Types and Records for Predication

Aarne Ranta
Department of Computer Science and Engineering, University of Gothenburg
aarne@chalmers.se

Abstract

This paper studies the use of records and dependent types in GF (Grammatical Framework) to build a grammar for predication with an unlimited number of subcategories, also covering extraction and coordination. The grammar is implemented for Chinese, English, Finnish, and Swedish, sharing the maximum of code to identify similarities and differences between the languages. Equipped with a probabilistic model and a large lexicon, the grammar has also been tested in wide-coverage machine translation. The first evaluations show improvements in parsing speed, coverage, and robustness in comparison to earlier GF grammars. The study confirms that dependent types, records, and functors are useful in both engineering and theoretical perspectives.

1 Introduction

Predication is the basic level of syntax. In logic, it means building atomic formulas by predicates. In linguistics, it means building sentences by verbs. Categorial grammars (Bar-Hillel, 1953; Lambek, 1958) adapt logical predication to natural language. Thus for instance transitive verbs are categorized as $(n\backslash s/n)$, which is the logical type $n \rightarrow n \rightarrow s$ with the information that one argument comes before the verb and the other one after. But most approaches to syntax and semantics, including (Montague, 1974), introduce predicate categories as primitives rather than as function types. Thus transitive verbs are a category of its own, related to logic via a semantic rule. This gives more expressive power, as it permits predicates with different syntactic properties and variable word order (e.g. inversion in questions). A drawback is that a grammar may need a large number of categories and rules. In GPSG (Gazdar et al., 1985), and later in HPSG (Pollard and Sag, 1994), this is solved by introducing a feature called **subcat** for verbs. Verbs taking different arguments differ in the subcat feature but share otherwise the characteristic of being verbs.

In this paper, we will study the syntax and semantics of predication in GF, Grammatical Framework (Ranta, 2011). We will generalize both over subcategories (as in GPSG and HPSG), and over languages (as customary in GF). We use **dependent types** to control the application of verbs to legitimate arguments, and **records** to control the placement of arguments in sentences. The record structure is inspired by the **topological model** of syntax in (Diderichsen, 1962).

The approach is designed to apply to all languages in the GF Resource Grammar Library (RGL, (Ranta, 2009)), factoring out their typological differences in a modular way. We have tested the grammar with four languages from three families: Chinese, English, Finnish, and Swedish. As the implementation reuses old RGL code for all parts but predication, it can be ported to new languages with just a few pages of new GF code. We have also tested it in wide coverage tasks, with a probabilistic tree model and a lexicon of 60,000 lemmas.

We will start with an introduction to the abstraction mechanisms of GF and conclude with a summary of some recent research. Section 2 places GF on the map of grammar formalisms. Section 3 works out an example showing how abstract syntax can be shared between languages. Section 4 shows how parts of concrete syntax can be shared as well. Section 5 gives the full picture of predication with dependent types and records, also addressing extraction, coordination, and semantics. Section 6 gives preliminary evaluation. Section 7 concludes.

2 GF: an executive summary

GF belongs to a subfamily of categorial grammars inspired by (Curry, 1961). These grammars make a distinction between **tectogrammar**, which specifies the syntactic **structures** (tree-like representations), and **phenogrammar**, which relates these structures to linear representations, such as sequences of characters, words, or phonemes. Other formalisms in this family include ACG (de Groote, 2001) and Lambda grammars (Muskens, 2001).

GF inherits its name from LF, **Logical Frameworks**, which are type theories used for defining logics (Harper et al., 1993). GF builds on the LF called ALF, Another Logical Framework (Magnusson, 1994), which implements Martin-Löf's **higher-level type theory** (first introduced in the preface of (Martin-Löf, 1984); see Chapter 8 of (Ranta, 1994) for more details). Before GF was introduced as an independent formalism in 1998, GF-like applications were built as plugins to ALF (Ranta, 1997). The idea was that the LF defines the tectogrammar, and the plug-in defines the phenogrammar. The intended application was natural language interfaces to formal proof systems, in the style of (Coscoy et al., 1995).

GF was born via two additions to the natural language interface idea. The first one was **multilinguality**: one and the same tectogrammar can be given multiple phenogrammars. The second addition was **parsing**: the phenogrammar, which was initially just **linearization** (generating strings from type theoretical formulas), was reversed to rules that parse natural language into type theory. The result was a method for **translation**, which combines parsing the source language with linearization into the target language. This idea was indeed suggested in (Curry, 1961), and applied before GF in the Rosetta project (Landsbergen, 1982), which used Montague's **analysis trees** as tectogrammar.

GF can be seen as a formalization and generalization of Montague grammar. Formalization, because it introduces a formal notation for the linearization rules that in Montague's work were expressed informally. Generalization, because of multilinguality and also because the type system for analysis trees has dependent types.

Following the terminology of programming language theory, the tectogrammar is in GF called the **abstract syntax** whereas the phenogrammar is called the **concrete syntax**. As in compilers and logical frameworks, the abstract syntax encodes the structure relevant for **semantics**, whereas the concrete syntax defines "syntactic sugar".

The resulting system turned out to be equivalent to **parallel multiple context-free grammars** (Seki et al., 1991) and therefore parsable in polynomial time (Ljunglöf, 2004). Comprehensive grammars have been written for 29 languages, and later work has optimized GF parsing and also added probabilistic disambiguation and robustness, resulting in state-of-the-art performance in wide-coverage deep parsing (Angelov, 2011; Angelov and Ljunglöf, 2014).

3 Example: subject-verb-object sentences

Let us start with an important special case of predication: the subject-verb-object structure. The simplest possible rule is

```
fun PredTV : NP -> TV -> NP -> S
```

that is, a **function** that takes a subject NP, a transitive verb TV, and an object NP, and returns a sentence S. This function builds abstract syntax trees. Concrete syntax defines **linearization rules**, which convert trees into strings. The above rule can give rise to different word orders, such as SVO (as in English), SOV (as in Hindi), and VSO (as in Arabic):

```
lin PredTV s v o = s ++ v ++ o
lin PredTV s v o = s ++ o ++ v
lin PredTV s v o = v ++ s ++ o
```

where ++ means concatenation.

The above rule builds a sentence in one step. A more flexible approach is to do it in two steps: **complementation**, forming a VP (verb phrase) from the verb and the object, and **predication** proper that provides the subject. The abstract syntax is

```
fun Compl : TV -> NP -> VP
fun Pred  : NP -> VP -> S
```

These functions are easy to linearize for the SVO and SOV orders:

```
lin Compl v o = v ++ o     -- SVO
lin Compl v o = o ++ v     -- SOV
lin Pred s vp = s ++ vp    -- both
```

where `--` marks a comment. However, the VSO order cannot be obtained in this way, because the two parts of the VP are separated by the subject. The solution is to generalize linearization from strings to **records**. Complementation can then return a record that has the verb and the object as separate **fields**. Then we can also generate VSO:

```
lin Compl v o = {verb = v ; obj = o}
lin Pred s vp = vp.verb ++ s ++ vp.obj
```

The dot (`.`) means **projection**, picking the value of a field in a record.

Records enable the abstract syntax to abstract away not only from word order, but also from whether a language uses **discontinuous constituents**. VP in VSO languages is one example. Once we enable discontinuous constituents, they turn out useful almost everywhere, as they enable us to delay the decision about linear order. It can then be varied even inside a single language, if it depends on syntactic context (as e.g. in German; cf. (Müller, 2004) for a survey).

The next thing to abstract away from is **inflection** and **agreement**. Given the lexicon

```
fun We, She : NP
fun Love : TV
```

we can build the abstract syntax tree

```
Pred We (Compl Love She)
```

to represent *we love her*. If we swap the subject and the object, we get

```
Pred She (Compl Love We)
```

for *she loves us*. Now, these two sentences are built from the same abstract syntax objects, but no single word is shared between them! This is because the noun phrases **inflect** for **case** and the verb **agrees** to the subject.

In contrast to English, Chinese just reorders the words:

women ai ta - "we love her"

ta ai women - "she loves us"

Thus the above rules for SVO languages work as they are for Chinese. But in English, we must include case and agreement as **features** in the concrete syntax. Thus the linearization of an NP is a record that includes a **table** producing the case forms, and agreement as an **inherent feature**:

```
lin She = {
  s = table {
    Nom => "she" ;
    Acc => "her"
    } ;
  a = {n = Sg ; p = P3} ;
  }
```

The agreement feature (field `a`) is itself a record, with a number and a gender. In other languages, case and agreement can of course have different sets of values.

Verbs likewise include tables that inflect them for different agreement features:

```
lin Love = {
  s = table {
    {n = Sg ; p = P3} => "loves" ;
    _ => "love"
   }
  }
```

We can now define English linearization:

```
lin Compl v o =
  {s = table {a => v.s ! a ++ o.s ! Acc}}
lin Pred s vp =
  {s = s.s ! Nom ++ vp.s ! np.a}
```

using the same type of records for VP as for TV, and a one-string record for S. The `Compl` rule passes the agreement feature to the verb of the VP, and selects the Acc form of the object (with `!` denoting **selection** from a table). The `Pred` rule selects the `Nom` form of the subject, and attaches to this the VP form selected for `np.a`, i.e. the agreement feature of the subject.

4 Generalized concrete syntax

To see the full power of GF, we now take a look at its type and module system. Figure 1 shows a complete set of grammar modules implementing transitive verb predication for Finnish and Chinese with a maximum of shared code.

The first module in Figure 1 is the abstract syntax `Pred`, where the `fun` rules are preceded by a set of `cat` rules defining the **categories** of the grammar, i.e. the basic types. `Pred` defines five categories: S, Cl, NP, VP, and TV. S is the top-level category of sentences, whereas Cl (**clause**) is the intermediate category of predications, which can be used as sentences in many ways—here, as declaratives and as questions.

The concrete syntax has corresponding `lincat` rules, which equip each category with a **linearization type**, i.e. the type of the values returned when linearizing trees of that category. The module `PredFunctor` in Figure 1 contains four such rules. In `lincat NP`, the type `Case => Str` is the type of tables that produce a string as a function of a case, and `Agr` is the type of agreement features.

When a GF grammar is compiled, each `lin` rule is type checked with respect to the `lincat`s of the categories involved, to guarantee that, for every

$$\text{fun } f : C_1 \rightarrow \cdots \rightarrow C_n \rightarrow C$$

we have

$$\text{lin } f : C_1^* \rightarrow \cdots \rightarrow C_n^* \rightarrow C^*$$

```
abstract Pred = {
  cat S ; Cl ; NP ; VP ; TV ;
  fun Compl : TV -> NP -> VP ;  fun Pred : TV -> NP -> Cl ;
  fun Decl : Cl -> S ; fun Quest : Cl -> S ;
  }
incomplete concrete PredFunctor of Pred = open PredInterface in {
  lincat S = {s : Str} ;   lincat Cl = {subj,verb,obj : Str} ;
  lincat NP = {s : Case => Str ; a : Agr} ;
  lincat VP = {verb : Agr => Str ; obj : Str} ;  lincat TV = {s : Agr => Str} ;
  lin Compl tv np = {verb = tv.s ; obj = np.s ! objCase} ;
  lin Pred np vp = {subj = np.s !subjCase ; verb = vp.verb ! np.a ; obj = vp.obj} ;
  lin Decl cl = {s = decl cl.subj cl.verb cl.obj} ;
  lin Quest cl = {s = quest cl.subj cl.verb cl.obj} ;
  }
interface PredInterface = {
  oper Case, Agr : PType ;
  oper subjCase, objCase : Case ;
  oper decl, quest : Str -> Str -> Str -> Str ;
  }
instance PredInstanceFin of PredInterface = {          concrete PredFin of Pred =
  oper Case = -- Nom | Acc | ... ;                          PredFunctor with
  oper Agr = {n : Number ; p : Person} ;                      (PredInterface =
  oper subjCase = Nom ; objCase = Acc ;                          PredInstanceFin) ;
  oper decl s v o = s ++ v ++ o ;
  oper quest s v o = v ++ "&+ ko" ++ s ++ o ;
  }
instance PredInstanceChi of PredInterface = {          concrete PredChi of Pred =
  oper Case, Agr = {} ;                                     PredFunctor with
  oper subjCase, objCase = <> ;                               (PredInterface =
  oper decl s v o = s ++ v ++ o ;                                PredInstanceChi) ;
  oper quest s v o = s ++ v ++ o ++ "ma" ;
  }
```

Figure 1: Functorized grammar for transitive verb predication.

where A^* is the linearization type of A. Thus linearization is a **homomorphism**. It is actually an instance of denotational semantics, where the `lincat`s are the **domains of possible denotations**.

Much of the strength of GF comes from using different linearization types for different languages. Thus English needs case and agreement, Finnish needs many more cases (in the full grammar), Chinese needs mostly only strings, and so on. However, it is both useful and illuminating to unify the types. The way to do this is by the use of **functors**, also known as a **parametrized modules**.

`PredFunctor` in Figure 1 is an example; functors are marked with the keyword `incomplete`. A functor depends on an `interface`, which declares a set of parameters (`PredInterface` in Figure 1). A concrete module is produced by giving an `instance` to the interface (`PredInstanceFin` and `PredInstanceChi`).

The rules in `PredFunctor` in Figure 1 are designed to work for both languages, by varying the definitions of the constants in `PredInterface`. And more languages can be added to use it. Consider for example the definition of NP. The experience from the RGL shows that, if a language has case and agreement, its NPs inflect for case and have inherent agreement. The limiting case of Chinese can be treated by using the unit type ({} i.e. the record type with no fields) for both features. This would not be so elegant for Chinese alone, but makes sense in the code sharing context.

Discontinuity now appears as another useful generalization. With the `lincat` definition in `PredFunctor`, we can share the `Compl` rule in all of the languages discussed so far. In clauses (Cl), we continue on similar lines: we keep the subject, the verb, and the object on separate fields. Notice that `verb` in `Cl` is a plain string, since the value of Agr gets fixed when the subject is added.

The final sentence word order is created as the last step, when converting `Cl` into `S`. As `Cl` is discontinuous, it can be linearized in different orders. In Figure 1, this is used in Finnish for generating the SVO order in declaratives and VSO on questions (with an intervening question particle *ko* glued to the verb). It also supports the other word orders of Finnish (Karttunen and Kay, 1985).

By using an abstract syntax in combination with unordered records, parameters, and functors for the concrete syntax, we follow a kind of a "principles and parameters" approach to language variation (Chomsky, 1981). The actual parameter set for the whole RGL is of course larger than the one shown here.

Mathematically, it is possible to treat *all* differences in concrete syntax by parameters, simply by declaring a new parameter for every `lincat` and `lin` rule! But this is both vacuous as a theory and an unnecessary detour in practice. It is more illuminating to keep the functor simple and the set of parameters small. If the functor does not work for a new language, it usually makes more sense to **override** it than to grow the parameter list, and GF provides a mechanism for this. Opposite to "principles and parameters", this is "a model in which language-particular rules take over the work of parameter settings" (Newmeyer, 2004). A combination of the two models enables language comparison by measuring the amount of overrides.

5 The full predication system

So far we have only dealt with one kind of verbs, TV. But we need more: intransitive, ditransitive, sentence-complement, etc. The general verb category is a dependent type, which varies over argument type lists:

```
cat V (x : Args)
```

The list `x : Args` corresponds to the subcat feature in GPSG and HPSG. Verb phrases and clauses have the same dependencies. Syntactically, a phrase depending on `x : Args` has "holes" for every argument in the list `x`. Semantically, it is a function over the denotations of its arguments (see Section 5.3 below).

5.1 The code

Figure 2 shows the essentials of the resulting grammar, and we will now explain this code. The full code is available at the GF web site.

1. **Argument lists and dependent categories**. The argument of a verb can be an adjectival phrase (AP, *become old*), a clause (Cl, *say that we go*), a common noun (CN, *become a president*), a noun phrase (NP, *love her*), a question (QCl, *wonder who goes*), or a verb phrase (VP, *want to go*). The definition allows an arbitrary list of arguments. For example, NP+QCl is used in verbs such as *ask* (someone whether something).

What about PP (prepositional phrase) complements? The best approach in a multilingual setting is to treat them as NP complements with designated cases. Thus in Figure 2.5, the linearization type of VP has fields of type `complCase`. This covers cases and prepositions, often in combination. For instance, the German verb *lieben* ("love") takes a plain accusative argument, *folgen* ("love") a plain dative, and *warten* ("wait") the preposition *auf* with the accusative. From the abstract syntax point of view, all of them are NP-complement verbs. Cases and prepositions, and thereby transitivity, are defined in concrete syntax.

The category `Cl`, clause, is the discontinuous structure of sentences before word order is determined. Its instance `Cl (c np O)` corresponds to the **slash categories** `S/NP` and `S/PP` in GPSG. Similarly, `VP (c np O)` corresponds to `VP/NP` and `VP/PP`, `Adv (c np O)` to `Adv/NP` (prepositions), and so on.

2. **Initial formation of verb phases**. A VP is formed from a V by fixing its tense and polarity. In the resulting VP, the `verb` depends only on the agreement features of the expected subject. The complement case comes from the verb's lexical entry, but the other fields—such as the objects—are left empty. This makes the VP usable in both complementation and slash operations (where the subject is added before some complement).

VPs can also be formed from adverbials, adjectival phrases, and common nouns, by adding a copula. Thus *was in* results from applying `UseAdv` to the preposition (i.e. `Adv/NP`) *in*, and expands to a VP with `ComplNP` (*was in France*) and to a slash clause with `PredVP` (*she was in*).

3. **Complementation, VP slash formation, reflexivization**. The `Compl` functions in Figure 2.3 provide each verb phrase with its "first" complement. The `Slash` functions provide the "last" complement, leaving a "gap" in the middle. For instance, `SlashCl` provides the slash clause used in the question *whom did you tell that we sleep*. The `Refl` rules fill argument places with reflexive pronouns.

4. **NP-VP predication, slash termination, and adverbial modification**. `PredVP` is the basic NP-VP predication rule. With `x = c np O`, it becomes the rule that combines NP with VP/NP to form S/NP. `SlashTerm` is the GPSG "slash termi-

1. **Argument lists and some dependent categories**

```
cat Arg ; Args                      -- arguments and argument lists
fun ap, cl, cn, np, qcl, vp : Arg  -- AP, Cl, CN, NP, QCl, VP argument
fun O : Args                        -- no arguments
fun c : Arg -> Args -> Args         -- one more argument

cat V   (x : Args)                  -- verb in the lexicon
cat VP  (x : Args)                  -- verb phrase
cat Cl  (x : Args)                  -- clause
cat AP  (x : Args)                  -- adjectival phrase
cat CN  (x : Args)                  -- common noun phrase
cat Adv (x : Args)                  -- adverbial phrase
```

2. **Initial formation of verb phases**

```
fun UseV   : (x : Args) -> Temp -> Pol -> V x   -> VP x  -- loved (X)
fun UseAP  : (x : Args) -> Temp -> Pol -> AP x  -> VP x  -- was married to (X)
fun UseCN  : (x : Args) -> Temp -> Pol -> CN x  -> VP x  -- was a son of (X)
fun UseAdv : (x : Args) -> Temp -> Pol -> Adv x -> VP x  -- was in (X)
```

3. **Complementation, VP slash formation, reflexivization**

```
fun ComplNP : (x : Args) -> VP (c np x)        -> NP   -> VP x        -- love her
fun ComplCl : (x : Args) -> VP (c cl x)        -> Cl x -> VP x        -- say that we go
fun SlashNP : (x : Args) -> VP (c np (c np x)) -> NP   -> VP (c np x) -- show (X) to him
fun SlashCl : (x : Args) -> VP (c np (c cl x)) -> Cl x -> VP (c np x) -- tell (X) that..
fun ReflVP  : (x : Args) -> VP (c np x)                -> VP x        -- love herself
fun ReflVP2 : (x : Args) -> VP (c np (c np x))         -> VP (c np x) -- show (X) to herself
```

4. **NP-VP predication, slash termination, and adverbial modification**

```
fun PredVP    : (x : Args) -> NP          -> VP x  -> Cl x            -- she loves (X)
fun SlashTerm : (x : Args) -> Cl (c np x) -> NP    -> Cl x            -- she loves + X
```

5. **The functorial linearization type of VP**

```
lincat VP = {
  verb   : Agr => Str * Str * Str ;    -- finite: would,have,gone
  inf    : VVType => Str ;             -- infinitive: (not) (to) go
  imp    : ImpType => Str ;            -- imperative: go
  c1     : ComplCase ;                 -- case of first complement
  c2     : ComplCase ;                 -- case of  second complement
  vvtype : VVType ;                    -- type of VP complement
  adj    : Agr => Str ;                -- adjective complement
  obj1   : Agr => Str ;                -- first complement
  obj2   : Agr => Str ;                -- second complement
  objagr : {a : Agr ; objCtr : Bool} ; -- agreement used in object control
  adv1   : Str ;                       -- pre-verb adverb
  adv2   : Str ;                       -- post-verb adverb
  ext    : Str ;                       -- extraposed element e.g. that-clause
  }
```

6. **Some functorial linearization rules**

```
lin ComplNP  x vp np = vp ** {obj1 = \\a => appComplCase vp.c1 np}
lin ComplCl  x vp cl = vp ** {ext  = that_Compl ++ declSubordCl cl}
lin SlashNP2 x vp np = vp ** {obj2 = \\a => appComplCase vp.c2 np}
lin SlashCl  x vp cl = vp ** {ext  = that_Compl ++ declSubordCl cl}
```

7. **Some interface parameters**

```
oper Agr, ComplCase : PType              -- agreement, complement case
oper appComplCase : ComplCase -> NP -> Str  -- apply complement case to NP
oper declSubordCl : Cl -> Str             -- subordinate question word order
```

Figure 2: Dependent types, records, and parameters for predication.

nation" rule.

5. **The functorial linearization type of VP**. This record type contains the string-valued fields that can appear in different orders, as well as the inherent features that are needed when complements are added. The corresponding record for `Cl` has similar fields with constant strings, plus a subject field.

6. **Some functorial linearization rules**. The verb-phrase expanding rules typically work with **record updates**, where the old VP is left unchanged except for a few fields that get new values. GF uses the symbol `**` for record updates. Notice that `ComplCl` and `SlashCl` have exactly the same linearization rules; the difference comes from the argument list *x* in the abstract syntax.

7. **Some interface parameters**. The code in Figure 2.5 and 2.6 is shared by different languages, but it depends on an interface that declares parameters, some of which are shown here.

5.2 More constructions

Extraction. The formation of questions and relatives is straighforward. Sentential (yes/no) questions, formed by `QuestCl` in Figure 3.1, don't in many languages need any changes in the clause, but just a different ordering in final linearization. Wh questions typically put one interrogative (IP) in the focus, which may be in the beginning of the sentence even though the corresponding argument place in declaratives is later. The `focus` field in `QCl` is used for this purpose. It carries a Boolean feature saying whether the field is occupied. If its value is True, the next IP is put into the "normal" argument place, as in *who loves whom*.

Coordination. The VP conjunction rules in Figure 3.2 take care of both intransitive VPs (*she walks and runs*) and of verb phrases with arguments (*she loves and hates us*). Similarly, Cl conjuction covers both complete sentences and slash clauses (*she loves and we hate him*). Some VP coordination instances may be ungrammatical, in particular with inverted word orders. Thus *she is tired and wants to sleep* works as a declarative, but the question is not so good: ?*is she tired and wants to sleep*. Preventing this would need a much more complex rules. Since the goal of our grammar is not to define grammaticality (as in formal language theory), but to analyse and translate existing texts, we opted for a simple system in this case (but did not need to do so elsewhere).

5.3 Semantics

The abstract syntax has straightforward denotational semantics: each type in the `Args` list of a category adds an argument to the type of denotations. For instance, the basic VP denotation type is `Ent -> Prop`, and the type for an arbitrary subcategory of `VP x` is

```
(x : Args) -> Den x (Ent -> Prop)
```

where `Den` is a type family defined recursively over `Args`,

```
Den : Args -> Type -> Type
Den 0 t = t
Den (c np xs) t = Ent  -> Den xs t
Den (c cl xs) t = Prop -> Den xs t
```

and so on for all values of `Arg`. The second argument *t* varies over the basic denotation types of VP, AP, Adv, and CN.

Montague-style semantics is readily available for all rules operating on these categories. As a logical framework, GF has the expressive power needed for defining semantics (Ranta, 2004). The types can moreover be extended to express **selectional restrictions**, where verb arguments are restricted to **domains of individuals**. Here is a type system that adds a domain argument to NP and VP:

```
cat NP (d : Dom)
cat VP (d : Dom)(x : Args)
fun PredVP : (d : Dom) -> (x : Args)
                 -> NP d -> VP d x -> Cl x
```

The predication rule checks that the NP and the VP have the same domain.

6 Evaluation

Coverage. The dependent type system for verbs, verb phrases, and clauses is a generalization of the old Resource Grammar Library (Ranta, 2009), which has a set of hard-wired verb subcategories and a handful of slash categories. While it covers "all usual cases", many logically possible ones are missing. Some such cases even appear in the Penn treebank (Marcus et al., 1993), requiring extra rules in the GF interpretation of the treebank (Angelov, 2011). An example is a function of type

```
V (c np (c vp 0)) ->
              VPC (c np 0) -> VP (c np 0)
```

which is used 12 times, for example in *This is designed to get the wagons in a circle and defend the smoking franchise*. It has been easy to write conversion rules showing that the old coverage is preserved. But it remains future work to see what new cases are covered by the increased generality.

1. **Extraction**.

```
cat QCl (x : Args)                                          -- question clause
cat IP                                                      -- interrogative phrase
fun QuestCl    : (x : Args)       -> Cl x         -> QCl x   -- does she love him
fun QuestVP    : (x : Args) -> IP -> VP x         -> QCl x   -- who loves him
fun QuestSlash : (x : Args) -> IP -> QCl (c np x) -> QCl x   -- whom does she love

lincat QCl = Cl ** {focus : {s : Str ; isOcc : Bool}}       -- focal IP, whether occupied
```

2. **Coordination**.

```
cat VPC (x : Args)                                          -- VP conjunction
cat ClC (x : Args)                                          -- Clause conjunction
fun StartVPC : (x : Args) -> Conj -> VP x -> VP x  -> VPC x  -- love or hate
fun ContVPC  : (x : Args)         -> VP x -> VPC x -> VPC x  -- admire, love or hate
fun UseVPC   : (x : Args) -> VPC x                 -> VP  x  -- [use VPC as VP]
fun StartClC : (x : Args) -> Conj -> Cl x -> Cl x  -> ClC x  -- he sells and I buy
fun ContClC  : (x : Args) -> Cl x -> ClC x         -> ClC x  -- you steal, he sells and I buy
fun UseClC   : (x : Args) -> ClC x                 -> Cl  x  -- [use ClC as Cl]
```

Figure 3: Extraction and coordination.

Multilinguality. How universal are the concrete syntax functor and interface? In the standard RGL, functorization has only been attempted for families of closely related languages, with Romance languages sharing 75% of syntax code and Scandinavian languages 85% (Ranta, 2009). The new predication grammar shares code across all languages. The figure to compare is the percentage of shared code (abstract syntax + functor + interface) of the total code written for a particular language (shared + language-specific). This percentage is 70 for Chinese, 64 for English, 61 for Finnish, and 76 for Swedish, when calculated as lines of code. The total amount of shared code is 760 lines. One example of overrides is negation and questions in English, which are complicated by the need of auxiliaries for some verbs (*go*) but not for others (*be*). This explains why Swedish shares more of the common code than English.

Performance. Dependent types are not integrated in current GF parsers, but checked by post-processing. This implies a loss of speed, because many trees are constructed just to be thrown away. But when we specialized dependent types and rules to nondependent instances needed by the lexicon (using them as **metarules** in the sense of GPSG), parsing became several times faster than with the old grammar. An analysis remains to do, but one hypothesis is that the speed-up is due to fixing tense and polarity earlier than in the old RGL: when starting to build VPs, as opposed to when using clauses in full sentences. Dependent types made it easy to test this refactoring, since they reduced the number of rules that had to be written.

Robustness. Robustness in GF parsing is achieved by introducing **metavariables** ("question marks") when tree nodes cannot be constructed by the grammar (Angelov, 2011). The subtrees under a metavariable node are linearized separately, just like a sequence of **chunks**. In translation, this leads to decrease in quality, because dependencies between chunks are not detected. The early application of tense and polarity is an improvement, as it makes verb chunks contain information that was previously detected only if the parser managed to build a whole sentence.

7 Conclusion

We have shown a GF grammar for predication allowing an unlimited variation of argument lists: an abstract syntax with a concise definition using dependent types, a concrete syntax using a functor and records, and a straightforward denotational semantics. The grammar has been tested with four languages and shown promising results in speed and robustness, also in large-scale processing. A more general conclusion is that dependent types, records, and functors are powerful tools both for computational grammar engineering and for the theoretical study of languages.

Acknowledgements. I am grateful to Krasimir Angelov and Robin Cooper for comments, and to Swedish Research Council for support under grant nr. 2012-5746 (Reliable Multilingual Digital Communication).

References

K. Angelov and P. Ljunglöf. 2014. Fast statistical parsing with parallel multiple context-free grammars. In *Proceedings of EACL-2014, Gothenburg*.

K. Angelov. 2011. *The Mechanics of the Grammatical Framework*. Ph.D. thesis, Chalmers University of Technology.

Y. Bar-Hillel. 1953. A quasi-arithmetical notation for syntactic description. *Language*, 29:27–58.

N. Chomsky. 1981. *Lectures on Government and Binding*. Mouton de Gruyter.

Y. Coscoy, G. Kahn, and L. Thery. 1995. Extracting text from proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Second Int. Conf. on Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 109–123.

H. B. Curry. 1961. Some logical aspects of grammatical structure. In Roman Jakobson, editor, *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

Ph. de Groote. 2001. Towards Abstract Categorial Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Toulouse, France*, pages 148–155.

P. Diderichsen. 1962. *Elementær dansk grammatik*. Gyldendal, København.

G. Gazdar, E. Klein, G. Pullum, and I. Sag. 1985. *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford.

R. Harper, F. Honsell, and G. Plotkin. 1993. A Framework for Defining Logics. *JACM*, 40(1):143–184.

L. Karttunen and M. Kay. 1985. Parsing in a free word order language. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Parsing, Psychological, Computational, and Theoretical Perspectives*, pages 279–306. Cambridge University Press.

J. Lambek. 1958. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170.

J. Landsbergen. 1982. Machine translation based on logically isomorphic Montague grammars. In *COLING-1982*.

P. Ljunglöf. 2004. *The Expressivity and Complexity of Grammatical Framework*. Ph.D. thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University. `http://www.cs.chalmers.se/~peb/pubs/p04-PhD-thesis.pdf`.

L. Magnusson. 1994. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph.D. thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg.

M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

P. Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis, Napoli.

R. Montague. 1974. *Formal Philosophy*. Yale University Press, New Haven. Collected papers edited by Richmond Thomason.

S. Müller. 2004. Continuous or Discontinuous Constituents? A Comparison between Syntactic Analyses for Constituent Order and Their Processing Systems. *Research on Language and Computation*, 2(2):209–257.

R. Muskens. 2001. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam. `http://let.uvt.nl/general/people/rmuskens/pubs/amscoll.pdf`.

F. J. Newmeyer. 2004. Against a parameter-setting approach to language variation. *Linguistic Variation Yearbook*, 4:181–234.

C. Pollard and I. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

A. Ranta. 1994. *Type Theoretical Grammar*. Oxford University Press.

A. Ranta. 1997. Structures grammaticales dans le français mathématique. *Mathématiques, informatique et Sciences Humaines*, 138/139:5–56/5–36.

A. Ranta. 2004. Computational Semantics in Type Theory. *Mathematics and Social Sciences*, 165:31–57.

A. Ranta. 2009. The GF Resource Grammar Library. *Linguistics in Language Technology*, 2. `http://elanguage.net/journals/index.php/lilt/article/viewFile/214/158`.

A. Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford.

H. Seki, T. Matsumura, M. Fujii, and T. Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.