2 = V ** {c2:Case} < V`. Hence

- any $f : C \to V2$ is a $f : C \to V$
- any $g : V \to C$ is a $g : V2 \to C$.
  `PassV2 : V2 -> VP` is not applicable to arbitrary `v:V`.

$\langle$*Example from Verb.gf, using coersive subyping*$\rangle\equiv$

```
fun SlashV2a : V2 -> VPSlash ;  -- aka V2 < VPSlash
    ReflVP : VPSlash -> VP ;
```

Hence `ReflVP` is applicable to `v2:V2`, via `(SlashV2a v2)`.

Record subtyping in GF's concrete grammars

Although GF's concrete syntax has record subtyping, the hidden `lock`-fields of implementation types block apparent subtypings:

⟨*Example from CatEng.gf*⟩≡

```
NP = {s : NPCase => Str ; a : Agr} ;
Pron = {s : NPCase => Str ; sp : Case => Str ; a : Agr} ;
```

⟨*Actual implementation types are different*⟩≡

```
Lang> cc -unqual NP
{s : NPCase => Str; a : Agr; lock_NP : {}}
Lang> cc -unqual Pron
{s : NPCase => Str; a : Agr; sp : Case => Str;
 lock_Pron : {}}
```

Hence, we don't have `Pron < NP`. Instead, GF uses a coercion `UsePron : Pron -> NP` to drop fields and adjust the lock-field.

⟨*Implementation of* `UsePron : Pron -> NP`⟩≡

```
\p -> {s = p.s; a = p.a; lock_NP : {} = <>}
```

⟨*Example: abstract with coercion function*⟩≡

```
abstract Subtype = {
cat A ; B ; C ; D ;
fun b : B ; UseB : B -> A ; }
```

⟨*Example: concrete with $B < A$ via coercion*⟩≡

```
concrete SubtypeConc of Subtype = {
lincat A = {s:Str; r:C} ; B = {s,t:Str; r:D} ;
       C = {c:Str} ;      D = {c,d:Str} ;
lin b = {s,t = "b"; r = lin D {c = "c" ; d = "d"}} ;
    UseB x = lin A {s = x.s; r = x.r} ; }
```

⟨*Coercion function, omitting field* `t` *and subfield* `d` *of field* `r`⟩≡

```
Subtype> cc UseB b
{s : Str = "b";
 r : {c : Str; lock_C : {}}
   = {c : Str = "c"; d : Str = "d"; lock_D : {} = <>};
 lock_A : {} = <>}
```

How then is record subtyping used in concrete grammars – without coercion functions?

Functions with record argument and result types are defined as operations, as in

⟨*From ResEng.gf*⟩≡

```
oper
  Verb : Type = { s : VForm => Str ; isRefl : Bool } ;
  VP : Type = { s : VerbForms ; ... } ;
  predV : Verb -> VP = \verb -> { s = ... ; ... }
```

Then, different subtypes of `Verb` are introduced by

⟨*From CatEng.gf*⟩≡

```
lincat V, VS, VQ, VA = Verb ;
```

leading to record types `Verb ** { lock_V : {} }` etc:

⟨*Implementation types of V, VS, etc.* < `Verb`⟩≡

```
{s : VForm => Str; isRefl : Bool; lock_V : {}}
{s : VForm => Str; isRefl : Bool; lock_VS : {}}
```

Finally, as `V,VS,VA,VQ < Verb`, `predV` can be applied to all kinds of verbs (without using coercion functions):

⟨*From abstract/Verb.gf and VerbEng.gf*⟩≡

```
data
  UseV     : V   -> VP ;          -- sleep
  ComplVS  : VS  -> S  -> VP ;  -- say that she runs
  ComplVQ  : VQ  -> QS -> VP ;  -- wonder who runs
  ComplVA  : VA  -> AP -> VP ;  -- they become red
lin
  UseV = predV ;
  ComplVS v s  = insertExtra (conjThat ++ s.s) (predV v)
  ComplVQ v q  = insertExtra (q.s ! QIndir) (predV v) ;
  ComplVA v ap = insertObj (ap.s) (predV v) ;
```

Remark: In the abstract syntax, a *category* `Verb` does not exist, and `V,VS,VQ,VA` are just different categories.

Besides the record subtyping, the concrete grammars use coersive subtyping to extend parameter types:

$\langle$*from ResGer.gf*$\rangle\equiv$

```
  param
    GenNum = GSg Gender | GPl ;
    NPForm = NPCase Case | NPPoss GenNum Case ;
```

in this case making disjoint unions

$$\begin{array}{rcl} D_{GenNum} & \simeq & D_{Gender} \cup \{Pl\}, \\ D_{NPForm} & \simeq & D_{Case} \cup (D_{GenNum} \times D_{Case}) \end{array}$$

It would sometimes be nice to have simple subsumptions between datatypes, such as

$\langle$*fake code*$\rangle\equiv$

```
  subparam Case-Nom < Case ;
  fun ReflPron : { s: Case-Nom => Str ; a : Agr } ;
```

Dependent types in the abstract grammar

The abstract grammar of GF has dependent types, but no subtypes.

⟨*Type hypothesis*⟩≡

```
Hyp := (x : T) | (_ : T) | T
```

⟨*Context*⟩≡

```
G :=     | Hyp G
```

⟨*Basic category declaration*⟩≡

```
cat C G ;
```

A category declaration `cat C (x1:T1) ... (xn:Tn)` introduces a type constructor

```
C : T1 -> ... -> Tn -> Type
```

For $a_i : T_i : \mathit{Type}$ and $n > 0$, `(C a1 ... an)` is a *dependent* type.

GF-book, Exercises 6-4*, 6-5* , p.132.

Subject-verb agreement in number could be built into predication:

⟨*NumberAgr.gf*⟩≡

```
abstract NumberAgr =
  cat S ; Number ;
      NP Number ; VP Number
  fun Pred : (n:Number) -> NP n -> VP n -> S ;
```

Verb types could be made dependent on subcat-frames (HPSG):

⟨*Subcat.gf*⟩≡

```
abstract Subcat = {
  cat VSub ; VP ;
      Comps VSub ;
  fun Compl : (sub : VSub) -> V sub -> Comps sub -> VP
```

But: Current RGL does not use dependent types (as far as I know).

Example (Subcat frames)

Aarne Ranta's grammar in "Types and Records for Predication" parameterizes phrase categories by a list of argument types. When combining expressions, the list of argument types is reduced by the type of the argument expression.

⟨*from AR's grammar*⟩≡

```
  cat Arg ; Args ; V Args ; VP Args ; ...
  fun ap, cl, cn, np, qcl, vp : Arg ;
      0 : Args ; c : Arg -> Args -> Args ; -- lists

      UseV : (x:Args) -> V x -> VP x ; -- (simplified)
      ComplNP : (x:Args) -> VP (c np x) -> NP -> VP x ;
      ReflVP : (x:Args) -> VP (c np x) -> VP x ;
      ...
```

Is this kind of parameterization a substitute to subcategories `V x` < `V` etc.? The code is simpler than with different categories `V_x` etc.

Example (Adverbial dimensions)

Different verbs are modifiable in different adverbial dimensions. Let Vs and VPs carry the dimensions in which they can be modified, and when combining with a modifier, remove the dimension at VP.

$\langle$*Pseudocode* $\rangle\equiv$

```
cat Kind ;     fun loc, dir, tmp, instr, mod : Kind

cat Adv Kind ; V Kind ; V Kind Kind ; ...
              VP Kind ; VP Kind Kind ; ...
fun here : Adv loc ; later : Adv tmp ; ...
    live : V loc mod ; -- live nicely in Paris
    travel : V dir instr ; -- travel to Malta by plane
    ModVP : (x, y : Kind) -> VP x y -> Adv y -> VP x ;
    ...
```

There is a subcat hierarchy converse to $\subseteq$ on sets of `Kind`: if $X \subseteq Y \subseteq$ `Kind`, then `(subcat VP Y < VP X)` (using c-lists X).

Example (Number restrictions in NPs and VPs)

Not only in GF, NPs have inherent number, gender, person. But coordinated NPs actually don't, so GF uses artificial values:

$\langle$*agreement values of "du oder wir"*$\rangle\equiv$

```
Lang> cc -unqual (ConjNP or_Conj
                   (BaseNP (UsePron youSg_Pron)
                           (UsePron we_Pron))).a
Ag Fem Pl P1
```

As number and person are needed to select the the verb form, such NPs cannot be used as subjects:

*(du oder wir) *(mußt | müssen) es tun*
$\mapsto$ *((du oder wir), jemand$_{3P,Sg}$) muß es tun*

Also: verbs may demand their subjects (or objects) to be plural; determiners split into mass- vs. individual-det's and create NPs in sg resp. pl. (`*many gold`, `*much days`)

So, to be precise, we seem to need subcategories:

- NPs with fixed number: NPsg and NPpl, (usable as subject or object, when meeting a possible number constraint of the verb)
- NPs with no definite number: NPnone (usable as object,. . .)
- VPs constraining the number of its NP-argument: VPpl
- VPs not constraining the number of its NP-argument: VPany

This gives: $VP_{any} \leq VP_{pl}$ and $NP_{sg}, NP_{pl} \leq NP_{none}$.

Or can we do it with dependent types and several predication rules?

$\langle$*from DepReciprocals.gf*$\rangle\equiv$

```
cat Number ;  fun sg, pl, any : Number ;
cat NP Number ; V1 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.
    mix2   : V2 any pl ;     -- object must be pl
```

⟨*from DepReciprocals.gf*⟩+≡

```
      UseV1  : (n:Number) -> V1 n -> VP n ;
      PredVP : (n:Number) -> NP n -> VP n -> S ; -- n-agree
      PredVPany : (n:Number) -> NP n -> VP any -> S ;
      ...
      ComplV3 : (n,m,l:Number) -> V3 n m any -> NP l
                                          -> VPSlash n m ;
      -- reciprocal obj reduce VP's arity and enforce pl
      Reci2any : VPSlash any any -> VP pl ;
      Reci2pl  : VPSlash any pl -> VP pl ;
      Reci3 : V3 any any any -> VPSlash any pl ;
              -- (I|we) introduce them(pl) to each other
```

Works partly, not precisely, as I misused NP_{any} for NP_{none}.

Doable, but clumsy, if we need different kinds `KindNP`, `KindVP` and many functions `Pred_k1_k2 : NP k1 -> VP k2 -> S`.

Where could we use subtypes in abstract syntax?

Ignoring the possible usefulness of subtypes for *application grammars* – why would we want subtypes in abstract *syntax*?

Example (Verbs: V0 < V?)

- 0-ary verbs admit only impersonal constructions:
  `it rains (heavily) (today)`
- grammar rules use suitable kinds of verbs (CG)

Notice: GF's RGB does not separate `V0` from `V`: `fun rain_V0 : V`

⟨*V0 with (good:) impersonal and (bad:) personal construction*⟩≡

```
Lang> parse -cat=Cl "it rains"
ImpersCl (UseV rain_V0)
PredVP (DetNP (DetQuant DefArt NumSg)) (UseV rain_V0)
PredVP (UsePron it_Pron) (UseV rain_V0)
```

Example (V1 with passives < V1?)

- German intransitive action verbs admit a passive:
  `sie arbeiten` – `es wird gearbeitet`
- other intransitve verbs don't:
  `die Sonne geht auf` – `*es wird aufgegangen`

Likewise for GF's `V2`: should there be a subtype `V2pass < V2`?

⟨*from abstract/Verb.gf*⟩≡

```
  -- *Note*. the rule can be overgenerating, since
  -- the $V2$ need not take a direct object.
     PassV2   : V2 -> VP ;                 -- be loved
```

Example (Deponent verbs < V?)

- deponent verbs in Latin and AGreek lack active forms and use passive/middle forms instead (hence have no passive)
- Should we have `(subcat Vdep < V)`?

Example (Prons and noun phrases: ReflPron < Pron < NP?)

- grammars subsume pronouns under noun phrases, but:
- reflexive and reciprocal pronouns cannot be used as subjects
  `(They | *Themselves) saw the movie`
  `(They | *each other) like (apples | each other)`

Hence: `ReciPron`,`ReflPron` $\not\leq$ `Pron`. We already saw `Pron` $\not\leq$ `NP`.

A. Conclusion so far: For lexical categories, what intuitively may seem to be a subtype `B` of category `A` corresponds often to a *subset of words*, *lacking* some behaviour or having *some special* behaviour. That is, $A = (B + \ldots)$ is a disjoint sum with summand B,

$$D_A = D_B \,\dot{\cup}\, \ldots$$

- If Bs lack some forms other As have, then $B \not\leq A$, but perhaps $A \leq B$. ($RelfPron \not\leq Pron < ReflPron$, $Vdep \not\leq V < Vdep$)
- If Bs enter special constructions, then $A \not\leq B$. ($V \not\leq Vpass$)

B. For phrasal categories, we have seen candidates for $\leq$

- VPs, VPSlashs having/lacking a plural-constraint (per arg)
- VPs,VPSlashs p.ordered by modifiability in adverb kinds[1]
- NPs with/without clear number and person feature.

The last example extends to other categories:

Example (Coordinated Cs < simple Cs?)

If Cs have a governing feature, `(C coord C)` lacks the feature, if the component `C`s disagree on it. So `(C coord C)` is not usable in every context where `C` is.

- `(ein oder der) (*kleiner|*kleine) Hund`

Similar to NP_{none}, we have $Det_{strong}, Det_{weak}, Det_{mixed} < Det_{none}$.

---

[1] Similarly for VPs with alternative complement frames.

Example (Number restrictions, (cont.))

We arrived at: $VP_{any} \leq VP_{pl}$ and $NP_{sg}, NP_{pl} \leq NP_{none}$.

- NPs with fixed number: NPsg and NPpl, (usable when ...)
- NPs with no definite number: NPnone (usable as object,. . .)
- VPs constraining the number of its NP-argument: VPpl
- VPs not constraining the number of its NP-argument: VPany

Which predication rules do we need to distinguish?

```
   NPnone      VPpl     PredAny: NPsg+NPpl -> VPany -> S
   /    \        |      PredPl: NPpl -> VPpl -> S
 NPsg  NPpl    VPany    ComplAny: VP/any -> NPnone -> VP
                        ComplPl : VP/pl -> NPpl -> VP
```

How much does the $\leq$ save? Conclusion: Only very flat hierarchies?

Example (SC as subjects – for verbs of suitable kind)

Sentences, questions, infinitival phrases can be used as subjects:

⟨*from abstract/Sentence.gf, with coercive subtyping*⟩≡

```
data
  EmbedS     : S  -> SC ;        -- that she goes
  EmbedQS    : QS -> SC ;        -- who goes
  EmbedVP    : VP -> SC ;        -- to go

  PredSCVP   : SC -> VP -> Cl ; -- that she goes is good
```

But each of the three kinds of SC can be subjects of particular verbs only, so the rule `PredSCVP` is overgenerating. In fact, the semantic domains are fairly different, satisfying a domain equation like

$$\begin{aligned} D_{SC} &\simeq D_S + D_Q + D_{VP} \\ &\simeq D_t + D_{e^*} + D_{(e \to t)} \end{aligned}$$

To restrict overgeneration, one ought to have dependent categories:

⟨*categories SC and VP separated into three kinds*⟩≡

```
cat Kind ;
fun sK, qsK, vpK : Kind ;

cat SC Kind ; VP Kind ;
fun EmbedS : S -> SC sK ;
    EmbedQS : QS -> SC qsK ;
    EmbedVP : VP -> SC vpK ;

    PredSCVP : (k : Kind) -> SC k -> VP k -> Cl ;
```

Of course, `(VP k)` had to be build from `(V k)`, `(V2 k)` etc.

The `(VP k)` are *special* `VP`s with `SC` subject, so $(\texttt{VP k}) \not< \texttt{VP}$, rather

$$\texttt{VP} \simeq (\texttt{VP sK}) + (\texttt{VP qsK}) + (\texttt{VP vpK}) + (\texttt{VP nom}) + \ldots$$

Subtyping in HPSG

HPSG comes with a hierarchy of sign sorts and constraints on signs.

The sort hierarchy is a finite tree $(S, \leq)$; its root is $\leq$-maximal. Each node $\sigma \in S$ in the tree is *partitioned* by it's immediate predecessors $\sigma_1, \ldots, \sigma_n < \sigma$, i.e.

$$D_\sigma \simeq D_{\sigma_1} + \ldots + D_{\sigma_n}$$

is the disjoint sum of its immediate subsorts.

HPSG: expletive and referential NPs as subtypes of NP:

$$NP \simeq NP[expl] + NP[ref]$$

Then, with subtyping for functions,

$$\frac{NP[expl] < NP}{V\langle NP\rangle := (NP \to S) < (NP[expl] \to S) =: V\langle expl\rangle}\,(\to\leq)$$

But: does $V\langle NP\rangle < V\langle ref\rangle, V\langle expl\rangle$ mean more than $V\langle NP\rangle = \emptyset$?

Problems with subtyping in the abstract grammar

- P1 Subtype declaration: `(cat Pron < NP)` versus coercion construction: `(UsePron : Pron -> NP)`.

  With subtype declarations we might omit the coercion constructors in abstract syntax trees – if they are unique.

  But: if $B \leq A \leq C$ and $B \leq A' \leq C$, how to coerce Bs to Cs?

- P2 Which properties of < have to be checked when compiling an abstract grammar with subtype declarations?

  Just antisymmetry of the reflexive transitive closure $\leq^*$?
  How expensive is that?

- P3 Can an abstract `(cat A < B)` always be implemented by record subtyping? (And do we have to block the converse? I.e. distinguish `(lincat A < B)` implementing `(cat A < B)` from structural record subtyping (independent of abstract <)?

P4 Which subtypings (`cat A < B`) hold for many languages?

Example: Reflexive verbs: language dependent, inherent vs. reflexive use, different forms:

⟨*English*⟩≡

```
to enjoy oneself : V[refl]      -- obj reflexive
to blow one's nose : V2[refl'] -- poss reflexive
```

⟨*German*⟩≡

```
sich freuen : V[refl]           -- inh. reflexive
sich[acc] schneuzen : V[refl]  -- inh. reflexive
sich[dat] die Nase putzen : V3[nom,dat,acc]
```

Example: ACI-verbs are different in different languages (Jpn: only “lassen”, many Langs: perception verbs)

Implementing subtypes of abstract syntax

GF has, if all linearization categories are record types:

abstract	concrete	implementation
`cat A`	`lincat A = {...}`	`A = {...; lock_A:{}}`

It seems natural to implement subcategory declarations as follows:

abstract	concrete	implementation
`subcat B < A`	`lincat B < A`	`B = {...; lock_B,lock_A:{}}`

- The implementation type of `B` would have to collect the `lock_A`-labels of all (immediate) supercats `A > B` of `B`.
- The compiler ought to check that the `(subcat B < A)`-declarations span a partial order $<$ on the categories.
- The reflexive transitive closure $<^*$ of $<$ were needed to check applications of `lin`-functions.

Subtypes for dependent types?

Currently, GF ignores the kind argument `k` of a dependent category `(A k)` both in the `lincat` and in the `lock`-field for `(A k)`:

abstract	concrete	implementation
`cat Kind ;`	`lincat Kind = {...}`	`Kind = {...;lock_Kind:{}}`
`cat A Kind`	`lincat A _ = {...}`	`A = {...; lock_A:{}}`

It follows that linearization functions `lin f : (A k) -> C` are independent of `k`.

With `subcat`, one would probably need `lincat A k = {..}` and lock-fields `lock_(A k)` depending on `k`.

Should dependent type constructors be monotone? In which sense?

$$\frac{\texttt{subcat Kind < Kind' ; cat A Kind' ;}}{\texttt{cat A Kind ;}}\,(\mathit{cat} <)$$

$$\frac{\texttt{subcat Kind < Kind' ; cat A Kind'}}{\texttt{subcat (A Kind) < (A Kind')}}\,(\mathit{subcat} <)$$

For $(\mathit{cat} <)$, note that $\texttt{Kind} \subseteq_{\mathsf{c}} \texttt{Kind}'$ is not injective. If $c(k1) = c(k2)$, should `A k1` and `A k2` be different or equal?

Do we by $(\mathit{subcat}\ <)$ really want

$$\frac{\texttt{subcat Kind < Kind' ; cat A Kind' ; fun k:Kind, k':Kind' ;}}{\texttt{subcat A k < A k'}}\,?$$

In the GF-list discussion in March(?) Aarne wanted:

$$\frac{\mathit{car} < \mathit{vehicle}}{\mathit{NP}\ \mathit{car} < \mathit{NP}\ \mathit{vehicle}}$$

More generally, for record types ρ, a dependent type constructor $C : \rho \to \mathit{Type}$ ought to be monotone:

$$\frac{r : \rho, \quad t : \tau, \quad \rho \leq \tau : \mathit{Type}}{Cr \leq Ct}\,(\mathit{dep}\ \ \leq)$$

With dependent type constructor $B : A \to \mathit{Type}$, we can make (ordered) *contexts* $(a : A)\ (b : B(a))$ but not *record* types $\{a : A, b : B(a)\}$: the set of labels in a record is unordered.

Typability with subtyping

Let *Ty* be the set of simple types $\sigma, \tau := \iota \mid (\sigma \to \tau)$. Let $\leq$ be a partial order on the set of atomic types ι, extended to $\to$-types by

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{(\sigma \to \tau) \leq (\sigma' \to \tau')}\,(\to \leq)$$

Theorem (J.Mitchell) Typability with respect to the typing rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\,(\mathit{Var}) \qquad \frac{\Gamma \vdash t : \tau, \quad \tau \leq \rho}{\Gamma \vdash t : \rho}\,(\mathit{Sub})$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x t : (\sigma \to \tau)}\,(\mathit{Abs}) \quad \frac{\Gamma \vdash t : (\sigma \to \tau), \quad \Gamma \vdash s : \sigma}{\Gamma \vdash (t \cdot s) : \tau}\,(\mathit{App})$$

reduces to the subtype satisfiability problem (see below).

Proof idea: Push (Sub) to the leaves of the derivation tree.

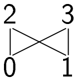
Subtype satisfiability problem $SS(\leq)$

Problem

Given a finite set $E \subseteq Ty(Var) \times Ty(Var)$, *is there a solution* $S : Var \to Ty$ *such that*

$$S\sigma \leq S\tau, \quad \textit{for each } (\sigma, \tau) \in E?$$

For atomic + funcional types:

- Mitchell (1992): $SS(\leq)$ is decidable in NEXPTIME.
- Tiuryn, Wand (1993): $SS(\leq)$ is decidable in DEXPTIME.
- Tiuryn (1992) $SS(\leq)$ is PSPACE hard for $\leq$ = 

- Tiuryn (1992) If $\leq$ is a disjoint union of lattices, then $SS(\leq)$ is in PTIME.
- Benke (1993) If $\leq$ has the Helley property (generalizes lattices and trees), then $SS(\leq)$ is in PTIME.

- Kozen/Palsberg/Schwartzbach (1994) Without $\leq$ on atomic types, but a largest type $\top$, SS for $\to$-types is in PTIME

Remark: With types $\top, \bot$, we get "strange" solutions, like $\bot \to \top$.

For object subtyping, having a type $\top = \{\} = \{\, i : \sigma_i \mid i \in \emptyset \,\}$, and

$$\frac{I \subseteq J \text{ finite}}{\{\, i : \sigma_i \mid i \in J \,\} \leq \{\, i : \sigma_i \mid i \in I \,\}}\,(obj\ \leq)$$

- Palsberg (1995) $SS(\leq)$ for object types is PTIME complete. $SS(\leq)$ is PTIME equivalent to type reconstruction for OOLs

For record subtyping,

$$\frac{I \subseteq J \text{ finite}, \quad \sigma_i \leq \tau_i \text{ for all } i \in I}{\{\, i : \sigma_i \mid i \in J \,\} \leq \{\, i : \tau_i \mid i \in I \,\}} \, (rec \;\; \leq)$$

and systems using at least record-types:

- Vorobyov (1998): *SS* is NP-hard even without atomic types, if we have type constructors → and $\{i : \tau_i, \ldots\}$ ($\neq \{\} = \top$).
- Vorobyov (1998): *SS* is NP-hard with a single atomic type and just the $\{i : \tau_i, \ldots\}$ ($\neq \top$) type constructor.
- Vorobyov (1998): *SS* is NP-hard with the $\{i : \tau_i, \ldots\}$ ($\neq \top$) type constructor and *some* other type constructor with "structural" subtyping (i.e. comparable types have the same top-level constructor)

Subtype satisfiability for GF?

Is there a SS problem for abstract syntax of GF + `subcat`?

GF can introduce

- atomic types `C:Type`, via declarations `cat C`,
- function types `Arrow A C : Type`, via declarations `cat Arrow (_:A) (_:C)`.

What is or should be intended by a “structural” subtype declaration

```
cat C A1 ...  An < C B1 ...  Bm
```

for dependent types? Would it imply $m = n$ and the premise of

$$\frac{A_1 \leq B_1 : \mathit{Type},\ \ldots,\ A_n \leq B_n : \mathit{Type}}{C\ A_1 \ldots A_n \leq C\ B_1 \ldots B_n : \mathit{Type}}\ (\mathit{dep}\ \ \leq)$$

Or should any dependent type constructor `C` be assumed monotone in this sense, without the need of a declaration?

Conclusion ?

We looked at possible uses of subtypes in the abstract syntax and compared them with dependent types for use in resource grammars:

- Using dependent types to split a category C into disjoint parts

  $$C \simeq (C\ k_1) + \ldots + (C\ k_n)$$

  and (if that is possible) split constructions *uniformly*

  $$f : (k : Kind) \to (C\ k) \to \ldots \to D$$

  reduces overgeneration and results in parametric code.
- Language-independet subtype relations seem rare, and mainly related to (a) constraints on arguments of constructions ($\texttt{VP}_{any} < \texttt{VP}_{pl}$) (b) limited usability of coordinations due to feature conflics ($\texttt{NP}_{sg} < \texttt{NP}_{none}$).

  Subcat hierarchies in syntax seem not very deep, so how big is the gain if we have them in the abstract syntax?

References:

1. L. Cardelli: A Semantics of Multiple Inheritance. Information and Computation 76, p.138-164, 1988.
2. J. Tiuryn: Subtype Inequalities. Proc. LICS'92, p.308-315, 1992.
3. D. Kozen, J. Palsberg, M. Schwarzbach: Efficient Inference of Partial Types. J. Compt. Syst. Sci. 49, p.306-3024, 1994.
4. J. Palsberg: Efficient Inference of Object Types. Information and Computation 123, p. 198-209, 1995.
5. S. Vorobyov: Subtyping Functional + Non-Empty Record Types. Proc. CSL 1998.
6. A. Ranta: Types and Records for Predication. Proc. EACL Workshop on Type Theory and Natural Language Semantics, p.1-9, 2014.