Computational Semantics in Type Theory

Aarne Ranta*
Department of Computing Science,
Chalmers University of Technology and the University of Gothenburg
aarne@cs.chalmers.se

December 17, 2003

Abstract. This paper aims to show how Montague-style grammars can be completely formalized and thereby declaratively implemented by using the Grammatical Framework GF. The implementation covers the fundamental operations of Montague's PTQ model: the construction of analysis trees, the linearization of trees into strings, and the interpretation of trees as logical formulas. Moreover, a parsing algorithm is derived from the grammar. Given that GF is a constructive type theory with dependent types, the technique extends from classical Montague grammars to ones in which the Curry-Howard isomorphism is used to explain anaphoric reference. On the other hand, GF has a built-in compositionality requirement that is stronger than in PTQ and prevents us from formulating quantifying-in rules of Montague style. This leads us to alternative formulations of such rules in terms of combinators and discontinuous constituents. The PTQ fragment will moreover be presented as an exmple of how a GF grammar is modified by replacing English with another target language, French. The paper concludes by a discussion of the complementary rôles of logically and linguistically oriented syntax.

1 Introduction

Montague grammar [14] is the corner stone of most work in the computational semantics of natural language. The "grammar" is actually a series of papers written in the late 1960's and early 1970's, showing how certain fragments of English are analysed syntactically and semantically. Montague's syntax was special-designed for the purpose of semantics, which gave rise to a series of attempts by linguists to combine his semantics with a more familiar kind of syntax [16, 4, 8]. As a consequence, what is found in modern linguistic theory under the title of Montague grammar is seldom a uniform theory, but rather a bundle of ideas, inspired by Montague and reused in another context.

In computational linguistics, some attempts have been made towards a uniform treatment of Montague grammar. The most prominent implementations are

*This paper is based on material used at a graduate course in formal semantics at the Linguistics Department at the University of Gothenburg in Spring 2001. The author is grateful to the participants of the course for stimulating discussions, and to Pascal Boldini and Robin Cooper for valuable comments on the paper. The work was partly financed from grant 2002-4879, *Records, Types and Computational Dialogue Semantics*, from Vetenskapsrådet.

written in Prolog [17, 1], to illustrate the suitability of the Prolog programming language [3] for computational linguistics in general and for computational semantics in particular. The main advantages of Prolog are its *declarativity*—implementation and theory are not separated—and its built-in support for *syntactic analysis*, in the form of definite clause grammars and their parsing algorithm.

However, Prolog lacks some of the central features of the logical theory that Montague used: types, functions, and variable bindings. These features have to be implemented separately,[1] and to do so Prolog is of course as adequate as any other general-purpose programming language. But, on the other hand, the very extension of Prolog into a general-purpose programming language, with "impure" features going beyond its logic-programming core, weaken the claim of declarativity, and thereby the status of Prolog as the language of choice for implementing Montague grammar.

Besides Prolog, *logical frameworks* [27, 9, 12] are another approach combining logic and programming. Most logical frameworks are based on some version of *constructive type theory*, which, despite its name, is an extension rather than a restriction of classical type theory. In particular, they include as special cases the lambda calculus and the simple type hierarchy that were used by Montague. For the implementation of Montague's semantics, logical frameworks thus give more support than Prolog, but syntactic analysis is not supported. Now, since a logical framework can be used as a full-scale functional programming language, it would be possible to implement the syntax part of Montague grammar by writing functional programs for parsing and generation. But the result would of course no longer be a declarative definition of the grammar.

Grammatical Framework (GF) [22, 21] is an extension of logical frameworks with built-in support for syntax. The purpose of this paper is to show how to implement Montague-style grammars in GF, including not only Montague's own PTQ fragment, but also an extension using *dependent types* [19], as well as a grammar of French.[2]

The contents of this paper

We begin with a formalization of a small Montague-style grammar in type theory, in a classical setting, i.e. without using specifically constructive type-theoretical concepts (Section 2). Then we will introduce dependent types and give a generalization of the classical grammar, together with a model in Martin-Löf's type theory [13] (Section 3). This will lead us to a compositional grammar of anaphora (Section 4). We conclude the theoretical discussion with the so-called quantifying-in rules (Section 5).

The presentation of grammatical and logical rules will be strictly formal, using the notation of GF[3]. Thus the rules can be directly used in the GF interpreter to

[1] As an alternative, the semantics of variable bindings is sometimes changed so that Prolog's so-called logical variables can be used.

[2] We have previously studied a Montague-style type-theoretical grammar of French in [20]. GF can be seen as a formal theory of the concrete syntax which in that paper was defined in type theory using all available means.

[3] All typeset GF judgements of this paper have been produced from actual GF code by a

perform computations, to parse natural-language expressions, and to edit grammatical objects interactively. We hope to demonstrate the usability of GF for implementing Montague-style grammars, independently of the constructive extensions that GF makes available. This demonstration is completed by two supplementary examples: a fairly complete formalization of Montague's PTQ fragment (Section 6) and a parallel concrete syntax for French (Section 7). We conclude by making a distinction between application grammars and resource grammars, which clarifies the complementary rôles of logically and linguistically oriented grammar descriptions (Section 8).

The paper presupposes some knowledge of Montague grammar [14], type-theoretical grammar [19], and GF [22, 21]. Since we focus on semantics, we keep syntactic structures as few and as simple as possible.

2 Formalizing Montague grammar

Montague is duly praised as the one who brought logical rigour to the semantics of natural language. Interestingly, the main bulk of his work was carried out in the late 1960's in parallel with the work of Scott and Strachey in *denotational semantics* [24], partly in contact with Scott. What Montague did was, in a word, denotational semantics of English. This means, exactly like in computer science, that he defined an *abstract syntax* of the language, and for each syntax tree an *interpretation* (*denotation*) of the tree as a mathematical object of some appropriate type.

In logical frameworks, there is a firm tradition of implementing abstract syntax and denotational semantics. What is missing is *concrete syntax*—the relation between abstract syntax trees and linguistic string representations. Forms of judgement for defining concrete syntax are precisely what GF adds to logical frameworks. A set of such judgements is a declarative definition, from which the printing and parsing functions are automatically derived.

2.1 Levels of presentation and mappings between them

The main levels of representation of linguistic objects in Montague grammar are

strings: *John walks*,

analysis trees: $\mathbf{F}_4(\mathit{john}, \mathit{walk})$,

logical formulas: $\mathit{walk}^*(\mathit{john}^*)$,

model-theoretic objects: **True**.

In Montague's original presentation, there are four kinds of rules:

construction: how analysis trees are built,

linearization: how trees are translated into strings,

translation: how trees are translated into formulas,

interpretation: how formulas are mapped into model-theoretic objects.

`gf2latex` program.

Later work has also considered other operations:

parsing: how strings are translated into trees [7],

generation: how formulas are translated into trees [6].

It should be noticed that, while the original operations are functions, in the sense that they always give a unique result, these latter operations are search procedures: they may yield multiple results or even fail. Thus it is natural that these procedures do not belong to the definition of a grammar, even though their correctness is of course defined by the grammar. However, since these algorithms are among the most important applications of the grammar, it is desirable that they can be mechanically derived from it.

2.2 Abstract and concrete syntax

Following computer science terminology, we call the level of trees and their construction rules the *abstract syntax* of a language. The rules translating trees into strings are called the *concrete syntax*. A phrase structure grammar à la Chomsky [2] defines these two things simultaneously. For instance, the predication rule combining a verb phrase with a noun phrase to form a sentence,

$$S \longrightarrow NP\ VP$$

simultaneously defines the construction of an S tree from an NP tree and a VP tree, and the linearization of the tree by prefixing the linearization of the NP tree to the linearization of the VP tree. To tell apart these two rules, we write as follows in GF:

fun *Pred* : $NP \to VP \to S$

lin *Pred np vp* = $\{s = np.s ++ vp.s\}$

The **fun** judgement belongs to abstract syntax. It says that the *type* of the tree-forming constant *Pred* is the function type with the argument types *VP* and *VP* and the value *S*. The **lin** judgement belongs to concrete syntax. It says that any tree constructed by *Pred* is linearized by concatenating the linearizations of the two subtrees.

The use of a category symbol, such as *NP* above, presupposes that the category has been introduced in the abstract syntax. The judgement introducing the category *NP* is

cat *NP*

The linearization of a tree is not simply a string but a *record*, which contains information on inflection, gender, etc. What information there is depends on the *linearization type* corresponding to the type of the tree. We will start with the simplest kind of linearization types, which are record types with just one field, of type *Str*,

$$\{s : Str\}$$

The linearization type of a category is defined by a **lincat** judgement, for instance,

lincat $NP \ = \ \{s : Str\}$

Even though we in this paper try to work with as simple a concrete syntax as possible, we will later have to show some more complex linearization types. By using records, GF can handle complex linguistic objects such as inflection tables, agreement structures, and discontinuous constituents.[4]

A concrete syntax is *complete* w.r.t. an abstract syntax if it has for every **cat** judgement a corresponding **lincat** judgement, and for every **fun** judgement a corresponding **lin** judgement. The concrete syntax is *sound* if every **lin** judgement is well-typed w.r.t. the **lincat** judgements. Formally this can be expressed as follows: if a function f has been introduced by the judgement

$$\textbf{fun } f \ : \ A_1 \to \ldots \to A_n \to A$$

then the linearization rule of f must have the form[5]

$$\textbf{lin } f \ = \ t$$

where, denoting the linearization type of a type C by C°,

$$t \ : \ A_1^{\circ} \to \ldots \to A_n^{\circ} \to A^{\circ}.$$

Mathematically, an abstract syntax defines a free algebra of syntax trees. A concrete syntax defines a homomorphism from this algebra to a system of concrete-syntax objects (records of strings etc.).

2.3 A fragment of English

Let us now consider the full formalization of a small fragment of English. What we need first is a sequence of **cat** judgements telling what categories there are. The following judgements define the categories of sentence, noun phrase, verb phrase, transitive verb, common noun, and proper name:

cat *S*
cat *NP*
cat *VP*
cat *TV*
cat *CN*
cat *PN*

The next five judgements define the syntactic structures of predication (already shown above) and complementization, followed by three rules for forming noun phrases:

fun *Pred* : $NP \to VP \to S$
fun *Compl* : $TV \to NP \to VP$
fun *Every* : $CN \to NP$
fun *Indef* : $CN \to NP$
fun *Raise* : $PN \to NP$

[4]Notice that records are similar to *feature structures* in formalisms such as PATR [25] and HPSG [18]. The main difference is that GF records are obtained as linearizations of trees, whereas PATR and HPSG records are obtained as parses of strings.

[5]The notation **lin** $f \ x_1 \ldots x_m \ = \ t$ is syntactic sugar for **lin** $f \ = \ \lambda \, x_1 \ldots x_m \ \to \ t$.

Notice that, unlike in PTQ, we have a separate category PN of proper names and an explicit raising coercion to NP. This gives better modularity in semantics.

Finally, we need a lexicon, i.e. a list of atomic rules:

$$\begin{array}{l}\textbf{fun } \mathit{Man}, \mathit{Woman} \ : \ \mathit{CN} \\ \textbf{fun } \mathit{Love} \ : \ \mathit{TV} \\ \textbf{fun } \mathit{Walk} \ : \ \mathit{VP} \\ \textbf{fun } \mathit{John}, \mathit{Bill}, \mathit{Mary} \ : \ \mathit{PN}\end{array}$$

Syntax trees can now be built as type-theoretical terms. For instance, the term

$$\mathit{Pred}\ (\mathit{Every}\ \mathit{Woman})\ (\mathit{Compl}\ \mathit{Love}\ (\mathit{Raise}\ \mathit{Bill}))$$

is a syntax tree corresponding to the sentence

every woman loves Bill.

The relation between trees and strings is defined in the concrete syntax. First we have to assign a linearization type to every category. In the present example, it is enough to use the simplest possible type, i.e.

$$\textbf{lincat } \mathit{CN} \ = \ \{s \ : \ \mathit{Str}\}$$

and so on, for every category.

To make the linearization rules as simple as possible, we define two *auxiliary operations* in concrete syntax: one making a string into a record, and another one concatenating two records:

$$\begin{array}{l}\textbf{oper } \mathit{ss} : \mathit{Str} \to \{s \ : \ \mathit{Str}\} \ = \ \lambda s \to \{s \ = \ s\} \\ \textbf{oper } \mathit{cc2} : (\_, \_ : \{s \ : \ \mathit{Str}\}) \to \{s \ : \ \mathit{Str}\} \ = \ \lambda x, y \to \mathit{ss}\ (x.s \ +\!\!+ \ y.s)\end{array}$$

So the linearization rules are:

$$\begin{array}{l}\textbf{lin } \mathit{Pred}\ Q\ F \ = \ \mathit{cc2}\ Q\ F \\ \textbf{lin } \mathit{Compl}\ F\ Q \ = \ \mathit{cc2}\ F\ Q \\ \textbf{lin } \mathit{Every}\ A \ = \ \mathit{ss}\ (\text{“}\mathit{every}\text{”} \ +\!\!+ \ A.s) \\ \textbf{lin } \mathit{Indef}\ A \ = \ \mathit{ss}\ (\text{“}a\text{”} \ +\!\!+ \ A.s) \\ \textbf{lin } \mathit{Raise}\ a \ = \ a \\ \textbf{lin } \mathit{Man} \ = \ \mathit{ss}\ \text{“}\mathit{man}\text{”} \\ \textbf{lin } \mathit{Woman} \ = \ \mathit{ss}\ \text{“}\mathit{woman}\text{”} \\ \textbf{lin } \mathit{Love} \ = \ \mathit{ss}\ \text{“}\mathit{loves}\text{”} \\ \textbf{lin } \mathit{Walk} \ = \ \mathit{ss}\ \text{“}\mathit{walks}\text{”} \\ \textbf{lin } \mathit{John} \ = \ \mathit{ss}\ \text{“}\mathit{John}\text{”} \\ \textbf{lin } \mathit{Bill} \ = \ \mathit{ss}\ \text{“}\mathit{Bill}\text{”} \\ \textbf{lin } \mathit{Mary} \ = \ \mathit{ss}\ \text{“}\mathit{Mary}\text{”}\end{array}$$

What we have shown above is a complete GF grammar for a fragment of English. In the same way as Montague, the grammar has explicit statements of construction and linearization rules. The GF implementation moreover derives a parser for the grammar. In the present case, it is a simple context-free (even finite-state!) parser.

2.4 Predicate calculus

The next task is to define the logical formulas. Here we use GF as a logical framework, and declare the usual connectives and quantifiers. The syntactic categories are propositions and entities:

cat *Prop*
cat *Ent*

fun *And, Or, If* : *Prop* → *Prop* → *Prop*
fun *All, Exist* : (*Ent* → *Prop*) → *Prop*

Notice how quantifiers are treated as *higher-order functions*: they take as their argument a function from entities to propositions.

In ordinary logical frameworks, the users have to read and write formulas in a purely functional syntax. For instance,

$$All\ (\lambda x \rightarrow If\ (Woman\ x)\ (Exist\ (\lambda y \rightarrow And\ (Man\ y)\ (Love\ x\ y))))$$

is the purely functional notation for

$$(\forall x)(Woman(x) \supset (\exists y)(Man(y)\&Love(x, y))).$$

We can produce exactly this notation by making linearization rules produce LaTeX code:

oper *par* : *Str* → *Str* = λ*s* → "(" ++ *s* ++ ")"

lin *And A B* = *ss* (*par* (*A.s* ++ "\&" ++ *B.s*))
lin *Or A B* = *ss* (*par* (*A.s* ++ "*vee*" ++ *B.s*))
lin *If A B* = *ss* (*par* (*A.s* ++ "*supset*" ++ *B.s*))
lin *Not A* = *ss* ("*sim*" ++ *A.s*)
lin *All P* = *ss* (*par* ("*forall*" ++ *P.v*) ++ *P.s*)
lin *Exist P* = *ss* (*par* ("*exists*" ++ *P.v*) ++ *P.s*)

The record projection *P.v* refers to the bound variable of the function argument of the quantifiers. The linearization of an expression of a function type (such as *Ent* → *Prop*) is a record of the type

$$\left\{ \begin{array}{lcl} v & : & Str \\ s & : & Str \end{array} \right\}$$

where the *v* field stores the bound variable and the *s* field the function body. The function has to be in an η-expanded form, i.e. have a λ-bound variable for each argument type.

Now the above formula is expressed

$$(\forall x)(Woman(x) \supset (\exists y)(Man(y)\&Love(x, y))).$$

as desired.

2.5 Translation into predicate calculus

We define for each category an *interpretation function* that takes terms of that category into a type. The value type is what Montague called the *domain of possible denotations* of the category.

$$\begin{array}{lll}
\textbf{fun}\ iS & : & S \to Prop \\
\textbf{fun}\ iNP & : & NP \to (Ent \to Prop) \to Prop \\
\textbf{fun}\ iVP & : & VP \to Ent \to Prop \\
\textbf{fun}\ iTV & : & TV \to Ent \to Ent \to Prop \\
\textbf{fun}\ iCN & : & CN \to Ent \to Prop \\
\textbf{fun}\ iPN & : & PN \to Ent
\end{array}$$

Each interpretation function has a defining equation (a **def** judgement) for each syntactic form:

$$\begin{array}{lll}
\textbf{def}\ iS\ (Pred\ Q\ F\ ) & = & iNP\ Q\ (\lambda x \to iVP\ F\ x) \\
\textbf{def}\ iVP\ (Compl\ F\ Q\ )\ x & = & iNP\ Q\ (\lambda y \to iTV\ F\ x\ y) \\
\textbf{def}\ iNP\ (Every\ A\ )\ F & = & All\ (\lambda x \to If\ (iCN\ A\ x)\ (F\ x)) \\
\textbf{def}\ iNP\ (Indef\ A\ )\ F & = & Exist\ (\lambda x \to And\ (iCN\ A\ x)\ (F\ x)) \\
\textbf{def}\ iNP\ (Raise\ a\ )\ F & = & F\ (iPN\ a)
\end{array}$$

Thus the sentence

$$every\ woman\ loves\ John$$

has a syntax tree whose translation is expressed by the GF term

$$iS\ (Pred\ (Every\ Woman)\ (Compl\ Love\ (Raise\ John)))$$

By using the **def** equations, this term is computed into the term

$$All\ (\lambda x \to If\ (iCN\ Woman\ x)\ (iTV\ Love\ x\ (iPN\ John)))$$

We have left the interpretations of lexical rules undefined. In GF, this means simply that the computation stops at such applications of interpretation functions. To view the result of translation as a familiar-looking formula, we may give to the intrepretation functions linearization rules where expressions are followed by stars:

$$\begin{array}{lll}
\textbf{lin}\ iTV\ F\ x\ y & = & ss\ (F.s\ ++\ "*"\ ++\ par\ (x.s\ ++\ ","\ ++\ y.s)) \\
\textbf{lin}\ iCN\ F\ x & = & ss\ (F.s\ ++\ "*"\ ++\ par\ (x.s)) \\
\textbf{lin}\ iPN\ a & = & ss\ (a.s\ ++\ "*")
\end{array}$$

Thus our example is linearized into the formula

$$(\forall x)(woman^*(x) \supset loves^*(x, John^*)).$$

2.6 Semantics of predicate calculus

We have now formalized three of the four basic operations of Montague grammar: construction, linearization, and translation into logic. It remains to define the *model-theoretic interpretation* of logic. This can be done in type theory in a classical way by introducing a domain of truth values and a comain of individuals. The usual Boolean operations are defined for truth values:

cat *Bool*
cat *Ind*

fun *True*, *False* : *Bool*
fun *conj* : *Bool* → *Bool* → *Bool*
fun *neg* : *Bool* → *Bool*

def *conj True True* = *True*
def *conj* _ _ = *False*
def *neg True* = *False*
def *neg False* = *True*

We go on by defining valuation functions for formulas and terms:

fun *vProp* : *Prop* → *Bool*
fun *vEnt* : *Ent* → *Ind*

def *vProp* (*And A B*) = *conj* (*vProp A*) (*vProp B*)
def *vProp* (*Or A B*) = *neg* (*conj* (*neg* (*vProp A*)) (*neg* (*vProp B*)))
def *vProp* (*If A B*) = *neg* (*conj* (*vProp A*) (*neg* (*vProp B*)))

However, to extend the valuation function to quantifiers is not quite straightforward. The first problem is to define the semantic values of the quantifiers, for instance, for the universal quantifier, the function

fun *univ* : (*Ind* → *Bool*) → *Bool*

This is only possible if the set *Ind* is finite; otherwise we cannot compute a value of type *Bool*. Secondly, even if we manage with this, the problem remains to define the valuation of the universal quantifier. We cannot simply put

def *vProp* (*All P*) = *univ* (λ*x* → *vProp* (*P x*))

since this definition contains a type error: the variable x is first bound to the type *Ind*, but then used in the type *Ent*.

The proper formalization of model theory in type theory needs some form of satisfiability, identifying the free variables in formulas and assignments of *Ind* values to those variables. This can of course be done, but it is usually not the thing one does when working in a logical framework. It is much more common to have a *direct semantics* of logic. In the classical case, this means that formulas are directly defined as truth values, which means that the categories *Prop* and *Bool* are identified, and so are *Ent* and *Ind*. In the constructive case, this means that formulas are directly defined as sets of proofs, following the Curry-Howard isomorphism. The semantic rules for propositions are a certain kind of inference rules, called *introduction rules*. This kind of semantic is thus *proof-theoretical* rather than model-theoretical. In the rest of this paper, we will mainly follow the constructive interpretation of logic, with direct, proof-theoretical semantics.

2.7 Compositionality

A function on syntax trees is *compositional* if its application to a complex is constructed from the values of its application to the immediate constituents. Symbolically, an operation * from T to S is compositional if, for each constructor C of

type T, it holds uniformly that

$$(C\, x_1\, \ldots\, x_n)^* \;=\; F\, x_1^*\, \ldots\, x_n^*$$

for some F operating in the semantic domain. Thus F may not operate on the arguments $x_1, \ldots, x_n$, but only on their values under *. In particular, it is not permitted to do case analysis on the arguments $x_1, \ldots, x_n$.

It was emphasized by Montague and his followers that the translation of syntax trees into logic should be compositional: this is a guarantee that the construction of syntax trees is a semantically meaningful way of constructing them. It is easy to check that the translation functions we have defined above are compositional: there is only one **def** clause for each **fun** function, and each clause calls the arguments of the syntactic construction under the interpretation function of its category.

A less well-known requirement is the compositionality of linearization. This is what is needed to guarantee that the construction of syntax trees is a *linguistically* meaningful way of constructing them. If linearization goes into strings, as in the simple examples above, the constructor function F in the definition of compositionality consists of string operations, such as constant strings and concatenation.

The compositionality of linearization is not discussed in the PTQ paper, and it is far from obvious; the so-called *quantifying-in* rules are indeed formulated in a way that violates it, since they force to change the already formed linearizations of some constituents, e.g. to replace the first occurrence of a pronoun by a quantifier phrase. Yet it should be emphasized that a grammar is not compositional as a grammar of natural language if linearization is not compositional as well.

In GF, it has been a leading design principle that linearization rules are forced to be compositional. One consequence of this principle is that the formalization of quantifying-in rules is not possible in the same way as in Montague's PTQ. We will return to alternative, compositional formulations in Section 5 below.

3 Dependent types

3.1 Logic in constructive type theory

A dependent type is a type that depends on arguments belonging to other types. As the first example, we declare the non-dependent type of sets and, for any set, the dependent type of its elements:

cat *Set*
cat *El Set*

Here *Set* is not a dependent type, but *El* is. Using these types, we will now give a definition of the type theory of [13], also known as *lower-level type theory*, which can be used for interpreting (and, actually, extending) first-order predicate logic. We start with the operator Σ, which corresponds to existential quantification.

The Σ *formation* rule of type theory says that a Σ set is built from a set A and a family of sets over A:[6]

[6]The function name in GF is *Sigma*, since Greek letters are not permitted in identifiers.

fun *Sigma* : $(A : Set) \to (El\ A \to Set) \to Set$

Read as a quantified proposition, $\Sigma\, A\, B$ has the domain A and the propositional function B over A.

The Σ *introduction* rule tells that an element of a Σ set is a pair of an element of the domain and a proof of the propositional function as applied to that element:

fun *pair* : $(A : Set) \to (B : El\ A \to Set) \to (a : El\ A) \to El\ (B\ a) \to El\ (Sigma\ A\ B)$

The Σ *elimination* rules introduce *projection* functions that take apart the element and the proof from a pair:

fun *p* : $(A : Set) \to (B : El\ A \to Set) \to El\ (Sigma\ A\ B) \to El\ A$
fun *q* : $(A : Set) \to (B : El\ A \to Set) \to (c : El\ (Sigma\ A\ B)) \to El\ (B\ (p\ A\ B\ c))$

The elimination rules are justified by Σ *equality* rules, which tell how projections are computed for canonical elements, i.e. pairs:

def *p _ _ (pair _ _ a _)* = *a*
def *q _ _ (pair _ _ _ b)* = *b*

The definition of logical operators and inference rules using **fun** judgements produces full functional terms, which corresponds to what in [15] is called *monomorphic type theory*. What makes it monomorphic rather than polymorphic is that the constants *p*, *q*, etc., show all their type arguments. The polymorphic notation (as used in [13]) is in GF obtained by linearization rules that suppress the type arguments:

oper *f1* : $Str \to \{s : Str\} \to \{s : Str\} = \lambda f, x \to ss\ (f\ ++\ par\ (x.s))$
oper *f2* : $Str \to (x, y : \{s : Str\}) \to \{s : Str\} = \lambda f, x, y \to ss\ (f\ ++\ par\ (x.s\ ++\ ",\ ++\ y.s))$

lin *Sigma A B* = *ss* (*par* ("*Sigma*" ++ *B.v* ++ " : " ++ *A.s*) ++ *B.s*)
lin *pair _ _ a b* = *ss* (*par* (*a.s* ++ "," ++ *b.s*))
lin *p _ _ c* = *f1* "*p*" *c*
lin *q _ _ c* = *f1* "*q*" *c*

Another constant we need is Π.

fun *Pi* : $(A : Set) \to (El\ A \to Set) \to Set$
fun *lambda* : $(A : Set) \to (B : El\ A \to Set) \to ((x : El\ A) \to El\ (B\ x)) \to El\ (Pi\ A\ B)$
fun *app* : $(A : Set) \to (B : El\ A \to Set) \to El\ (Pi\ A\ B) \to (a : El\ A) \to El\ (B\ a)$

def *app _ _ (lambda _ _ b) a* = *b a*

lin *Pi A B* = *ss* (*par* ("*Pi*" ++ *B.v* ++ " : " ++ *A.s*) ++ *B.s*)
lin *lambda _ _ b* = *ss* (*par* ("*lambda*" ++ *b.v*) ++ *b.s*)
lin *app _ _ c a* = *f2* "*app*" *c a*

In the next section, we will show how Σ and Π are used in the semantics of natural-language constructions.

3.2 A syntax and its interpretation

The type-theoretical quantifiers Π and Σ have an argument A for a domain of individuals, and the second argument B must be a propositional function over A. In the formalization of mathematics, this has the effect that e.g.

$$(\Pi x : R)(\Sigma y : \mathit{Circle})(\mathit{radius}(y) = x)$$

is a well-formed proposition, since radius is a real number assigned to circles, whereas

$$(\Pi x : R)(\Sigma y : \mathit{Circle})(\mathit{radius}(x) = y)$$

is ill-formed by the same typing of the radius function. In linguistics, this gives us the possibility of expressing *selectional restrictions* in the grammar. One way to do this with dependent types is to relativize the categories of verbs and noun phrases to sets:

cat S
cat CN
cat $NP\ \ Set$
cat $PN\ \ Set$
cat $VP\ \ Set$
cat $TV\ \ Set\ Set$

The interpretation functions of dependent categories are dependently typed:

fun $iCN\ \ :\ \ CN \to Set$
fun $iS\ \ :\ \ S \to Set$
fun $iPN\ \ :\ \ (A : Set) \to PN\ A \to El\ A$
fun $iNP\ \ :\ \ (A : Set) \to NP\ A \to (El\ A \to Set) \to Set$
fun $iVP\ \ :\ \ (A : Set) \to VP\ A \to El\ A \to Set$
fun $iTV\ \ :\ \ (A, B : Set) \to TV\ A\ B \to El\ A \to El\ B \to Set$

The relativization is easy to extend to the formation of syntax trees...

fun $Raise\ \ :\ \ (A : Set) \to PN\ A \to NP\ A$
fun $Every\ \ :\ \ (A : CN) \to NP\ (iCN\ A)$
fun $Indef\ \ :\ \ (A : CN) \to NP\ (iCN\ A)$
fun $Pred\ \ :\ \ (A : Set) \to NP\ A \to VP\ A \to S$
fun $Compl\ \ :\ \ (A, B : Set) \to TV\ A\ B \to NP\ B \to VP\ A$

...and to their interpretations:

def $iNP\ \_\ (Raise\ A\ a\ )\ B\ \ =\ \ B\ (iPN\ A\ a)$
def $iNP\ \_\ (Every\ A\ )\ B\ \ =\ \ Pi\ (iCN\ A)\ B$
def $iNP\ \_\ (Indef\ A\ )\ B\ \ =\ \ Sigma\ (iCN\ A)\ B$
def $iS\ (Pred\ A\ Q\ F\ )\ \ =\ \ iNP\ A\ Q\ (\lambda x \to iVP\ A\ F\ x)$
def $iVP\ \_\ (Compl\ A\ B\ F\ Q\ )\ a\ \ =\ \ iNP\ B\ Q\ (\lambda y \to iTV\ A\ B\ F\ a\ y)$

The linearization rules are similar to the first example, just ignoring the extra domain arguments. However, to prepare the way for future extensions of the fragment, we introduce a number of *parameters*,

$$\begin{array}{l}
\textbf{param}\ Gen\ =\ He \mid She \mid It \\
\textbf{param}\ Case\ =\ Nom \mid Acc \\
\textbf{param}\ Num\ =\ Sg \mid Pl
\end{array}$$

and define the linearization types thus:

$$\begin{array}{l}
\textbf{lincat}\ PN\ =\ \{s\ :\ Case \Rightarrow Str\} \\
\textbf{lincat}\ NP\ =\ \left\{ \begin{array}{lcl} s & : & Case \Rightarrow Str \\ n & : & Num \end{array} \right\} \\
\textbf{lincat}\ CN\ =\ \left\{ \begin{array}{lcl} s & : & Num \Rightarrow Str \\ g & : & Gen \end{array} \right\} \\
\textbf{lincat}\ VP\ =\ \{s\ :\ Num \Rightarrow Str\} \\
\textbf{lincat}\ TV\ =\ \{s\ :\ Num \Rightarrow Str\}
\end{array}$$

The *Case* parameter in *PN* and *NP* will be used to produce nominative and accusative forms of pronouns. The inherent *Gen* feature of *CN* will be used in producing pronouns. We introduce some operations to deal with these parameters:

$$\begin{array}{l}
\textbf{oper}\ regPN\ =\ \lambda bob \rightarrow \{s\ =\ \textbf{table}\ \{_ \Rightarrow bob\}\} \\
\textbf{oper}\ regNP\ =\ \lambda bob \rightarrow \left\{ \begin{array}{lcl} s & = & \textbf{table}\ \{_ \Rightarrow bob\} \\ n & = & Sg \end{array} \right\} \\
\textbf{oper}\ regCN\ =\ \lambda bike \rightarrow \left\{ \begin{array}{lcl} s & = & \textbf{table}\ \left\{ \begin{array}{lcl} Sg & \Rightarrow & bike \\ Pl & \Rightarrow & bike\ +\ \text{“}s\text{”} \end{array} \right\} \\ g & = & It \end{array} \right\}
\end{array}$$

and formulate the linearization rules thus:[7]

$$\begin{array}{l}
\textbf{lin}\ Raise\ _\ a\ \ =\ \ \left\{ \begin{array}{lcl} s & = & a.s \\ n & = & Sg \end{array} \right\} \\
\textbf{lin}\ Every\ A\ \ =\ \ regNP\ (\text{“}every\text{”}\ ++\ A.s\ !\ Sg) \\
\textbf{lin}\ Indef\ A\ \ =\ \ regNP\ (\text{“}a\text{”}\ ++\ A.s\ !\ Sg) \\
\textbf{lin}\ Pred\ _\ Q\ F\ \ =\ \ \{s\ =\ Q.s\ !\ Nom\ ++\ F.s\ !\ Q.n\} \\
\textbf{lin}\ Compl\ _\ _\ F\ Q\ \ =\ \ \{s\ =\ \textbf{table}\ \{n \Rightarrow F.s\ !\ n\ ++\ Q.s\ !\ Acc\}\}
\end{array}$$

4 Anaphora

In the same way as in [19] (chapter 3), we define *If* and *And* as variable-binding connectives, interpreted as Π and Σ, respectively:

$$\begin{array}{l}
\textbf{fun}\ If\ \ :\ \ (A : S) \rightarrow (El\ (iS\ A) \rightarrow S) \rightarrow S \\
\textbf{fun}\ And\ \ :\ \ (A : S) \rightarrow (El\ (iS\ A) \rightarrow S) \rightarrow S \\
\\
\textbf{def}\ iS\ (If\ A\ B\ )\ \ =\ \ Pi\ (iS\ A)\ (\lambda x \rightarrow iS\ (B\ x)) \\
\textbf{def}\ iS\ (And\ A\ B\ )\ \ =\ \ Sigma\ (iS\ A)\ (\lambda x \rightarrow iS\ (B\ x)) \\
\\
\textbf{lin}\ If\ A\ B\ \ =\ \ \{s\ =\ \text{“}if\text{”}\ ++\ A.s\ ++\ B.s\} \\
\textbf{lin}\ And\ A\ B\ \ =\ \ \{s\ =\ A.s\ ++\ \text{“}and\text{”}\ ++\ B.s\}
\end{array}$$

Thus we interpret e.g.

[7] The keyword **table** designates a table of alternative forms, such as the singular and the plural of a noun. The exclamation mark ! expresses the selection of a form from a table.

as

$$(\Pi z : (\Sigma x : man)(\Sigma x : donkey)own(x, y))beat(p(z), p(q(z))).$$

What remains to be seen is how the pronouns *he* and *it* are created from the variables.

4.1 A system of anaphoric expressions

Like in [19] (chapter 4), we represent pronouns and definite noun phrases as functions whose interpretations are identity mappings:

$$\begin{array}{l}\textbf{fun}\ Pron\ :\ (A : CN) \to El\,(iCN\ A) \to PN\,(iCN\ A)\\ \textbf{fun}\ Def\ :\ (A : CN) \to El\,(iCN\ A) \to PN\,(iCN\ A)\\ \\ \textbf{def}\ iPN\ \_\,(Pron\ \_\ a\,)\ =\ a\\ \textbf{def}\ iPN\ \_\,(Def\ \_\ a\,)\ =\ a\end{array}$$

Even though they have the same interpretation, each of the terms a, $Pron(A, a)$, and $Def\,(A, a)$ is linearized in a different way:

$$\begin{array}{l}\textbf{lin}\ Pron\ A\ \_\ =\ \left\{\begin{array}{lcl} s & = & pron\,!\,A.g\\ n & = & Sg\end{array}\right\}\\ \textbf{lin}\ Def\ A\ \_\ =\ regPN\,(\text{“}the\text{”}\ ++\ A.s\,!\,Sg)\\ \\ \textbf{oper}\ pron : Gen \Rightarrow Case \Rightarrow Str\ =\ \textbf{table}\left\{\begin{array}{lcl} He & \Rightarrow & \textbf{table}\left\{\begin{array}{lcl} Nom & \Rightarrow & \text{“}he\text{”}\\ Acc & \Rightarrow & \text{“}him\text{”}\end{array}\right\}\\ She & \Rightarrow & \textbf{table}\left\{\begin{array}{lcl} Nom & \Rightarrow & \text{“}she\text{”}\\ Acc & \Rightarrow & \text{“}her\text{”}\end{array}\right\}\\ It & \Rightarrow & \textbf{table}\,\{\_ \Rightarrow \text{“}it\text{”}\}\end{array}\right\}\end{array}$$

Thus the syntax tree representing the donkey sentence is

```
If (Pred (iCN Man) (Indef Man) (Compl (iCN Man) (iCN Donkey) Own (Indef Donkey)))
   (λx → Pred (iCN Man) (Raise (iCN Man) (Pron Man (p (iCN Man)
   (λx' → Sigma (iCN Donkey) (λy → iTV (iCN Man) (iCN Donkey) Own x' y)) x)))
   (Compl (iCN Man) (iCN Donkey) Beat (Raise (iCN Donkey) (Pron Donkey (p (iCN Donkey)
   (λx' → iTV (iCN Man) (iCN Donkey) Own (p (iCN Man) (λy → Sigma (iCN Donkey)
   (λy' → iTV (iCN Man) (iCN Donkey) Own y y')) x) x') (q (iCN Man)
   (λy → Sigma (iCN Donkey) (λy' → iTV (iCN Man) (iCN Donkey) Own y y')) x))))))
```

This is the full tree. By suppressing most of the type information, we get a more readable term,

```
If (Pred (Indef Man) (Compl Own (Indef Donkey)))
   (λx → Pred (Raise (Pron Man (p x))) (Compl Beat (Raise (Pron Donkey (p (q x))))))
```

As for modified anaphoric expressions, [19] has a general rule *Mod* subsuming both of

the man that owns a donkey,

the donkey that a man owns.

The general rule is an instance of quantifying-in structures, to which we return in Section 5. Now we just give two less general rules, corresponding to relative clauses binding the subject and the object, respectively:

fun *ModVP* : $(A : CN) \to (F : VP\ (iCN\ A)) \to (a : El\ (iCN\ A)) \to El\ (iVP\ (iCN\ A)\ F\ a) \to PN\ (iCN\ A)$
fun *ModTV2* : $(A : Set) \to (B : CN) \to (Q : NP\ A) \to (F : TV\ A\ (iCN\ B)) \to (b : El\ (iCN\ B)) \to El\ (iNP\ A\ Q\ (\lambda x \to iTV\ A\ (iCN\ B)\ F\ x\ b)) \to PN\ (iCN\ B)$

def $iPN\ \_\ (ModVP\ \_\ \_\ a\ \_\ ) = a$
def $iPN\ \_\ (ModTV2\ \_\ \_\ \_\ \_\ b\ \_\ ) = b$

lin $ModVP\ A\ F\ \_\ \_ = regPN\ (\text{“}the\text{”} ++ A.s\,!\,Sg ++ \text{“}that\text{”} ++ F.s\,!\,Sg)$
lin $ModTV2\ \_\ B\ Q\ F\ \_\ \_ = regPN\ (\text{“}the\text{”} ++ B.s\,!\,Sg ++ \text{“}that\text{”} + + Q.s\,!\,Nom ++ F.s\,!\,Sg)$

The reader can check that the pronoun *he* in the donkey sentence can be paraphrased by

the man,

the man that owns a donkey,

the man that owns the donkey,

the man that owns the donkey that he owns,

since all of the arguments needed for these expressions can be constructed from the bound variable, and all the results are definitionally equal.

5 Quantifying in

The treatment of quantification by the functions *Pred* and *Compl*

fun *Pred* : $NP \to VP \to S$
fun *Compl* : $TV \to NP \to VP$

(Section 2.3) does not employ variable binding in syntax trees. It also permits simple linearization rule: just attach a noun phrase before or after a verb:

lin $Pred\ Q\ F = cc2\ Q\ F$
lin $Compl\ F\ Q = cc2\ F\ Q$

This mechanism is, however, less powerful than variable binding in logic. The function *Pred* corresponds to quantification binding a variable only having one single occurrence in one single place (the first argument of a verb), and the function *Compl* binds a variable in another place. For instance, the functions permit us to parse

every man loves a woman

as the tree

$$\mathit{Pred}\ (\mathit{Every\ Man})\ (\mathit{Compl\ Love}\ (\mathit{Indef\ Woman}))$$

which expresses the proposition

$$(\forall x)(\mathit{Man}(x) \supset (\exists y)(\mathit{Woman}(y)\&\mathit{Love}(x,y))).$$

But we cannot interpret the same sentence as

$$(\exists y)(\mathit{Woman}(y)\&(\forall x)(\mathit{Man}(x) \supset \mathit{Love}(x,y))).$$

In other words, we do not find the scope ambiguity of the quantifiers in the English sentence. (If the latter does not feel like a possible interpretation of the sentence, consider the variant *every reporter ran like mad after a previously unknown Estonian pop singer*.)

One of the things that Montague wanted to achieve with his grammars [14] was to reveal scope ambiguities. In addition to rules corresponding to *Pred* and *Compl* (see Section 6) below, he had what has come to be known as quantifying-in rules. In words, these rules can be formulated as follows:

> $(\forall x : A)B$ is linearized by substituting the phrase *every* A for the variable x in B.
>
> $(\exists x : A)B$ is linearized by substituting the phrase *a* A for the variable x in B.

In the light of these rules, the sentence *every man loves a woman* can result from both of the formulas cited above, and is hence ambiguous.

Another advantage of quantifying-in rules is that they readily explain sentences like

the mother of every man loves the father of some woman

where the bound variable occurs deeper than as an immediate argument of the verb. The occurrence can, obviously, be arbitrarily deep:

the mother of the father of the wife of every man.

A computational disadvantage of quantifying-in rules is that they are difficult to reverse into parsing rules. A solution of the problem was presented by Friedman and Warren in [7]; they noticed that the rules formulated by Montague have the consequence that every sentence, even a simple one such as

John walks

has infinitely many interpretations:

$\mathit{Walk}(\mathit{John})$,

$(\forall x : \mathit{Man})\,\mathit{Walk}(\mathit{John})$,

$(\forall x : \mathit{Man})(\forall y : \mathit{Man})\,\mathit{Walk}(\mathit{John})$,

etc! This problem is called *vacuous binding*: there is no occurrence of the variable in the formula, so that substitution does not leave any trace of the quantifier phrase. Another, related problem is *multiple binding*: the proposition

$$(\forall x : Man)Love(x, x)$$

should not be linearized into

every man loves every man

but rather to

every man loves himself.

Montague's $\forall$ linearization rule did give a solution to the problem of multiple binding:

> $(\forall x : A)B$ is linearized by substituting the phrase *every* A for the first occurrence of the variable x in B, and the pronoun corresponding to the gender of A for the other occurrences.

But this is a brittle solution, which does not e.g. get reflexive pronouns right. Thus quantifier rules in Montague grammar were one of the most prominent topics in formal linguistics in the 1970's (see e.g. [4]).

From the GF point of view, quantifying-in rules are problematic since they are not compositional: a compositional linearization should construct the linearization of a complex from the linearizations of its parts, but, since linearizations are strings (or records of string-related objects), there is no such operation as substituting a noun phrase for a certain occurrence of a variable. In a string, there are no distinguishable variables left, but just linearizations of variables, which may be observable to the human eye as substrings, but which cannot be separated from the whole string by a reliable formal procedure.

Thus quantifying-in rules cannot be formulated directly in GF. But the rules can be approximated, in ways that also avoid the drawbacks of the quantifying-in rules, due to their being too strong. We will present two solutions: one with combinators and the other with discontinuous constituents.

5.1 Solution with combinators

The combinator solution is inspired by Steedman's combinatory categorial grammar [26]. The idea is the same as in combinatory logic [5]: to eliminate variable bindings by using higher-order functions. The *Pred* and *Compl* functions are, already, examples of combinators. What we need is some more combinators and, in fact, syntactic categories.

We introduce a category *VS* ("slash verbs"), interpreted, like *VP*, as functions from entities to propositions:

cat *VS*

fun *iVS* : $VS \to Ent \to Prop$

Expressions of *VS* are formed and used by combinators that are dual to *Pred* and *Compl*: the *fractioning* function *Fract* forms a *VS* by providing a subject *NP* to a *TV*, and the complementizer *ComplFract* forms a sentence from a *VS* by providing an object *NP*:

fun *Fract* : $NP \to TV \to VS$
fun *ComplFract* : $VS \to NP \to S$

def $iVS\ (Fract\ Q\ F)\ y = iNP\ Q\ (\lambda x \to iTV\ F\ x\ y)$
def $iS\ (ComplFract\ F\ Q) = iNP\ Q\ (\lambda y \to iVS\ F\ y)$

lin $Fract\ Q\ F = cc2\ Q\ F$
lin $ComplFract\ F\ Q = cc2\ F\ Q$

Now the sentence

every man loves a woman

has the two parses

Pred (*Every Man*) (*Compl Love* (*Indef Woman*))

ComplFract (*Fract* (*Every Man*) *Love*) (*Indef Woman*)

which are interpreted with the two different scopings of the quantifiers.

The category *VS* is similar to the *slash category* S/NP ("sentence missing noun phrase") of GPSG [8]. However, GF does not have a general rule for forming such categories; an unrestricted general rule would, like the quantifying-in rules, be too strong and difficult to manage. Thus we have to introduce each slash category separately. Another example is the category of one-place functions from entities to entities,

cat *F1*
fun *iF1* : $F1 \to Ent \to Ent$
fun *AppF1* : $F1 \to NP \to NP$
def $iNP\ (AppF1\ f\ Q)\ F = iNP\ Q\ (\lambda x \to F\ (iF1\ f\ x))$
lin $AppF1\ f\ Q = cc2\ f\ Q$

an example being

fun *Father* : *F1*
lin *Father* = *ss* ("*the*" ++ "*father*" ++ "*of*")

In GPSG terms, *F1* is the category NP/NP. Fortunately, the number of slash categories that are really needed is limited.

Slash categories have other applications besides modelling quantifying-in rules. One is coordination, as e.g. in

Peter loves and John adores every woman.

Another one is relative clauses. We can interpret relative clauses as propositional functions, and turn any *VP* and *VS* into a relative clause:

cat *RC*

fun *iRC* : $RC \to Ent \to Prop$

fun *RelVP* : $VP \to RC$

fun *RelVS* : $VS \to RC$

def $iRC\,(RelVP\,F)\,x = iVP\,F\,x$

def $iRC\,(RelVS\,F)\,x = iVS\,F\,x$

lin $RelVP\,F = ss\,(\text{“}that\text{”} ++ F.s)$

lin $RelVS\,F = ss\,(\text{“}that\text{”} ++ F.s)$

Relative clauses can be attached to common nouns to modify them:

fun *RelCN* : $CN \to RC \to CN$

def $iCN\,(RelCN\,A\,R)\,x = And\,(iCN\,A\,x)\,(iRC\,R\,x)$

lin $RelCN = cc2$

Thus we can form

man that walks,

woman that every man loves.

GPSG also has “slash propagation” rules, such as

$$S/NP \longrightarrow S/NP\ NP/NP.$$

In type theory, this rule can be formalized as an instance of function composition:

fun *CompVS* : $VS \to F1 \to VS$

def $iVS\,(CompVS\,F\,f)\,x = iVS\,F\,(iF1\,f\,x)$

lin $CompVS = cc2$

Thus we can form

man that every woman loves the father of the father of the father of.

5.2 Solution with discontinuous constituents

Combinators can be seen as an indirect solution to the quantifying-in problem, because they require changes in the abstract syntax: the addition of new categories and combinators. However, GF also provides a more direct solution, which can do with fewer rules. The solution uses *discontinuous constituents* to mark the “slot” in which there is a variable binding. Thus we define the category *P1* of one-place predicates as one with a linearization consisting of two strings: the substrings before and after the place of binding:

cat *P1*

lincat $P1 = \left\{ \begin{array}{lcl} s1 & : & Str \\ s2 & : & Str \end{array} \right\}$

The rule *PredP1* performs an insertion of a noun phrase:

$$\textbf{fun}\ \mathit{PredP1}\ :\ \mathit{NP} \to \mathit{P1} \to \mathit{S}$$
$$\textbf{lin}\ \mathit{PredP1}\ Q\ F\ =\ \mathit{ss}\,(F.s1\ ++\ Q.s\ ++\ F.s2)$$

A *P1* can be constructed from a transitive verb in two ways:

$$\textbf{fun}\ \mathit{P1Compl}\ :\ \mathit{TV} \to \mathit{NP} \to \mathit{P1}$$
$$\textbf{fun}\ \mathit{P1Subj}\ :\ \mathit{NP} \to \mathit{TV} \to \mathit{P1}$$
$$\textbf{lin}\ \mathit{P1Compl}\ F\ Q\ =\ \left\{ \begin{array}{lcl} s1 & = & [\,] \\ s2 & = & F.s\ ++\ Q.s \end{array} \right\}$$
$$\textbf{lin}\ \mathit{P1Subj}\ Q\ F\ =\ \left\{ \begin{array}{lcl} s1 & = & Q.s\ ++\ F.s \\ s2 & = & [\,] \end{array} \right\}$$

This treatment can be extended to predicates with any number n of argument places, by using $n + 1$ discontinuous constituents.

6 The PTQ fragment

6.1 PTQ rule-to-rule

The rule-to-rule fashion of formalizing the PTQ grammar ([14], chapter 8) groups the rules in triples. For each category, we have

a **cat** judgement introducing the category,

a **fun** judgement declaring its interpretation function (and thereby defining its domain of possible denotations),

a **lincat** judgement defining its linearization type.

Thus we have e.g.[8]

$$\textbf{cat}\ \mathit{S}$$
$$\textbf{fun}\ \mathit{iS}\ :\ \mathit{S} \to \mathit{Prop}$$
$$\textbf{lincat}\ \mathit{S}\ =\ \{s\ :\ \mathit{Str}\}$$

$$\textbf{cat}\ \mathit{CN}$$
$$\textbf{fun}\ \mathit{iCN}\ :\ \mathit{CN} \to \mathit{Ent} \to \mathit{Prop}$$
$$\textbf{lincat}\ \mathit{S}\ =\ \{s\ :\ \mathit{Str}\}$$

$$\textbf{cat}\ \mathit{NP}$$
$$\textbf{fun}\ \mathit{iNP}\ :\ \mathit{NP} \to (\mathit{Ent} \to \mathit{Prop}) \to \mathit{Prop}$$
$$\textbf{lincat}\ \mathit{NP}\ =\ \{s\ :\ \mathit{Case} \Rightarrow \mathit{Str}\}$$

[8] To simplify the presentation, we ignore possible worlds and intensional operations. They would be no problem in GF, since we could introduce a new basic type of possible words.

cat *IV*

fun *iIV* : $IV \to Ent \to Prop$

lincat *IV* = $\{s : Mod \Rightarrow Str\}$

cat *IAV*

fun *iIAV* : $IAV \to (Ent \to Prop) \to Ent \to Prop$

lincat *IAV* = $\{s : Str\}$

For syntactic rules, we have

a **fun** judgement introducing the function,

a **def** defining the interpretation function for the value category of the function for the expressions formed by this function.

a **lin** judgement defining the linearization rule of the function.

Examples of such rules are the following, where the function names refer to Montague's rule names:[9]

fun *S2F0* : $CN \to NP$

def $iNP\ (S2F0\ A)\ F = All\ (\lambda x \to If\ (iCN\ A\ x)\ (F\ x))$

lin $S2F0\ A = mkNP\ (\text{“}every\text{”}\ ++\ A.s)$

fun *S2F1* : $CN \to NP$

def $iNP\ (S2F1\ A)\ F = Exist\ (\lambda x \to And\ (iCN\ A\ x)\ (F\ x))$

lin $S2F1\ A = mkNP\ (indef\ ++\ A.s)$

fun *S3F3* : $CN \to (Ent \to S) \to CN$

def $iCN\ (S3F3\ A\ F)\ x = And\ (iCN\ A\ x)\ (iS\ (F\ x))$

lin $S3F3\ A\ B = ss\ (A.s\ ++\ \text{“}such\text{”}\ ++\ \text{“}that\text{”}\ ++\ B.s)$

fun *S4F4* : $NP \to IV \to S$

def $iS\ (S4F4\ Q\ F) = iNP\ Q\ (iIV\ F)$

lin $S4F4\ Q\ F = ss\ (Q.s\ !\ Nom\ ++\ F.s\ !\ Ind)$

We have used two new auxiliary operations:

oper $indef : Str = \mathbf{pre}\ \{\text{“}a\text{”}; \text{“}an\text{”}\ /\ \mathbf{strs}\ \{\text{“}a\text{”}; \text{“}e\text{”}; \text{“}i\text{”}; \text{“}o\text{”}\}\}$
oper $mkNP : Str \to \{s : Case \Rightarrow Str\} = \lambda s \to \{s = \mathbf{table}\ \{\_ \Rightarrow s\}\}$

Using the random syntax tree generator of GF followed by linearization, we generated some random example sentences:[10]

[9] They are formed by combining the rule name, of form S_n, with the syntactic constructor name, of form F_k. This combination is necessary, since neither the rule nor the constructor names alone unique determine the constructions.

[10] The complete grammar is available from the GF web page [21].

necessarily every price has not risen or mary has not run and a temperature wishes to seek the woman

every woman changes

the price has changed

the price or every man will try to change

necessarily every price has risen

the park has not run voluntarily

a unicorn walks about a pen

a man will not change

7 Grammars for different languages

A *multilingual grammar* in GF is a grammar with one abstract syntax and many concrete syntaxes. *Translation* in a multilingual grammar is parsing with one concrete syntax followed by linearization with another concrete syntax.

We illustrate multilingual grammars first by giving a French concrete syntax to the PTQ fragment. We continue by showing how some subtleties of translation can be managed by semantic disambiguation using the type system.[11]

7.1 The PTQ fragment in French

To define a new concrete syntax, the starting point is a new set of **param** and **lincat** definitions. The differences between these definitions in different languages can be enormous, and they explain much of the abstraction power that the abstract syntax has over concrete syntaxes. The PTQ fragment is small, but even there some of these differences are instantly shown. For instance, French has a gender parameter belonging to common nouns as inherent feature:

$$\textbf{param}\ \mathit{Gen} \ = \ \mathit{Masc} \mid \mathit{Fem}$$

$$\textbf{lincat}\ \mathit{CN} \ = \ \left\{ \begin{array}{lcl} s & : & \mathit{Str} \\ g & : & \mathit{Gen} \end{array} \right\}$$

Noun phrases also have a gender, but they can moreover be inflected in case, which has effect on pronouns. A further subtlety is introduced by the *clitic* (unstressed) pronouns, such as the accusative case *la* ("her") of *elle* ("she"). When used as an object, a clitic pronoun is placed before the verb and not after it, as other kinds of noun phrases. We solve this puzzle by introducing a parameter type *NPType* and making it an inherent feature of noun phrases:

$$\textbf{param}\ \mathit{NPType} \ = \ \mathit{Pron} \mid \mathit{NoPron}$$

$$\textbf{lincat}\ \mathit{NP} \ = \ \left\{ \begin{array}{lcl} s & : & \mathit{Case} \Rightarrow \mathit{Str} \\ g & : & \mathit{Gen} \\ p & : & \mathit{NPType} \end{array} \right\}$$

[11] The fact that semantic information is needed in high-quality translation is one of the most important motivations of computational semantics; cf. [23] for a project using Montague grammar for machine translation.

The syntax rule in which this distinction matters is the complementization rule of transitive verbs.

$$\textbf{fun } \mathit{S5F5} \ : \ \mathit{TV} \to \mathit{NP} \to \mathit{IV}$$

$$\textbf{lin } \mathit{S5F5}\ F\ Q \ = \ \textbf{case } \ Q.p \ \textbf{of} \left\{ \begin{array}{lll} \mathit{Pron} & \Rightarrow & \left\{ \begin{array}{lll} s & = & \textbf{table}\ \{m \Rightarrow Q.s\,!\,\mathit{CAcc} \ ++ \ F.s\,!\,m\} \\ s2 & = & [\,] \\ \mathit{aux} & = & F.\mathit{aux} \end{array} \right\} \\ \mathit{NoPron} & \Rightarrow & \left\{ \begin{array}{lll} s & = & F.s \\ s2 & = & Q.s\,!\,\mathit{Acc} \\ \mathit{aux} & = & F.\mathit{aux} \end{array} \right\} \end{array} \right\}$$

The complementization rule shows some other subtleties as well: first, we distinguish between the clitic and stressed accusative (the stressed form of *la* is *elle*). The stressed form is needed with prepositions, but also for coordinated pronouns: *john seeks him or her* is translated to *john cherche lui ou elle*. The second subtlety is the need to indicate the auxiliary verb required by verbs: *john has talked* translates to *john a parlé*, whereas *john has arrived* translates to *john est arrivé*. The third subtlety is that verb phrases have two discontinuous constituents—the second one is for the complement. In the PTQ fragment, this feature is needed in the formation of the negation: the particle *pas* appears between the two discontinuous parts. Thus

$$\textbf{lincat } \mathit{IV} \ = \ \left\{ \begin{array}{lll} s & : & \mathit{VForm} \Rightarrow \mathit{Str} \\ s2 & : & \mathit{Str} \\ \mathit{aux} & : & \mathit{AuxType} \end{array} \right\}$$

$$\textbf{lin } \mathit{S17F11}\ Q\ F \ = \ ss\,(Q.s\,!\,\mathit{Nom} \ ++ \ \mathit{ne} \ ++ \ F.s\,!\,\mathit{Ind} \ ++ \ \text{“}\mathit{pas}\text{”} \ ++ \ F.s2)$$

One might argue that the category *IV* of verb phrases is an artifact of PTQ, or at most something that works in English but not in French. However, using discontinuous constituents in the concrete syntax makes it completely natural to have this category in the abstract syntax: all of the linearization rules given above are compositional.

The only structures in PTQ that we found impossible to linearize compositionally are the conjunction and disjunction of verb phrases:

$$\textbf{fun } \mathit{S12F8}, \mathit{S12F9} \ : \ \mathit{IV} \to \mathit{IV} \to \mathit{IV}$$

There are two problems. The first one is to form the negation in the case where one of the constituent verb phrases has a complement. The second one is to choose the auxiliary verb if the constituents have different ones. For these structures, paraphrases using sentence coordination would really be needed. For instance, one has to paraphrase

john does not run and seek a unicorn

as something like

jean ne court pas ou il ne cherche pas une licorne.

Alternatively, one may try to find a parameter indicating if a verb phrase is complex: case distinctions on syntax trees can always be replaced by distinctions on a parameter, if the number of cases is limited.

Here are the GF-produced French translations of the English sentences presented in the previous section.[12]

nécessairement chaque prix n'a pas monté ou mary n'a pas couru et une température espère de chercher la femme

chaque femme change

le prix a changé

le prix ou chaque homme va essayer de changer

nécessairement chaque prix a monté

le parc n'a pas couru volontairement

une licorne marche sur un stylo

un homme ne va pas changer

7.2 Translating anaphoric expressions

The French sentence

si un homme possède un âne il le bat

is ambiguous, since the masculine pronon (*il, le*) matches both the man (*un homme*) and the donkey (*un âne*). Leaving out the two interpretations where both pronouns refer to the same object, we are left with two English translations:

if a man owns a donkey he beats it,

if a man owns a donkey it beats him.

Now, it is possible that the type of the verb *battre* rules out e.g. the latter reading:

fun *Battre* : *TV Man Donkey*

Such is often the case in technical text, where selectional restrictions are dictated by semantic considerations. If not, we must conclude that the sentence is ambiguous.

On the other direction, suppose we want to take one of the above English sentences and produce an *unambiguous* French translation. We can then use the equality (**def**) rules of anaphoric expressions to generate unambiguous paraphrases:

si un homme possède un âne l'homme bat l'âne,

si un homme possède un âne l'âne bat l'homme.

[12]The complete grammar is available from the GF web page [21].

8 Grammar composition

8.1 Logical vs. linguistic syntax

Montague "fail[ed] to see any great interest in syntax except as a preliminary to semantics" ("Universal Grammar", chapter 7 in [14]). From this he drew the conclusion that syntax must be formulated with semantics in mind, which meant, in particular, that he considered syntax as practised by Chomsky and other linguists irrelevant. His own syntax, on the other hand, has been criticized on opposite grounds: as reflecting the structure of logic rather than the structure of language. What these two opposite requirements exactly mean can be characterized in terms of compositionality: a syntax reflects logical structure only if it has compositional semantics. It reflects linguistic structure only if it has compositional linearization. As pointed out above (Section 2.7), Montague only considered compositionality of semantics, not of linearization. This is one reason for the unnaturalness of his grammar for linguists.

With ingenious use of parameters and discontinuous constituents, it is often possible in GF to give compositional linearization rules to very unnatural constructions. Therefore, compositionality is just a necessary condition for reflecting linguistic structure. It is hard to pin down exactly where the lack of linguistic elegance then lies; from the programming point of view, the grammar appears as a hack, due to its complexity and lack of modularity. This is to some extent already true of a grammar as simple as the GF formulation of the PTQ fragment: a grammar covering (probably as a proper part) the same fragment of English or French but written from the linguistic perspective would certainly look different[13]. Why should it then be irrelevant for semantics? Wouldn't it be better to use the linguistically motivated grammar as a component of the complete system, where semantics is another component?

8.2 Application grammars and resource grammars

The original idea of GF was to make it possible to map abstract, logical structures to concrete, linguistic objects. This is what linearization rules do. An intended application was to study semantically well-defined domains, such as object-oriented software specifications [10] or business letters [11], and map each domain to different languages which have a common tradition of speaking about the domain. One advantage of this approach is that the linguistic aspects of such translations are simple, since only a limited number of linguistic structures are used in the domain. It is also clear that semantics, if it is to follow the rigorous standards of type theory, cannot be given to unlimited natural language, but only to a well-defined domain.

One disadvantage of domain-specific grammars is the lack of linguistic elegance due to grammars being written in an *ad hoc* way. Another disadvantage is their poor reusability: the fragment of e.g. French needed for software specifications is different from the fragment needed for business letters, and, even though there is some overlap, each concrete syntax essentially has to be written from scratch.

[13]For instance, the very different meachanisms of negation in these two languages would probably be described in different ways.

The solution to these problems is to distinguish between *application grammars* and *resource grammars*. An application grammar is one reflecting a semantic model. A resource grammar is one written from a purely linguistic perspective, aiming at complete coverage of linguistic facts such as morphology, agreement, word order, etc. There is no attempt to give semantics to the resource grammar: its semantics is only given indirectly through the uses to which it is put in application grammars.

The preferred way to write an application grammar is thus the following: an abstract syntax is defined from purely semantic considerations. The concrete syntax is defined by mapping the abstract syntax, not directly to strings and records, but to trees in the resource grammar. The final product is obtained by *grammar composition*, which is a compilation phase that eliminates the intermediate tree structures and gives a direct mapping from abstract syntax to strings and records.

In a multilingual grammar, the linearization may well assign to one and the same abstract structure quite different structures in different languages. To take a familiar example, the two-place predicate *x misses y* is in both English and French expressed by a two-place verb, but in French, the order of the arguments is the opposite (*x manque à y*). The resource-grammar based linearization rules look as follows:

$$\begin{aligned} \textbf{lin}\ \mathit{Miss}\ x\ y &= \mathit{PredVP}\ x\ (\mathit{Compl}\ (\mathit{verbS}\ \text{“}\mathit{miss}\text{”})\ y) \\ \textbf{lin}\ \mathit{Miss}\ x\ y &= \mathit{PredVP}\ y\ (\mathit{Compl}\ (\mathit{verbEr}\ \text{“}\mathit{manquer}\text{”})\ x) \end{aligned}$$

These rules show on a high level of abstraction what linguistic structures are used in expressing the predicate *Miss*. To the grammarian, they give the advantage that she does not have to care about the details of agreement and word order (including clitics). Thus there is a division of labour between logicians (or experts of other domains) writing application grammars and linguists writing resource grammars.

For the debate between logical and linguistic grammars, the implication is clear: there *is* a great interest in syntax that is *not* a preliminary to semantics. Such a syntax does not have a semantics by itself, but it is an excellent intermediate step between abstract semantic structures and concrete language.

References

[1] P. Blackburn and J. Bos. *Representation and Inference for Natural Language.* Studies in Logic, Language, and Information, CSLI Press, to appear.

[2] N. Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[3] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, 1984.

[4] R. Cooper. *Quantification and Syntactic Theory.* D. Reidel, 1981.

[5] H. B. Curry and R. Feys. *Combinatory Logic, Vol. 1.* North-Holland, Amsterdam, 1958.

[6] J. Friedman. Expressing logical formulas in natural language. In J. Groenendijk, T. Janssen, and M. Stokhof, editors, *Formal Methods in the Study of Language, Part 1*, pages 113–130. Mathematisch Centrum, Amsterdam, 1981.

[7] J. Friedman and D. Warren. A parsing method for Montague grammar. *Linguistics and Philosophy*, 2:347–372, 1978.

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

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

[10] R. Hähnle, K. Johannisson, and A. Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.

[11] J. Khegai, B. Nordström, and A. Ranta. Multilingual Syntax Editing in GF. In A. Gelbukh, editor, *Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico City, February 2003*, volume 2588 of *LNCS*, pages 453–464. Springer-Verlag, 2003.

[12] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS 806, pages 213–237. Springer, 1994.

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

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

[15] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Clarendon Press, Oxford, 1990.

[16] B. Partee. Montague grammar and transformational grammar. *Linguistic Inquiry*, 6:203–300, 1975.

[17] F. Pereira and S. Shieber. *Prolog and Natural-Language Analysis*. CSLI, Stanford, 1987.

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

[19] A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

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

[21] A. Ranta. Grammatical Framework Homepage. `http://www.cs.chalmers.se/~aarne/GF/`, 2000–2003.

[22] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, to appear.

[23] M. T. Rosetta. *Compositional Translation*. Kluwer, Dordrecht, 1994.

[24] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Microwave Research Institute Symposia Series*, 21:19–46, 1970.

[25] S. Shieber. *An Introduction to Unification-Based Approaches to Grammars*. University of Chicago Press, 1986.

[26] M. Steedman. Combinators and grammars. In R. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 417–442. D. Reidel, Dordrecht, 1988.

[27] The Coq Development Team. The Coq Proof Assistant Reference Manual. `pauillac.inria.fr/coq/`, 1999.