Grammatical Framework

A Type-Theoretical Grammar Formalism

AARNE RANTA*

Department of Computing Science
Chalmers University of Technology and the University of Gothenburg
41296 Gothenburg, Sweden.
(*e-mail:* aarne@cs.chalmers.se)

Abstract

Grammatical Framework (GF) is a special-purpose functional language for defining grammars. It uses a Logical Framework (LF) for a description of abstract syntax, and adds to this a notation for defining concrete syntax. GF grammars themselves are purely declarative, but can be used both for linearizing syntax trees and parsing strings. GF can describe both formal and natural languages. The key notion of this description is a grammatical object, which is not just a string, but a record that contains all information on inflection and inherent grammatical features such as number and gender in natural languages, or precedence in formal languages. Grammatical objects have a type system, which helps to eliminate run-time errors in language processing.

In the same way as a LF, GF uses dependent types in abstract syntax to express semantic conditions, such as well-typedness and proof obligations. Multilingual grammars, where one abstract syntax has many parallel concrete syntaxes, can be used for reliable and meaning-preserving translation. They can also be used in authoring systems, where syntax trees are constructed in an interactive editor similar to proof editors based on LF. While being edited, the trees can simultaneously be viewed in different languages.

The paper starts with a gradual introduction to GF, going through a sequence of simpler formalisms till the full power is reached. The introduction is followed by a systematic presentation of the GF formalism and outlines of the main algorithms: partial evaluation and parser generation. The paper concludes by brief discussions of the Haskell implementation of GF, existing applications, and related work.

1 Introduction: the goals of GF

The Grammatical Framework (GF) is a *grammar formalism*, i.e. a language for defining grammars. The development of GF started as a notation for *type-theoretical grammars* (Ranta, 1994), which use Martin-Löf's type theory (1984) to express the semantics of natural language. The first implementation was released in 1998 at

* Thanks to Markus Forsberg, Reiner Hähnle, Kristofer Johannisson, Bengt Nordström, and an anonymous referee for careful reading and valuable comments on this paper. GF itself has benefitted from contributions by many people at XRCE Grenoble, at Chalmers and Gothenburg University, at CAMS in Paris, and in the European TYPES Community. The work has been supported by the VINNOVA foundation within the project "Interactive Language Technology" (2001-06340).

Xerox Research Centre Europe in Grenoble, with focus on multilingual authoring via a type-theoretical pivot language. After the first publication (Mäenpää & Ranta, 1999), GF has developed into a functional programming language, whereby its notation has been completely revised, but it has preserved downward compatibility.

The focus of GF has thus shifted from an initial theoretical idea to practical applications. In this paper, we try to make explicit the theory that has proved useful for practical applications. The goal of GF is to serve both linguists, who want a high-level and reliable grammar formalism, and programmers, who want an elegant and efficient tool for building natural-language applications.

1.1 A Logical Framework with concrete syntax

When describing or implementing a language, it is customary to distinguish between its *abstract syntax*, i.e. the hierarchical structure of the language, and its *concrete syntax*, i.e. what the language looks like as it is read and written. The idea is that notions such as type checking and semantics are better defined on the level of abstract syntax, without the clutter of concrete syntax details.

In programming language design, concrete syntax is usually kept as simple as possible, with some concessions allowed to tradition, e.g. to include standard mathematical notations. In linguistics, the situation is different: the linguist has to take a natural language as it is, and describe it the best she can. It is not common to reach a level at which one can make a neat distinction between abstract and concrete syntax, or to discuss questions like type checking and semantics with the same precision as in programming languages.

GF was born from a synthesis between the computer science and linguistics ways of thinking: what about if we take an abstract syntax, with all type checking and semantics, and try to define a concrete syntax that looks exactly as we want, including natural languages? We took a powerful formalism for abstract syntax, a *Logical Framework* (LF)[1], and extended it with a notation for concrete syntax. With this formalism, it became possible to define all aspects of a language at once, which is an advantage for language implementation. At the same time, it gives a new perspective on natural languages, since it makes precise semantical notions applicable to those fragments of natural language that are recognized by GF grammars.

The practical issue of adding concrete syntax to LF was already addressed in the Mathematical Vernacular project of de Bruijn (1994). The goal was to make LF proof systems more accessible to users. An early implementation of this idea is a program that translates proofs in the logical framework Coq into an English-like notation (Coscoy *et al.*, 1995). However, the translation is part of the implementation of Coq itself: thus, even though the abstract syntax of new mathematical concepts is user-definable in Coq, their concrete syntax is not.

A step towards user-defined concrete syntax is taken in Isabelle (Paulson, 2002), which has a *mixfix* notation to define the concrete syntax of functions. Mixfix is

[1] The logical framework of GF is a version constructive type theory, as are LF (Harper *et al.*, 1993), ALF (Magnusson & Nordström, 1994), and Coq (The Coq Development Team, 1999).

a generalization of infix declarations—which of course are rudimentary concrete-syntax definitions—into full context-free rules. To some extent, mixfix notation is enough even for natural language. For instance, if we want to define an English notation for the length function, whose abstract syntax is given by the function declaration

$$length\ :\ (A\ :\ Set)\ \rightarrow\ List\ A\ \rightarrow\ Int$$

it is enough to write:[2]

$$length\ \_\ x\ =\ "the"\ ++"length"\ ++"of"\ ++\ x$$

However, if we want to express *length* in correct German, we need to inflect it in different cases, put its argument into the dative case prefixed by the preposition *von*, and tell what gender it has. All this is done by a GF definition

$$length\ \_\ =\ f1\ "Länge"\ Fem$$

where *f1* is a concrete syntax function taking care of the details of inflection and argument case. Of course, *f1* itself is user-defined,

$$f1\ :\ Str\ \rightarrow\ Gen\ \rightarrow\ \{s\ :\ Cas \Rightarrow Str\ ;\ g\ :\ Gen\}\ \rightarrow\ \{s\ :\ Cas \Rightarrow Str\ ;\ g\ :\ Gen\}\ =$$
$$\lambda F, G, x\ \rightarrow\ \{s\ =\ \mathbf{table}\ \{c \Rightarrow der\ Sg\ G\ c\ ++F\ ++"von"\ ++x.s\ !\ Dat\}\ ;\ g\ =\ G\}$$

This rule uses another user-defined function, *der*, which gives the inflectional forms of the German definite article.

Rules like the ones for *length* are typically written by persons who work in LF and are experts in the mathematical theories that they are working with. Functions like *f1* and *der* require expertise in linguistics and German grammar. A good division of labour is that functions of the latter kind are provided in *resource grammars* written by linguists and can be taken for granted by those who write applications. This requires a powerful notation permitting high levels of abstraction.

1.2 A grammar formalism with the linearization perspective

Both computer scientists and linguists have developed grammar formalisms—declarative descriptions of language from which language-processing algorithms can be automatically generated. The best-known algorithm is *parsing*, which takes strings into syntax trees (in the case of GF: to functional terms). Many grammar formalisms are designed to permit easy generation of parsers. The rules of GF, however, have their most direct readings in the direction of *linearization*, which takes functional terms into strings. To show that also a parser can be derived from every GF grammar requires a complicated argument.

In computer science, the best-known grammar formalisms are context-free grammars (=BNF) and attribute grammars (Knuth, 1968). BNF is used for describing languages in reports, but language implementations use extensions of the declarative format with *semantic actions* written in a general-purpose programming language.

[2] We use GF notation; Isabelle mixfix does not support argument suppression.

YACC (Johnson, 1975) is the classical model for such formalisms. Because of semantic actions, YACC grammars cannot generally be used for linearization. BNF does not cover the type-checking aspect of languages, whereas attribute grammars are able to do some of it. In YACC, some type checking can be performed in semantic actions, but this easily becomes a mess, and separate phases are recommended instead. In logical frameworks, type checking can be neatly included in abstract syntax definitions, but there are not yet any tools for doing so in practical language implementations.

In linguistics, the tradition closest to GF is Montague grammar (Montague, 1974), which uses simple type theory to express abstract syntax. In a way, GF is just a generic framework for implementing Montague-style grammars extended with dependent types[3]. Another point of reference are the grammar formalisms based on unification. These formalisms include DCG (Pereira & Warren, 1980) and PATR (Shieber, 1986), which are, like GF, pure frameworks, and HPSG (Pollard & Sag, 1994) and LFG (Bresnan, 1982), which have elements of built-in linguistic theory. While postponing the discussion of related work to Section 10, it can be useful to point out the major ways in which GF differs from unification grammar formalisms:

- GF has separate rules for abstract and concrete syntax.
- The primary perspective in GF is linearization.
- GF has a strong type system.
- GF is a functional language.
- GF can integrate semantics with abstract syntax.
- GF supports multilingual grammars.

A common feature between GF and formalisms like PATR and HPSG is the use of records to model complex grammatical objects,

1.3 A multilingual authoring system

GF can be used in batch mode for linearization, parsing, and translation. A new kind of application is inherited from type-theoretical proof editors: syntax editing natural language. This is the application that explains Xerox's interest in the GF project: with a multilingual grammar, the user of GF can edit a document in a language that she does not know, while at the same time seeing how it evolves in her own language. This activity is called *multilingual authoring* (Power & Scott, 1998; Dymetman *et al.*, 2000). For instance, a letter being edited in Swedish may look like this:

Kära [Recipient],
jag har äran att meddela dig att du har blivit befordrad till [Position].

[3] The semantic aspects of this extension are studied in (Ranta, 1994). A strong application of dependent types is the analysis of pronominal reference.

The expressions in brackets are *placeholders*, yet to be filled in to complete the letter. The possible fillings are either choices from menus (generated from the grammar) or strings of English or Swedish text (parsable by the grammar). In parallel with Swedish, an English version may be generated from the same abstract source:

Dear [Recipient],
I have the honour to inform you that you have been promoted to [Position].

The author can be sure that both letters come out grammatically correct and convey the same message. The type-theoretical representation works as a *pivot language*, which controls the consistency of the document, and guarantees that all translations have the same meaning. Interaction eliminates a notorious problem of automatic translation, which is that a source text written in natural language generally does not fully determine the semantic content (cf. Kay (1997)).

A multilingual GF grammar is a simple and efficient way to implement translation, which works surprisingly well, due to the long distance from concrete-syntax details that is possible in abstract syntax. Most other systems use separate *transfer rules* to translate between languages. The obvious disadvantage of transfer rules is that $n(n-1)$ transfer modules are needed to translate between n languages, whereas GF only needs $n+1$ grammar modules. The disadvantage of GF's pivot language method is that translation is limited to be structure-preserving. Even if there is no GF notation to express transfer rules, GF does not preclude them: the API module (Section 8.4) gives support for defining transfer rules in Haskell.

2 Context-free rewrite grammars

This section starts a series of more and more powerful fragments of GF. Most GF concepts and applications already make sense in this first fragment, but only a limited class of GF grammars can be written in it. Sections 3 and 4 extend the abstract syntax part of GF, whereas Section 5 independently extends the concrete syntax part.

2.1 From context-free grammars to context-free rewrite grammars

Context-free rewrite grammars are a generalization of context-free grammars, arising from a distinction between two aspects: abstract syntax and concrete syntax. Consider a context-free rule

$$f.\ C\ ::=\ t_1\ \ldots\ t_m$$

where f is the *rule label*, C is a *category symbol*, and each t_i is either a category symbol (a *nonterminal*) or a string (a *terminal*). The abstract syntax aspect of this rule is a a *function declaration*, declaring f as a function whose value type is C and argument types are the nonterminals $C_1\ \ldots\ C_n$ among $t_1\ \ldots\ t_m$:

$$\mathbf{fun}\ f\ :\ C_1\ \to\ \cdots \to\ C_n\ \to\ C$$

The concrete syntax aspect is a *linearization rule*, which assigns a *linear pattern* to the function f:

$$\textbf{pattern}\ f\ x_1\ \ldots\ x_n\ =\ t_1 ++ \ldots ++ t_m$$

Each t_i is either one of the variables x_j (a nonterminal) or a string (a terminal).

As an example of splitting a context-free rule into a function declaration and a linearization rule, consider the rule

$$\mathit{Div.\ Prop}\ ::=\ \mathit{Exp}\ \text{"}\mathit{is}\text{"}\ \text{"}\mathit{divisible}\text{"}\ \text{"}\mathit{by}\text{"}\ \mathit{Exp}$$

Splitting gives the rule pair

$$\begin{aligned}\textbf{fun}\ \mathit{Div}\ &:\ \mathit{Exp}\ \rightarrow\ \mathit{Exp}\ \rightarrow\ \mathit{Prop}\\ \textbf{pattern}\ \mathit{Div}\ x\ y\ &=\ x ++ \text{"}\mathit{is}\text{"} ++ \text{"}\mathit{divisible}\text{"} ++ \text{"}\mathit{by}\text{"} ++ y\end{aligned}$$

Abstract syntax rules alone define a system of *syntax trees*, i.e. functional terms formed by using rule labels as constants. Given two more function declarations,

$$\begin{aligned}\textbf{fun}\ \mathit{two}\ &:\ \mathit{Exp}\\ \mathit{sum}\ &:\ \mathit{Exp}\ \rightarrow\ \mathit{Exp}\ \rightarrow\ \mathit{Exp}\end{aligned}$$

we can form the syntax tree

$$\mathit{Div}\ (\mathit{sum}\ \mathit{two}\ \mathit{two})\ \mathit{two}$$

whose type is *Prop*. Given the linearization rules

$$\begin{aligned}\textbf{pattern}\ \mathit{sum}\ x\ y\ &=\ \text{"}\mathit{the}\text{"} ++ \text{"}\mathit{sum}\text{"} ++ \text{"}\mathit{of}\text{"} ++ x ++ \text{"}\mathit{and}\text{"} ++ y\\ \mathit{two}\ &=\ \text{"}\mathit{two}\text{"}\end{aligned}$$

we have a correspondence between this tree and the string

the sum of two and two is divisible by two

2.2 Permutation, suppression, and reduplication

To represent context-free rules in context-free rewrite grammar, the full expressive power of linear patterns is not needed, but only the special case in which the sequence of nonterminals in the linear pattern corresponds one-to-one to the arguments of the function. The full format extends this special case in three ways:

- Permutation: constituent order may be changed.
- Suppression: constituents may be hidden.
- Reduplication: constituents may be repeated.

Permutation is important if we want to have concrete syntaxes sharing an abstract syntax. For instance, it permits giving the same abstract syntax to prefix and infix notation, or to adjectival modification in English (prefix: *even number*) and French (postfix: *nombre pair*). Suppression is needed if we want a syntax tree to carry more information than the corresponding string, as in proof-carrying documents (Section 4.3). Reduplication shows that the formalism is more powerful than

context-free grammars: for instance, the copy language of the universal language U (over some alphabet)

$$\{xx | x \in U\}$$

is context-sensitive, but it is encoded by the context-free rewrite grammar

$$\textbf{fun } f : U \to S \text{ ; } \textbf{pattern } f\, x = x +\!+ x$$

2.3 Linearization

To linearize a syntax tree

$$f\ a_1\ \ldots\ a_n$$

the linearization algorithm reads the linear pattern given in the rule

$$\textbf{pattern } f\ x_1\ \ldots\ x_n = t_1 +\!+ \ldots +\!+ t_m$$

and scans the sequence $t_1 +\!+ \ldots +\!+ t_m$ from left to right:

$$(f\ a_1\ \ldots\ a_n)^\circ = s_1 +\!+\ \ldots\ +\!+\ s_m$$

where

$$s_i = \begin{cases} a_j^\circ & \text{if } t_i = x_j \quad \text{(the } j\text{'th argument)} \\ s & \text{if } t_i = s \quad \text{(string)} \end{cases}$$

The algorithm assumes that expressions are in the *full application form*, i.e. functions are endowed by all their arguments: otherwise it is in general not possible to fill the linear pattern in a meaningful way. To express functions, lambda abstraction must be used (Section 3).[4]

2.4 Type checking and syntax editing

The type checker uses abstract syntax to look up the types of functions and verifies that they are used in accordance with their types. Rather than formulating the algorithm explicitly, we give the typing rule of syntax trees:

$$\frac{f : C_1 \to \ldots \to C_n \to C \quad a_1 : C_1 \ldots a_n : C_n}{f\ a_1\ \ldots\ a_n : C}$$

This is of course the same as the full application rule of typed lambda calculus.

Syntax editing takes place in a *state*, which consists of a tree being edited and the subtree that is the current *focus*. The tree may be *incomplete*, that is, contain *placeholders* (also called *metavariables*), which are yet to be filled by subtrees. An example is the following arithmetical proposition, where the focus is marked by an asterisk ($*$) and placeholders by question marks (?):

$$\mathit{Div}\ (\mathit{sum}\ \mathit{two}\ *?)\ ?$$

[4] Full application is usually required in all functional languages whenever other kinds of functions are used than prefixes: one cannot write `(if b then 5 else)` to replace `(\x -> if b then 5 else x)`, and omitting two arguments would lead to the completely bizarre `(if b then else)`. One of the rare exceptions are Haskell's infix sections, such as `(4+)` (equal to `(\x -> 4 + x)`)

The linearization of this tree is

$$\textit{the sum of two and } (* \ ?) \textit{ is divisible by } ?$$

An important aspect of editing in GF is that it is possible to switch between the tree representation and its linearizations, even when the tree is incomplete.

For an efficient implementation of editing commands, we represent trees in a form in which the types of all subtrees are shown. The above tree is then represented

$$\begin{array}{llll} & \mathit{Div} & & : \mathit{Prop} \\ & & \mathit{sum} & : \mathit{Exp} \\ & & & \mathit{two} \ : \mathit{Exp} \\ * & & ? & : \mathit{Exp} \\ & ? & & : \mathit{Exp} \end{array}$$

The editor uses the typing rule for full applications to annotate each subtree with the value type of its function head.

The most important editing command is *refinement*, which replaces the metavariable in focus with a function. If the function takes arguments, like *sum*, refinement introduces new metavariables:

$$\textit{the sum of two and } (* \textit{ the sum of } ? \textit{ and } ?) \textit{ is divisible by } ?$$

The value type of the function must of course match the type of the focus metavariable. The editor guarantees this by maintaining a *menu* of type-correct refinements extracted from the grammar.

The editor focus is analogous to the cursor in a string editor: it marks the place to which editing commands apply. The analogue of cursor movements are *navigation commands*, which shift the focus without changing the tree. GF uses a *zipper* (Huet, 1997) to represent the editor state. As shown by Huet, navigation commands can be implemented efficiently for the zipper. Editing commands are efficient, as well, if the nodes locally contain all information that is needed when executing them—in particular, the types of all subtrees.

Refinement is a purely *top-down* editing command, inherited to GF from the proof editor ALF (Magnusson & Nordström, 1994). The zipper implementation has made it easy to add a generalized *bottom-up* command, the *local wrap*. The local wrap embeds the focus subtree, which need not be a metavariable, in a function application. If the focus is a tree

$$t \ : \ A$$

then any function

$$f \ : \ \cdots \ \rightarrow \ A \ \rightarrow \ \cdots \ \rightarrow \ A$$

can be used to replace t with $(f \ ? \ldots t \ \ldots ?)$; i.e. one argument place of f is filled by t, and the other places by metavariables. For instance, wrapping the focus term of

$$\mathit{Div} \ \mathit{two} \ (* \mathit{one})$$

in the first argument place of $\mathit{sum} \ : \ \mathit{Exp} \ \rightarrow \ \mathit{Exp} \ \rightarrow \ \mathit{Exp}$ results in the tree

$$\mathit{Div} \ \mathit{two} \ (* \mathit{sum} \ \mathit{one} \ ?)$$

In the corresponding English sentence, the word *one* gets embedded in the phrase *the sum of one and* ?. Such an insertion is of course trivial in a text editor working on strings, but not available in most tree-based structure editors. Yet it is a functionality that document authors expect from an editor: they want to make local modifications in a document without having to destroy and rebuild parts of it.

The wrap operation is similar to *tree adjoining* in the grammar formalism TAG (Tree Adjoining Grammars) (Joshi, 1985). A special case is wrapping the top node, which is pure bottom-up editing: there we can relax the requirement that the value type of the function be the same as the type of the focus.

2.5 Parsing

While linearization in context-free rewrite grammars is straightforward, parsing is a search problem. We reduce it to parsing in context-free grammars, which has a complete solution by e.g. the Earley algorithm (Earley, 1970). Context-free parsing is completed by postprocessing that involves a rearrangement of subtrees.

Translation into context-free rules. To each pair of a typing judgement and a linearization rule

$$\begin{cases} \textbf{fun}\ f\ :\ C_1 \to \ldots \to C_n \to C \\ \textbf{pattern}\ f\ x_1\ \ldots\ x_n\ \ =\ t_1\ ++\ \ldots\ ++\ t_m \end{cases}$$

we assign the context-free rule

$$f_p.\ C\ ::=\ c_1\ \ldots\ c_m$$

where

$$c_i = \begin{cases} C_j & \text{if } t_i = x_j \quad \text{(nonterminal)} \\ s & \text{if } t_i = s \quad \text{(terminal)} \end{cases}$$

The function f is indexed by a *profile* p, which is a list of lists of integers

$$[p_1, \ldots, p_n]$$

where

$$p_i\ =\ [j\ \mid\ j \in \{1, \ldots, k\},\ t'_j = x_i]$$

where $t'_1\ \ldots\ t'_k$ is the sequence of nonterminals in the **pattern** clause. In other words, each item p_i in the profile tells what places the ith argument occupies in the linear pattern. For instance, the pair of rules

$$\textbf{fun}\ f\ :\ A\ \to\ B\ \to\ C\ \to\ D\ ;\ \textbf{pattern}\ f\,x\,y\,z\ =\ y\,{+\!+}\text{"}kuin\text{"}\ {+\!+}y\,{+\!+}\text{"}on\text{"}\ {+\!+}z$$

generates the rule

$$f_{[[],[1,2],[3]]}.\ D\ ::=\ B\ \text{"}kuin\text{"}\ B\ \text{"}on\text{"}\ C$$

Postprocessing context-free parse trees. A parse tree produced by the context-free parser may have a wrong number of arguments in wrong places and even in inconsistent ways (because of reduplication). The transformation ^{R} of parse trees into proper functional terms is performed by reference to the profile of the

function head[5]:

$$(f_p\ c_1\ \ldots\ c_m)^R = f\ a_1\ \ldots\ a_n$$

where

$$a_i = \begin{cases} c_k^R & \text{if } k \in p_i \text{ and } x_i \text{ is consistently represented} \\ ? & \text{if } p_i \text{ is empty} \end{cases}$$

If an argument is suppressed in linearization, this operation thus introduces a metavariable to represent it. If different occurrences of an argument are not represented consistently[6], the operation of rearrangement fails.

Context-free parsing. The choice of context-free parsing algorithm is often the main efficiency issue when processing a language. Since GF grammars are implemented as first-class data objects in Haskell, well-known analyses and transformations (Hopcroft & Ullman, 1979) can be applied to them. For instance, even though the Earley algorithm is applicable to all grammars, some grammars may turn out to permit deterministic LR(1) parsing (Knuth, 1965), which can then be chosen for efficiency.

Some pathological rules cause problems for all context-free parsing algorithms. A cyclic rule

$$f.\,C\ ::=\ C$$

generates the infinite sequence of parse trees

$$t,\ f\,t,\ f\,(f\,t),\ f\,(f\,(f\,t)),\ldots$$

for any tree t of type C. Cyclic rules are sometimes generated from innocent-looking GF rules, such as

$$\textbf{fun}\ f\ :\ A\ \to\ C\ \to\ C\ ;\ \textbf{pattern}\ f\,x\,y\ =\ y$$

where the argument x is suppressed in linearization.

2.6 Semantic definitions and computation

The abstract syntax of a GF grammar can be thought of as a semantic model of the language, especially in those cases where GF is used as a syntactic annotation language for a logical framework. In such a model, we often want not only to declare functions but also to define them. To this end, GF has the form of judgement

$$\textbf{def}\ f\,x_1\ \ldots\ x_n\ =\ t$$

where $f\ :\ C_1\ \to\ \cdots\ \to\ C_n\ \to\ C$ and $t\ :\ C$ in the context $x_1 : C_1, \ldots, x_n : C_n$. An example is

$$\textbf{def}\ \mathit{double}\ x\ =\ \mathit{sum}\ x\ x$$

[5] In context-free rewrite grammar, profiles are an optimization that make it possible to avoid lookups in the grammar at the postprocessing stage. In full GF, profiles are indispensable (cf. Section 7.2).

[6] Consistent means here that the trees are the same. In a more general setting, it means that they are unifiable; cf. Section 7.2.

Definitions are used for *computing* trees, and they define thereby a notion of *equality* between trees. This equality is called *semantic equality*, since it does not affect linearization: even though the terms *two* and *double one* are equal by definition, they are linearized as two different strings. We say that strings resulting from semantically equal trees are *paraphrases* of each other.

3 Variable bindings

Many interesting languages have variable-binding operations. For instance, predicate calculus has the universal quantifier $\forall$ forming propositions such as

$$\forall x.x + 0 = x$$

where the variable x is bound in the subformula $x + 0 = x$. The context-free syntax of universally quantified propositions is

$$Prop \ ::= \ "\forall" \ Var \ "." \ Prop$$

This rule, however, does not capture the fact that the variable is bound in the subformula. The two parts of the structure, *Var* and *Prop*, are not constituents in the same sense: the *Prop* is an argument, whereas the *Var* is a binding. The distinction between arguments and bindings is fundamental for abstract syntax operations, such as type checking and computation—and in particular for syntax editing—and has to be stated somewhere, either by separate rules, or by using *higher-order abstract syntax* instead of context-free syntax. We will now explain how GF implements higher-order abstract syntax and corresponding concrete syntax.

3.1 The abstract syntax of bindings

In higher-order abstract syntax, variable-binding operators are treated as functions that take functions as arguments. For instance, the universal quantifier has just one argument: a function from expressions to propositions:

$$\textbf{fun} \ Univ \ : \ (Exp \ \rightarrow \ Prop) \ \rightarrow \ Prop$$

This function has a second-order type. The general form of a type is now

$$A_1 \ \rightarrow \ \cdots \rightarrow \ A_n \ \rightarrow \ C$$

where each A_i is a type and C is a category[7]. Objects of function types can be formed by λ-abstraction, of the form

$$\lambda x_1, \ldots, x_n \ \rightarrow \ b$$

We require that the number of λ-bound variables be the same as the number of argument types; this is known as the η *long normal form* of λ-terms.

[7] We could say: C is a type, but the resulting notion of type would be equivalent. As we formulate it now, we emphasize that each type has a basic type as its value type.

For example, the formula $\forall x.x + 0 = x$ has the syntax tree

$$Univ(\lambda x \ \rightarrow \ Eq \ (sum \ x \ Zero) \ x)$$

3.2 The concrete syntax of bindings

In the η long normal form, every subtree whose type is a function type has a bound variable for each of its argument types. We collect the variable symbols $x_1, \ldots, x_n$ and the linearization b of the body into a *record*

$$\{v_1 \ = \ x_1 \ ; \ \ldots, \ ; \ v_n \ = \ x_n \ ; \ s \ = \ b\}$$

which is the linearization of the whole tree. We use the record label s for the body, and the labels $v_1, v_2, v, \ldots$ for the variables; if there is just one variable, we use v.

For example, the linearization of $\lambda x \ \rightarrow \ Eq \ (sum \ x \ Zero) \ x$ is

$$\{v \ = \ "x" \ ; \ s \ = \ "x" \ ++ "+" \ ++ "0" \ ++ "=" \ ++ "x"\}$$

The linearization rule of the function *Univ* is

$$\textbf{lin} \ UnivP \ = \ \{s \ = \ "\forall" \ ++ \ P.v \ ++ \ "." \ ++ \ P.s\}$$

The general form of a linearization rule is now

$$\textbf{lin} \ f \ x_1 \ \ldots \ x_n \ = \ \{s \ = \ t_1 \ ++ \ldots ++ \ t_m\}$$

where each t_i has one of the forms "*foo*" (terminal string), $x_j.s$ (nonterminal body), $x_j.v_k$ (nonterminal variable). The linearization algorithm distinguishes three cases:

application:	$(f \ a_1 \ \ldots \ a_n)^o$	$=$	$t(x_1 := a_1^o, \ldots, x_n := a_n^o)$ if **lin** $f \ x_1 \ \ldots \ x_n \ = \ t$
abstraction:	$(\lambda z_1 \rightarrow \cdots \rightarrow \lambda z_n \rightarrow b)^o$	$=$	$\{v_1 = `z_1'; \ldots; v_n = `z_n'\} \ ** \ b^o$
variable:	x^o	$=$	$\{s = `x'\}$

This definition uses an operation $**$ for conjoining records, and a substitution operation $(x := a)$. It also presupposes a symbol-printing operation producing a string 'x' from a variable symbol x.[8] The full normal form of linearization is obtained by these rules, substitutions, and the record projection rule

$$\{\ldots; r \ = \ t; \ldots\}.r = t$$

Using records instead of strings as values of linearization is crucial for maintaining the *compositionality* of linearization: for each function f, the linearization rule assigns a concrete-syntax function f' such that

$$(f \ a_1 \ \ldots \ a_n)^o \ = \ f' \ a_1^o \ \ldots \ a_n^o$$

[8] One case is missing from this definition: variable applied to arguments. This case is needed if the abstract syntax uses third- or higher-order functions. GF then produces an *ad hoc* linearization where the symbol x is prefixed to the linearizations of the arguments in parentheses. In practice, functions of higher order than the second are rare in abstract syntax; strictly speaking, however, one should conclude that GF only supports user-defined concrete syntax for second-order abstract syntax.

where we denote the linearization of a tree t by t°. Thus the linearization of a tree depends only on the *linearizations* of its subtrees, not on the subtrees themselves. Compositionality guarantees a natural correspondence between abstract and concrete syntax. It also helps to make the implementation of GF efficient (Section 5.4).

The **pattern** format used for context-free rewrite grammars is a special case of the **lin** format, as defined in Section 6.4.2.

3.3 Parsing bindings

From the context-free point of view, variable bindings are constituents: the parser reading the input looks for items of a certain shape (e.g. "x") that match a particular nonterminal *Var*. The context-free rule generated from *Univ* looks as follows:

$$Univ_{[([[1]],[2])]}\ Prop\ ::=\ "\forall"\ Var\ "."\ Prop$$

The right-hand-side is as expected. The profile now contains information about where in the parse tree the bound variables are found: we extend the profiles of Section 2.5 to lists of pairs (b, c) where c is a profile item in the old sense, telling what places the constituent occupies, and b is a list of items telling what places each of the bound variables occupy.

When producing context-free grammars. we introduce a category *Var* distinct from all categories in the context-free grammar, with some rules for recognizing variables. We add a rule

$$var_C.\ C\ ::=\ Var$$

for each category C. Finally, when postprocessing the parse tree, a suppressed binding (i.e. one with the profile item []) is not replaced by a metavariable (?), but by a fresh ordinary variable. Notice that the unification phase of postprocessing is first-order, since variables in bindings are treated as ordinary arguments.

3.4 Type checking and syntax editing bindings

Type checking syntax trees with bindings is the same thing as monomorphic type checking in simply typed lambda calculus. In addition to the full application rule of Section 2.4, we have the full abstraction rule

$$\frac{\begin{array}{c}(x_1\ :\ A_1,\ \ldots,\ x_n\ :\ A_n)\\ c\ :\ C\end{array}}{\lambda x_1, \ldots, x_n \to c\ :\ A_1 \to \ldots \to A_n \to C}$$

For syntax editing, we continue to use the zipper. Nodes are extended to contain the actual bindings. Each binding shows the type of the variable, which makes it easy to look up the type. For instance, a tree for the logical formula

$$\forall x.x = x$$

looks as follows:

$$
\begin{array}{ll}
Univ & : Prop \\
\quad (x\ : Exp)\ Equal & : Prop \\
\quad\quad x & : Exp \\
\quad\quad x & : Exp
\end{array}
$$

4 Dependent types

Dependent types are types that depend on objects. We will give examples of two uses of them: first a grammar that defines the type checker of a small programming language, and secondly a grammar of proofs, which leads to the notion of *proof-carrying documents.*

4.1 Typed expressions

The following judgements define a category Typ of datatypes, and the category Exp of expressions, which depends on Typ.

$$
\begin{array}{ll}
\textbf{cat} & Typ \\
 & Exp\ Typ
\end{array}
$$

Examples of datatypes are integers, booleans, and lists:

$$
\begin{array}{lll}
\textbf{fun} & Int,\ Bool & :\ Typ \\
 & List & :\ Typ \rightarrow Typ
\end{array}
$$

To define expressions, we use generalized function types where the value type depends on the argument[9]:

$$
\begin{array}{rl}
\textbf{fun}\ Zero & :\ Exp\ Int \\
True & :\ Exp\ Bool \\
Nil & :\ (A\ :\ Typ) \rightarrow Exp\ A \\
Cons & :\ (A\ :\ Typ) \rightarrow Exp\ A \rightarrow Exp\ (List\ A) \rightarrow Exp\ (List\ A) \\
append & :\ (A\ :\ Typ) \rightarrow (\_,\_\ :\ Exp\ (List\ A)) \rightarrow Exp\ (List\ A)
\end{array}
$$

For example,

$$Cons\ Int\ Zero\ (Nil\ Int)$$

is a valid syntax tree of type $Exp\ (List\ Int)$, whereas

$$Cons\ Int\ Zero\ (Cons\ Bool\ True\ (Nil\ Int))$$

is not a valid tree of any type. The grammar thus expresses not only the syntactic well-formedness of the language but also its well-typedness.

Notice how dependent types define the functions *Nil*, *Cons*, and *append* as *polymorphic*: their *Exp* arguments can be expressions of any types, in virtue of the

[9] In all variable-binding constructions of GF, the wildcard _ can serve as a bound variable, if the variable is not used.

Typ argument. The same technique is used in *monomorphic type theory* (Nordström *et al.*, 1990) and in the dependently typed programming language Cayenne (Augustsson, 1998). Polymorphism in concrete syntax results from argument suppression: for instance, the Haskell notation for lists is defined by the rules

$$\begin{aligned}\textbf{lin}\ Nil\_ &= \{s = "[]"\}\\ Cons\_x\,y &= \{s = x.s ++ ":" ++ y.s\}\\ append\_x\,y &= \{s = "(" ++ x.s ++ "++" ++ y.s ++ ")"\}\end{aligned}$$

When completed with semantic definitions,

$$\begin{aligned}\textbf{def}\ append\_(Nil\_)\,y &= y\\ append\,A\,(Cons\_a\,x)\,y &= Cons\,A\,a\,(append\,A\,x\,y)\end{aligned}$$

the grammar defines a complete parser, pretty-printer, type checker, interpreter, and syntax editor for this little language[10].

4.2 Curry-Howard isomorphism

The definition of type-correct expressions in abstract syntax is a fairly simple application of dependent types. A more involved one, and originally the main motivation of logical frameworks, is to define logical calculi by formulating inference rules as declarations of proof functions. The structure is the same as with types and expressions: we have a basic type *Prop* of propositions and the dependent type *Proof A* of proofs of a proposition A. The idea to treat propositions as types of proofs is known as the *Curry-Howard isomorphism.* An example is implication *à la* Martin-Löf (1984): the formation, introduction, and elimination rules come out as follows:

$$\begin{aligned}\textbf{fun}\ Impl &: Prop \to Prop \to Prop\\ ImplI &: (A, B : Prop) \to (Proof\,A \to Proof\,B) \to Proof(Impl\,A\,B)\\ ImplE &: (A, B : Prop) \to Proof(Impl\,A\,B) \to Proof\,A \to Proof\,B\end{aligned}$$

If we now want to express formal proofs in natural language, we simply give linearization rules that produce texts, e.g.

$$\begin{aligned}\textbf{lin}\ ImplI\ A\,B\,b = \{s = &"assume" ++ A.s ++ "." ++ b.s ++ "." ++\\ &"Hence" ++ "if" ++ A.s ++ "then" ++ B.s\}\end{aligned}$$

On the top level of mathematical texts, we use a category *Text* for textual units such as theorems with or without proofs:

$$\begin{aligned}\textbf{fun}\ &ThmProof, ThmOmit : (A : Prop) \to Proof\,A \to Text\\ \textbf{lin}\ &ThmProof\ A\,a = \{s = "Theorem." ++ A.s ++ "Proof." ++ a.s ++ "QED"\}\\ &ThOmit\ A\_ = \{s = "Theorem." ++ A.s ++ "Proof." ++ "Omitted."\}\end{aligned}$$

[10] In GF's parameter system (Section 5), we could moreover define a precedence parameter to regulate the use of parentheses.

The typing of these functions forces the proof in *ThmProof* really to prove the theorem; in *ThmOmit*, a proof must exist even though it is not shown. Thus anyone who uses the GF syntax editor to build proof texts is forced to making them correct.

4.3 Proof-carrying documents

Besides mathematical texts, dependent types and the Curry-Howard isomorphism are useful for other kinds of texts, to guarantee semantic properties. Consider, for instance, texts describing train connections:

To get from Gothenburg to Hamburg, first take train 487 to Copenhagen and then change to train 36.

The semantic well-formedness conditions for this text are that train 487 runs from Gothenburg to Copenhagen, that train 36 runs from Copenhagen to Hamburg, and that train 487 arrives at Copenhagen before train 36 leaves. All of these conditions can be concisely expressed by a grammar with a dependent type $Train\,A\,B$ of train connections from the city A to the city B, and a type of proofs of the fact that one train arrives before another train leaves. New connections are generated by the rule

$$\textbf{fun}\ Connect\ :\ (A, B, C\ :\ City)\ \rightarrow (a\ :\ Train\,A\,B)\ \rightarrow\ (b\ :\ Train\,B\,C)\ \rightarrow\ Before\,A\,B\,C\,a\,b\ \rightarrow\ Train\,A\,C$$

It is easy to write a linearization rule for *Connect* generating texts like the example above. Linearization hides the proof of the *Before* condition, but anyone who uses the GF syntax editor to build the text is obliged to give a proof in order for the text to be complete. We call this idea *proof-carrying documents*, with a reference to proof-carrying code (Necula, 1997).

4.4 Concrete syntax and dependent types

Little need be said about the concrete syntax of dependent types, since linearization rules look precisely the same as without them. In parsing rules, the arguments of dependent categories are just ignored. The context-free parsing phase thus ignores type dependencies. It accepts ill-formed expressions such as "0 : *True* : []", which the subsequent type checking phase rejects.

It is possible to improve the performance of the GF parser by integrating parsing and type checking: errors are then detected at an earlier stage. Some amount of integration is necessary if the grammar has syntactically dummy *coercion rules* like

$$\textbf{fun}\ coerce\ :\ Exp\,Int\ \rightarrow\ Exp\,Float\,;\ \textbf{lin}\ coerce\,x\ =\ x.$$

The corresponding context-free rule is cyclic,

$$coerce.\ Exp\ ::=\ Exp,$$

and produces an infinity of parse trees, at most one of which is type-correct[11].

[11] Luo and Callaghan (1999) investigate coercion as a central phenomenon of informal mathematical language and suggest an algorithm for resolving it

4.5 Type checking and syntax editing dependent types

Type checking with dependent types is harder than without them, since it involves computation of expressions. For instance, a proof that 2 is even is also a proof that $1 + 1$ is even. In the presence of variables and metavariables, moreover, it cannot always be decided if an expression has a given type: whether a proof of $Even\,2$ is also a proof of $Even\,(1+?)$ depends on the value of ?. Therefore, what the type checker returns is not a boolean value but a set of *constraints*, which are equalities between terms. The value *True* corresponds to the empty set of constraints. The value *False* corresponds to the situation where some of the constraints is impossible, e.g. $1 = 0$ (cf. (Magnusson & Nordström, 1994)).

In general, constraints contain metavariables that appear in different positions in the tree. Because of this, metavariables have to carry unique identifiers; we use subindexed question marks for this. For instance, parsing the expression

$$0 : 1 : [\,]$$

in the grammar of Section 4.1 creates an incomplete term of an incomplete type:

$$Cons\,?_0\;Zero\,(Cons\,?_1\;One\,(Nil\,?_2))\;:\;Exp\,?_3$$

The type checker can easily find out the following constraints:

$$?_0\;=?_1\;=?_2\;=\;Int,\;?_3\;=\;List\,?_0$$

In this example, a simple constraint-solving mechanism is enough to automatically instantiate all the metavariables. Such is usually the case for hidden type arguments corresponding to polymorphism: the user of an editor does not need to fill in these arguments.

Even if constraints remain unsolved, they can be helpful in syntax editing. since they narrow down available choices. For instance, in a menu of refinements for $? \;:\; Exp\,Bool$, functions whose value type is $Exp\,Int$ are not shown.

Formal rules for dependent types are given in Section 6.2. As type checking algorithm, we have used the one in (Coquand, 1996), which we have modified so that it type-annotates terms into trees used by the zipper editor. In addition to bindings, function body, and value type, as in Section 3.4, the information stored in a node includes the constraints created when type checking that node.

5 Extending concrete syntax

The values returned by linearization have so far been strings and records of strings. In this section, we generalize this to records that may also contain string-valued finite functions, *tables*, as well as *parameters*. Different concrete syntaxes may use different record types for one and the same type in the abstract syntax. This extension is essential to keep abstract syntax independent of language-dependent features such as inflection.

5.1 Parameters, tables, and records

A *parameter type* is a finite set of *parameter values*, on which the linearization of an expression may depend. For example, the type of grammatical numbers in English has two values: the singular and the plural. Expressions linearized as English common nouns have two forms: e.g. *Int* has the singular form *integer* and the plural form *integers*.

We write

$$\textbf{param}\ Num\ =\ Sg\ \mid\ Pl$$

to define the parameter type *Num*. The type

$$Num\ \Rightarrow\ Str$$

is the type of string-valued tables on *Num*, and the expression

$$\textbf{table}\ \{Sg\ \Rightarrow\ "integer"\ ;\ Pl\ \Rightarrow\ "integers"\}$$

gives such a table in explicit form. The *selection* operation (!) is used for applying tables to arguments:

$$\textbf{table}\ \{Sg\ \Rightarrow\ "integer"\ ;\ Pl\ \Rightarrow\ "integers"\}\ !\ Pl = "integers"$$

Concrete syntax assigns to every category in abstract syntax a *linearization type*: for instance, the linearization type of *CN* is given by the judgement

$$\textbf{lincat}\ CN\ =\ \{s\ :\ Num\ \Rightarrow\ Str\}$$

Linearization rules for expressions of the category *CN* must have this value type. An example of such a rule is

$$\textbf{lin}\ Int\ =\ \{s\ =\ \textbf{table}\ \{Sg\ \Rightarrow\ "integer"\ ;\ Pl\ \Rightarrow\ "integers"\}\}$$

In German, common nouns do not only depend on number, but also on case (Nominative, Accusative, Genitive, Dative). The linearization type thus has a two-argument table, which we "curry" into a table of tables:

$$\textbf{lincat}\ CN\ =\ \{s\ :\ Num\ \Rightarrow\ Case\ \Rightarrow\ Str\}$$

In yet other languages, there may be three numbers (Arabic has the dual) or fifteen cases (Finnish has—well...). In context-free (rewrite) grammars, all this variation would have to be expressed by unrelated rules, which would make it impossible to use a common abstract syntax.

In addition to parameters that produce different forms, expressions may have parameters as *inherent features*. For instance, German common nouns have a gender (Masculine, Feminine, Neuter) associated to them, but not as inflection forms: any noun inherently has just one gender. Inherent features are expressed by record fields in linearization types. Here is an amended rule for German common nouns:

$$\textbf{lincat}\ CN\ =\ \{s\ :\ Num\ \Rightarrow\ Case\ \Rightarrow\ Str\ ;\ g\ :\ Gen\}$$

The record linearizing a tree contains all linguistic information concerning the expression: its inflection table and its inherent features. Such information is what

we normally find in dictionaries. In grammar, this information is not only needed for individual words, but for arbitrarily complex phrases. For instance, in English, when a common noun is modified by an adjective, the resulting complex common noun can still be inflected in number:

$$\begin{aligned}&\textbf{fun}\ \mathit{Mod} : \mathit{Adj} \to \mathit{CN} \to \mathit{CN}\\ &\textbf{lin}\ \mathit{Mod}\ F\ A = \{s = \textbf{table}\ \{\mathit{Sg} \Rightarrow F.s ++ A.s\,!\,\mathit{Sg}\,;\ \mathit{Pl} \Rightarrow F.s ++ A.s\,!\,\mathit{Pl}\}\}\end{aligned}$$

All **fun** rules must have **lin** rules of matching linearization types. This can be checked at compile time, before the grammar is used. It is also easy to check that the rules are complete—essentially, that all tables have values for all elements of their argument types. Using a type system to prevent run-time errors is one of the key ideas that GF has inherited from functional programming languages.

5.2 Hierarchical parameters

Parameter types are like `data` types in Haskell and other functional languages, with the restriction that they must be finite. *Hierarchical* parameter types are permitted, and they are, in fact, often very appropriate. To give an example, French verbs, as presented in the authoritative *Bescherelle* (1997), have three persons, two numbers, two genders, four (non-composite) tenses, and six modes. But the inflection tables display only 51 (non-composite) verb forms, not 288, which would be the case if the forms were simply cross-products of all parameters. The reason is that many combinations do not exist. A natural way of describing this parameter system is by using parameter types whose constructors have arguments from other parameter types. The following system is a straightforward GF formalization of the *Bescherelle*:[12]

$$\begin{aligned}\textbf{param}\ \mathit{Nombre} &= \mathit{Sg} \mid \mathit{Pl}\\ \mathit{Personne} &= \mathit{P1} \mid \mathit{P2} \mid \mathit{P3}\\ \mathit{Genre} &= \mathit{Masc} \mid \mathit{Fem}\\ \mathit{Temps} &= \mathit{Pres} \mid \mathit{Imparf} \mid \mathit{Passe} \mid \mathit{Futur}\\ \mathit{TSubj} &= \mathit{SPres} \mid \mathit{SImparf}\\ \mathit{TPart} &= \mathit{PPres} \mid \mathit{PPasse}\ \mathit{Genre}\ \mathit{Nombre}\\ \mathit{NImper} &= \mathit{SgP2} \mid \mathit{PlP1} \mid \mathit{PlP2}\\ \mathit{VForm} &= \mathit{Inf} \mid \mathit{Indic}\ \mathit{Temps}\ \mathit{Nombre}\ \mathit{Personne} \mid \mathit{Cond}\ \mathit{Nombre}\ \mathit{Personne}\\ &\quad \mid \mathit{Subj}\ \mathit{TSubj}\ \mathit{Nombre}\ \mathit{Personne} \mid \mathit{Imper}\ \mathit{NImper} \mid \mathit{Part}\ \mathit{TPart}\end{aligned}$$

[12] We have used this type system in a complete GF implementation of the *Bescherelle* conjugations; see GF Homepage (Ranta, 2002). Huet (2000) uses CAML datatypes in the same way in his morphology of Sanskrit.

5.3 Discontinuous constituents

In all examples so far, the linearization of a tree has been a record with one principal string or string-valued table, stored in a field labelled s. Now we consider cases where the linearization consists of separate parts, which can change order and get other expressions inserted between them. Such expressions are called *discontinuous constituents.*

A famous example of discontinuous constituents is German verb phrases. A verb phrase is a complex expression consisting of a verb and its complements, in English e.g. *loves Mary* in *John loves Mary.* The analysis of a sentence into a noun phrase (the subject) and a verb phrase (the predicate) is motivated both by logic (Aristotelian or modern) and by linguistic facts such as the conjunction *John loves Mary and hates Bill.* In German, however, the verb phrase (*liebt Maria*) cannot be found in all uses of the sentence (*Johann liebt Maria*). For instance, in the conditional

wenn Johann Maria liebt, liebt Johann Maria

the verb phrase is used with a reverse word order in the antecedent, and dissolved into two parts in the succedent. Those linguists who still believe that *liebt Maria* is a constituent of the sentence, have to treat it as a discontinuous constituent.

In GF, discontinuous constituents are records with more than one string-valued fields. The German linearization type of verb phrases can be defined as

$$\textbf{lincat}\ VP\ =\ \{s_1\ :\ Agr\ \Rightarrow\ Str\ ;\ s_2\ :\ Str\}$$

consisting of the verb part s_1 and the complement part s_2. The verb part depends on agreement features, such as number and person. It receives them from the subject of the sentence, which has them as inherent features. The sentence-forming predication rule

$$\textbf{fun}\ Pred\ :\ \mathrm{NP}\ \rightarrow\ VP\ \rightarrow\ S$$

is linearized under a three-valued parameter that produces different strings for direct, inverse, and subordinate sentences:

$$\begin{aligned}\textbf{lin}\ Pred\,N\,V\ =\ \{s\ =\ \textbf{table}\ \{&Dir\ \Rightarrow\ N.s\ ++V.s_1\ !\ N.a\ ++V.s_2\ ;\\ &Inv\ \Rightarrow\ V.s_1\ !\ N.a\ ++N.s\ ++V.s_2\ ;\\ &Sub\ \Rightarrow\ N.s\ ++V.s_2\ ++V.s_1\ !\ N.a\}\}\end{aligned}$$

The complementation rule forms a verb phrase from a transitive verb (TV) and a noun phrase:

$$\begin{aligned}\textbf{fun}\ Compl\ &:\ TV\ \rightarrow\ NP\ \rightarrow\ VP\\ \textbf{lin}\ Compl\,V\,N\ &=\ \{s_1\ =\ V.s\ ;\ s_2\ =\ N.s\ !\ Acc\}\end{aligned}$$

Given the noun phrases *Johann* and *Maria* and the transitive verb *Lieben*, we can form the syntax tree

$$Pred\,Johann\,(Compl\,Lieben\,Maria)$$

which has a linearization producing three forms: *Johann liebt Maria*, *liebt Johann Maria*, and *Johann Maria liebt.*

Parsing discontinuous constituents will be explained as a part of the full GF parsing algorithm (Section 7.2). We just notice that discontinuous constituents make it possible to define intricate non-context-free languages, such as

$$\{a^n b^n c^n \mid n = 1, 2, \ldots\}$$

This language is defined by the category S of the following GF grammar:

$$\begin{aligned}
\textbf{cat}\ & S\ ;\ Aux \\
\textbf{fun}\ & exp\ :\ Aux \to S\ ;\ first\ :\ Aux\ ;\ next\ :\ Aux \to Aux \\
\textbf{lincat}\ & Aux\ =\ \{s_1\ :\ Str\ ;\ s_2\ :\ Str\ ;\ s_3\ :\ Str\} \\
\textbf{lin}\ & exp\, x\ =\ \{s\ =\ x.s_1 ++ x.s_2 ++ x.s_3\} \\
& first\ =\ \{s_1\ =\ "a"\ ;\ s_2\ =\ "b"\ ;\ s_3\ =\ "c"\} \\
& next\, x\ =\ \{s_1\ =\ "a" ++ x.s_1\ ;\ s_2\ =\ "b" ++ x.s_2\ ;\ s_3\ =\ "c" ++ x.s_3\}
\end{aligned}$$

5.4 Canonical GF

We have extended context-free grammars into grammars where linearizations of syntax trees are records of tables of grammatical objects. The operational semantics of these grammars will be explained in terms of computation rules for table selections and record projections in Section 6.3. From this perspective, linearization is similar to evaluation in a functional programming language.

However, GF has a computational model that is simpler than evaluation in functional language, since it does not involve substitutions for variables. The only variables that are present in the right-hand-side t of a linearization rule

$$\textbf{lin}\ f\ x_1\ \ldots\ x_n\ =\ t$$

are $x_1\ \ldots\ x_n$, which stand for the linearizations of the arguments of f. The substitution of values for these variables can be performed in the same way as selections and projections: as look-up followed by simple replacement. Linearization as a whole is a single inorder traversal of the syntax tree.

We will refer to the GF concrete-syntax notation so far introduced as *canonical GF*. In the next section, we will go far beyond canonical GF by adding functions and pattern matching. This extension is important for the usability of GF. For the implementation, however, the important thing is that the rich notation can be compiled back into canonical GF. Even though linearization could be performed directly as evaluation on the rich notation, it is much more efficient to perform *partial evaluation* on the grammar and use canonical GF at runtime. Moreover, it is from canonical GF that parsers are derived. Partial evaluation and parsing will be explained in Section 7.

5.5 Abstraction mechanisms

Linguists, just like functional programmers, like to work with strong generalizations and on a high level of abstraction. GF makes accessible to linguists two abstraction mechanisms of functional programming: *function definitions* and *pattern matching*.

Function definitions in GF are called *operation definitions* to distinguish them from the **fun** judgements of abstract syntax. An example is the operation that produces regular common nouns in English:

$$\textbf{oper}\ regCN : Tok \to \{s : Num \Rightarrow Str\} = \lambda c \to \{s = Sg \Rightarrow c\,;\ Pl \Rightarrow c{+}"s"\}$$ [13]

The linearization rule of the datatype expression *Int* can now be concisely written

$$\textbf{lin}\ Int = regCN\ "integer"$$

Pattern matching is used in tables: branches can be defined not only for constructor expressions, but also for *patterns*, which may contain variables and wildcards (_). For instance, the following table defines the English adjectival modification rule by using a pattern variable n for number:

$$\textbf{lin}\ Mod\ F\ A = \{s = \textbf{table}\ \{n \Rightarrow F.s\ ++ A.s\ !\ n\}\}$$

It is possible to expand this table into the fully explicit form shown in Section 5.1; using patterns, however, captures the generalization that it is the noun part that receives the number of the whole phrase.

Function types ($A \to B$) and table types ($A \Rightarrow B$) have many common properties: both allow currying, full and partial application, and formation by abstraction. There are important differences, however:

- Tables, but not functions, are restricted to finite argument types.
- Tables, but not functions, can be formed by case analysis.[14]
- Tables, but not functions, are values in canonical GF.

The partial evaluation algorithm (Section 7.1) shows in fact that

- Functions can always be eliminated from linearization rules.

5.6 Resource grammars

The intended use of GF is to build natural-language fragments on top of semantic models, such as mathematical theories. This makes it possible to minimize the size of grammars and avoid many irrelevant linguistic problems. For instance, a French grammar for mathematics does not need to define all the 51 French verb forms, but two is often enough.

However, the *ad hoc* way of defining grammars may lead to duplication of work: if different parts of verb conjugation are needed in different applications, one cannot use the conjugation defined for one grammar as a resource for another grammar. And, of course, this style of grammar-writing favours linguistically unmotivated solutions.

The idea of *resource grammars* is to define common and unproblematic parts of concrete syntax—such as inflection tables—independently of abstract syntax. Using

[13] We use + instead of ++ between strings to say that they belong to the same token (*Tok*).
[14] If the argument type of a function f is a parameter type, case analysis is of course possible in the form $f = \lambda x \to \textbf{table}\ \{\ldots\}\ !\ x$.

a resource grammar needs some care, however: a grammar encoding 51 forms of one thousand verbs is a heavy tool for actually dealing with two forms of ten verbs. When used in a naïve way, it produces enormous runtime systems. What makes resource grammars practical is type-driven partial evaluation (Section 7.1). Suppose we only need two forms of French verbs—say, the indicative and subjunctive of third person singular present tense. The linearization type of verbs is then

$$\textbf{lincat}\ \mathit{Verb} = \{s : \mathit{Mode} \Rightarrow \mathit{Str}\}$$

Assume that we have a resource grammar with complete forms of conjugation *à la Bescherelle*, in a form like

$$\textbf{oper}\ \mathit{tenir} : \mathit{VForm} \Rightarrow \mathit{Str} = t$$

where *VForm* is the parameter type with 51 values. When evaluated, t yields the full table for the verb *tenir* (cf. Section 5.2). Now, to use objects of this type as linearizations of verbs in the category *Verb*, all we need is an interface operation

$$\begin{aligned}&\textbf{oper}\ \mathit{useVerb} : (\mathit{VForm} \Rightarrow \mathit{Str}) \to \{s : \mathit{Mode} \Rightarrow \mathit{Str}\} = \lambda t \to\\ &\quad \{s = \textbf{table}\ \{\mathit{Ind} \Rightarrow t\,!\,(\mathit{Indic\,Pres\,Sg\,P3})\ ;\ \mathit{Sub} \Rightarrow t\,!\,(\mathit{Subj\,Pres\,Sg\,P3})\}\}\end{aligned}$$

Linearizations can then be defined compactly, for instance,

$$\textbf{lin}\ \mathit{Tenir} = \mathit{useVerb}\ \mathit{tenir}$$

and the result is a two-element table with the forms *tient, tienne*, since the term is η-expanded with respect to the expected linearization type and then evaluated. If some other set of forms is needed, all that has to be changed is the definition of the interface operation *useVerb*.

Resource grammars are an obvious way to define morphology and lexicon, and they can often be compiled from existing resources or created by using general-purpose programming languages. But the idea also makes sense for syntax. For instance, the German linearization types for sentences, noun phrases, verb phrases, and transitive verbs, and the predication and complement rules (Section 5.3) can be written as operations:[15]

$$\begin{aligned}\textbf{oper}\ S &: \mathit{Type} = \{s : \mathit{Ord} \Rightarrow \mathit{Str}\}\\ NP &: \mathit{Type} = \{s : \mathit{Case} \Rightarrow \mathit{Str}\,;\ n : \mathit{Agr}\}\\ VP &: \mathit{Type} = \{s_1 : \mathit{Agr} \Rightarrow \mathit{Str}\,;\ s_2 : \mathit{Str}\}\\ TV &: \mathit{Type} = \{s : \mathit{Agr} \Rightarrow \mathit{Str}\}\\ \mathit{Pred} &: NP \to VP \to S = \ldots\\ \mathit{Compl} &: TV \to NP \to VP = \ldots\end{aligned}$$

[15] If the linguist prefers to write her grammar using **fun**, **cat**, **lin**, and **lincat**, as in Section 5.3, the **oper** definitions can be extracted automatically from them.

More functions can be defined in terms of these basic operations:

$$Pred1 : VP \to NP \to S = \lambda F, x \to Pred\ x\ F$$
$$Pred2 : VP \to NP \to NP \to S = \lambda F, x, y \to Pred\ x\ (Compl\ F\ y)$$

The writer of an application grammar for e.g. mathematics can use these operations without knowing anything about German word order and agreement. If she has decided that propositions are linearized as S, one-place predicates as VP, and two-place predicates as TV, all she has to know is which verbs (from the resource grammar) are used for each predicate[16]. For instance, to linearize the one-place convergence predicate and the two-place intersection predicate, she writes

$$\textbf{lin}\ Converge = Pred1\ konvergieren$$
$$Intersect = Pred2\ schneiden$$

In this way, a division of labour is achieved between authors of resource grammars, who are experts in linguistic rules, and authors of application grammars, who are experts in the domain of application.

6 The GF language

This section gives a concise definition of the GF formalism. The framework-level notions of type checking and evaluation are specified by inference rules. The notation we use is exactly the same as the notation recognized by the GF parser, with the exeption of a handful of non-ASCII symbols: in ASCII-written GF source code, we replace λ by \, $\to$ by `->`, and $\Rightarrow$ by `=>`.[17]

6.1 Grammars and judgements

A GF grammar is a sequence of *judgements*. Judgements are divided into two sorts: those of abstract syntax and those of concrete syntax. Figure 1 shows the forms of judgement used in GF grammars, together with their verbal readings.

Every form of judgement has a *keyword* (such as **cat**, **param**). Every judgement ends with a semicolon (;), which we usually omit in typeset text, where we have access to layout. Using the semicolon (or layout) makes it possible to omit keywords: once a keyword appears in the code, it is read as the first word of every semicolon-separated judgement, until a keyword is encountered again.

The forms of judgement shown in Figure 1 are the ones that may appear in GF grammars. On the metalevel, we also use judgements of the forms

$A : Type$	A is a type
$a : A$	a is an object of type A
$a = b$	a is definitionally equal to b

[16] Including adjectives as possible linearizations of predicates would not be a problem: the adjective-verb distinction could be hidden in slightly more general linearization types.

[17] Some GF structures that are supported by the actual implementation are left out, since we consider them experimental; we refer to documentation in (Ranta, 2002) for such features.

Abstract syntax.

cat $C\ \Gamma$	C is a category depending on the context Γ
fun $f\ :\ A$	f is a function of type A
def $a\ =\ b$	a is defined as b
data $C\ =\ f_1 \mid \ldots \mid f_n$	C has the constructors $f_1, \ldots, f_n$

Concrete syntax.

param $P = C_1\,\Gamma_1 \mid \ldots \mid C_n\,\Gamma_n$	P is a parameter type with the constructors C_1 with context Γ_1, ..., C_n with context Γ_n
lincat $C\ =\ L$	C has the linearization type L
lindef $C\ =\ t$	C has the default linearization t
lin $f\ =\ t$	f has the linearization function t
oper $h\ :\ T\ =\ t$	h is an operation of type T, defined as t

Syntactic sugar: omitting keywords.

$$\textbf{key}\ J\ ;\ \ldots\ ;\ K \quad \equiv \quad \textbf{key}\ J\ ;\ \ldots\ ;\ \textbf{key}\ K$$

Fig. 1. Forms of judgement in GF.

with their usual Logical Framework meanings (in e.g. (Nordström *et al.*, 1990)).

6.2 Abstract syntax

6.2.1 Categories, types, and functions

Judgements of **cat** and **fun** forms are used for building basic types and objects. The **cat** judgement

$$\textbf{cat}\ C\ \Gamma$$

presupposes that Γ is a *context*, i.e. a sequence of variable declarations

$$(x_1\ :\ A_1)\cdots(x_n\ :\ A_n)$$

where $A_i\ :\ \mathit{Type}\ (x_1\ :\ A_1)\cdots(x_{i-1}\ :\ A_{i-1})$ for every $i = 1, \ldots, n$. The rule of *basic type formation* (Figure 2) tells how types are formed from a category by instantiating the context. If $n = 0$, the context is empty, and C is itself a type.

The **fun** judgement

$$\textbf{fun}\ f\ :\ A$$

presupposes that A is a type. It generates an object f of type A, to which the rules of application and abstraction apply in accordance with the type A, as well as the β and η conversion rules. These rules are shown in Figure 2. They are more or less the standard rules of logical frameworks with dependent types, such as (Nordström *et al.*, 1990).

Syntactic sugar in Figure 3 is of two opposite kinds: variable elimination and type factorization. As usual in dependently typed languages, variables can be eliminated from contexts and function types whenever there are no dependencies on them. The resulting notation is similar to simply typed languages, such as Haskell. On

Basic type formation.

$$\frac{\textbf{cat}\ C\ (x_1\ :\ A_1)\cdots(x_n\ :\ A_n)\quad a_1\ :\ A_1\quad \ldots\quad a_n\ :\ A_n(x_1 := a_1, \ldots, x_{n-1} := a_{n-1})}{C\,a_1 \ldots a_n\ :\ \mathit{Type}}$$

Basic object formation.

$$\frac{\textbf{fun}\ f\ :\ A}{f\ :\ A}$$

Function type formation, application, and abstraction.

$$\frac{A\ :\ \mathit{Type}\quad \begin{array}{c}(x\ :\ A)\\ B\ :\ \mathit{Type}\end{array}}{(x\ :\ A) \to B\ :\ \mathit{Type}} \qquad \frac{f\ :\ (x\ :\ A) \to B\quad a\ :\ A}{f\,a\ :\ B(x := a)} \qquad \frac{\begin{array}{c}(x\ :\ A)\\ b\ :\ B\end{array}}{\lambda x \to b\ :\ (x\ :\ A) \to B}$$

β and η conversion.

$$(\lambda x \to b)\,a = b(x := a) \qquad \frac{c\ :\ (x\ :\ A)\ \to\ B}{c\ =\ \lambda x \to (c\,x)}$$

Definition expansion.

$$f\,a_1\ \ldots\ a_n\ =\ t\gamma_1 \ldots \gamma_n$$

for the first **def** $f\,p_1\ \ldots\ p_n = t$ such that $p_1 <\gamma_1> a_1,\ \ldots,\ p_n <\gamma_n> a_n$

Fig. 2. Rules for types and objects in abstract syntax.

Variable elimination.

$(\_\ :\ A)\Gamma$	$\equiv$	$(x\ :\ A)\Gamma$	if Γ does not depend on x
$A\,\Gamma$	$\equiv$	$(\_\ :\ A)\Gamma$	
$(\_\ :\ A) \to B$	$\equiv$	$(x\ :\ A) \to B$	if B does not depend on x
$A \to B$	$\equiv$	$(\_\ :\ A) \to B$	
$\lambda\_ \to b$	$\equiv$	$\lambda x \to b$	if b does not depend on x

Factorization.

fun $f, \ldots, g\ :\ A$	$\equiv$	**fun** $f\ :\ A\,; \ldots\,; g\ :\ A$
$(x, \ldots, y\ :\ A)$	$\equiv$	$(x\ :\ A)\cdots(y\ :\ A)$
$(x, \ldots, y\ :\ A) \to B$	$\equiv$	$(x\ :\ A) \to \cdots \to (y\ :\ A) \to B$
$\lambda x, \ldots, y \to b$	$\equiv$	$\lambda x \to \cdots \to \lambda y \to b$

Fig. 3. Syntactic sugar for abstract syntax.

the other hand, the use of variables allows type factorizations that are not possible in Haskell. For instance, the following abbreviation is useful if A is complex:

$$(\_, \_, \_ : A) \to B \equiv A \to A \to A \to B.$$

6.2.2 Normal forms of abstract syntax types and objects

The type of any **fun** function f has the form

$$(x_1 \ : \ A_1) \to \cdots \to (x_n \ : \ A_n) \to A$$

where A is a basic type $C\, t_1 \ \ldots \ t_m$ where C is a category. With reference to this form, we say that $A_1, \ldots, A_n$ are the *argument types* of f, that A is its *value type*, and that C is its *value category*. The *full application* of f has the form

$$f\, a_1 \ \ldots \ a_n$$

whose type is $A(x_1 := a_1 \ \ldots \ x_n := a_n)$. A term of a function type is in $\beta\eta$ normal form, if it is an abstraction

$$\lambda z_1 \to \cdots \to \lambda z_n \to b$$

and b is an application of a constant or a variable or a metavariable, with all arguments in $\beta\eta$ normal form. We can use the β and η conversion rules to bring any well-typed term into this form.[18]

6.2.3 Metavariables

There is an infinite supply of metavariables

$$?_0, ?_1, ?_2, \ldots$$

which can be terms of any type. Metavariables are generated in parsing and in interactive editing, and they do not appear in GF grammars. Since they are generated directly in η-expanded form $\lambda x_1, \ \ldots \ , x_m \ \to ?_k$, the metavariable $?_k$ itself has always a basic type $C\, a_1 \ \ldots \ a_n$.

6.2.4 Definitions

A judgement of the form

$$\textbf{data}\ C \ = \ f_1 \ | \ \ldots \ | \ f_n$$

presupposes that C is a category and that $f_1, \ldots, f_n$ are **fun** functions, such that the value category of each f_i is C. What the judgement says is that $f_1, \ldots, f_n$ are *constructors* of the category C. Like in ALF (Magnusson & Nordström, 1994), constructors can be added incrementally, by new **data** judgements.

A judgement of the form

$$\textbf{def}\ f\, p_1 \ \ldots \ p_m \ = \ d$$

presupposes that f is a **fun** function but not a constructor, and that d is an object of type determined by the types of f and $p_1, \ldots, p_m$[19]. The arguments $p_1, \ldots, p_m$

[18] Thus a function f alone is a term in normal form only if its type is a basic type: in the general case, the normal form of the term f is $\lambda z_1 \to \cdots \to \lambda z_n \to f\, z_1 \ \ldots \ z_n$.

[19] Cf. the definition of pattern contexts in Figure 6, modified for dependent types.

are *patterns*, i.e. terms formed from variables, the wildcard _, and constructors. Those *def* judgements that have one and the same f form the *implicit definition* of that f. They determine how applications of f are *computed* by using *pattern matching*. Matching is performed in the order in which the equations appear in the grammar, and the patterns may overlap; the pattern-matching rules are the same as the ones for concrete syntax in Figure 6.

Functions f that are neither constructors nor defined implicitly are *primitive notions*. The lexical rules of GF make no distinction between constructors, defined functions, and primitive notions.

6.3 Concrete syntax types and expressions

6.3.1 Tokens and strings

The type *Str*, informally called "strings", is actually a type of lists of *tokens*, which are objects of type *Tok*. Tokens in normal form are quoted strings ("*foo*"). The *agglutination* $t + u$ of two tokens is also a token. Token lists are built from the empty list [] and from tokens by means of *concatenation* $++$ (Table 4).

We treat *Tok* and *Str* as abstract types, which can be instantiated by any types that support the aforementioned methods. A simple model is one in which tokens are strings and token lists are lists of strings. In this model, $+$ is string concatenation and $++$ is list concatenation. Expressions for tokens can also be used as expressions for singleton token lists, and are thus overloaded. [20]

6.3.2 Parameters and parameter types

A *parameter type* (*PType*) P is defined by a *parameter declaration*

$$\textbf{param}\ P = C_1\,\Gamma_1 \mid \ldots \mid C_n\,\Gamma_n$$

where each Γ_i is a *parameter context*, i.e. a sequence $P_1 \ldots P_m$ of parameter types. The parameter declaration introduces the *parameter constructors* $C_1, \ldots, C_n$, which can be used as functions from their parameter contexts to P.

The parameter declarations of a grammar may not be recursive, nor mutually recursive. As a consequence, every parameter type P is finite, and we can form the list of all *parameter values* of type P,

$$V_P\ =\ [1_P, 2_P, \ldots, n_p]$$

generated by a left-to-right enumeration of the parameter values of type P[21].

[20] Another model of tokens are word descriptions obtained from a morphological analyser, e.g. "*point*+Noun+Pl" instead of "*points*". This demands separate morphology passes before parsing and after linearization, but is an efficient way to implement parsing when the lexicon is large.

[21] This is similar to the derivation of `Enum` class instances in Haskell, but more powerful, since it applies not only to enumerated types but also to disjunctive and conjunctive types.

The types of tokens and strings and parameter types.

$$Tok,\ Str,\ PType\ :\ Type$$

Tokens, the empty string, tokens as strings, agglutination, and concatenation.

$$"foo"\ :\ Tok \qquad [\,]\ :\ Str \qquad \frac{t\ :\ Tok}{t\ :\ Str} \qquad \frac{t,u\ :\ Tok}{t+u\ :\ Tok} \qquad \frac{s,t\ :\ Str}{s\ ++t\ :\ Str}$$

Parameter types and constructors.

$$\frac{\textbf{param}\ P\ =\ \ldots}{P\ :\ PType} \qquad \frac{P\ :\ PType}{P\ :\ Type} \qquad \frac{\textbf{param}\ P\ =\ \ldots\ |\ C\ P_1\ \ldots\ P_n\ |\ \ldots}{C\ :\ P_1\ \to\ \ldots\ \to\ P_n\ \to\ P}$$

Record type formation.

$$\frac{T_1,\ldots,T_n\ :\ Type}{\{r_1 : T_1\ ;\ \ldots\ ;\ r_n : T_n\}\ :\ Type}$$

Record formation and projection.

$$\frac{t_1\ :\ T_1\quad\ldots\quad t_n\ :\ T_n}{\{r_1 = t_1\ ;\ \ldots\ ;\ r_n = t_n\}\ :\ \{r_1 : T_1\ ;\ \ldots\ ;\ r_n : T_n\}} \qquad \frac{c\ :\ \{\ldots\ ;\ r\ :\ T\ ;\ \ldots\}}{c.r\ :\ T}$$

Projection computation.

$$\{\ldots ; r\ =\ t ; \ldots\}.r = t$$

Table type formation.

$$\frac{P\ :\ PType\quad T\ :\ Type}{P\ \Rightarrow\ T\ :\ Type}$$

Table formation and selection.

$$\frac{t_1\ :\ T\ \Gamma_P p_1\quad\ldots\quad t_n\ :\ T\ \Gamma_P p_n\quad [p_1,\ldots,p_n \text{ exhaustive for } P]}{\textbf{table}\ \{p_1\ \Rightarrow\ t_1\ ;\ \ldots\ ;\ p_n\ \Rightarrow\ t_n\}\ :\ P\ \Rightarrow\ T} \qquad \frac{c\ :\ P\ \Rightarrow\ T\quad p\ :\ P}{c\,!\,p\ :\ T}$$

Selection computation.

$$\textbf{table}\ \{\ldots\ ;\ p\ \Rightarrow\ t ; \ldots\}\,!\,v = t\gamma \quad \text{for the first } p \text{ such that } p{<}\gamma{>}v$$

Local definition.

$$\frac{\begin{array}{c}(x\ :\ T)\\ t\ :\ T\qquad e\ :\ E\end{array}}{\textbf{let}\ \{x : T = t\}\ \textbf{in}\ \ e\ :\ E} \qquad \textbf{let}\ \{x : t = T\}\ \textbf{in}\ \ e = e(x := t)$$

Global definition.

$$\frac{\textbf{oper}\ h\ :\ T\ =\ t}{h\ :\ T} \qquad \frac{\textbf{oper}\ h\ :\ T\ =\ t}{h = t}$$

Fig. 4. Types and objects of concrete syntax

Concatenation of tokens.

$$[\text{"}foo\ \ldots\ bar\text{"}] \equiv \text{"}foo\text{"} +\!+ \ldots +\!+ \text{"}bar\text{"}$$

Factorization.

$$\begin{array}{lcl}
\{\ldots; r, \ldots, s : T; \ldots\} & \equiv & \{\ldots; r : T; \ldots; s : T; \ldots\} \\
\{\ldots; r, \ldots, s = t; \ldots\} & \equiv & \{\ldots; r = t; \ldots; s = t; \ldots\} \\
\textbf{let}\ \ \{x_1 : T_1 = t_1; \ldots; x_n : T_n = t_n\}\ \textbf{in}\ \ e & \equiv & \textbf{let}\ \ \{x_1 : T_1 = t_1\}\textbf{in} \\
 & & \ldots\textbf{let}\ \ \{x_n : T_n = t_n\}\ \textbf{in}\ \ e
\end{array}$$

Case expression.

$$\textbf{case}\ \ e\ \textbf{of}\ \{\cdots\} \equiv \textbf{table}\ \{\cdots\}\ !\ e$$

Fig. 5. Syntactic sugar for concrete syntax.

6.3.3 Record types and records

The rules for labelled records in Figure 4 are completely standard. Record labels have local scopes, and their name space is distinct from identifiers. For a record r to be of the type R, it is enough that every label of R is given a value of appropriate type in r. The order of fields does not matter, nor do superfluous fields in r. In notation, fields in record types and records can be factorized.

6.3.4 Table types, tables, and pattern matching

Tables of type $P \Rightarrow T$ are finite functions from P to T. The argument type P must be a parameter type. The normal form of a table is a complete enumeration of argument-value pairs:

$$\textbf{table}\ \{1_P \Rightarrow t_1\ ;\ \ldots\ ;\ n_P \Rightarrow t_n\}$$

For convenience, and to capture generalizations, the use of *patterns* is permitted as well.[22] There are three kinds of patterns, as shown in Figure 6. The matching relation

$$p < \gamma > v$$

"the pattern p matches the value v with the substitution γ", is used in Figure 6 to define the computation of *selections* from tables with patterns. To test whether a list of patterns is *exhaustive* for a given type P, we just test whether all values of type P are matched by them.

Patterns are matched from left to right, and they are allowed to overlap, like in Haskell. Nonlinear patterns are forbidden (i.e. patterns where a variable x occurs more than once). It is in virtue of this that pattern contexts and substitutions can be simply concatenated.

[22] Tables with patterns are syntactically similar to `fn` expressions in ML.

Patterns: wildcard, variable, and constructor.

$$\_ \qquad x \qquad C\,p_1 \ldots p_n$$

Pattern contexts.

$$\Gamma_P \_ = () \qquad \Gamma_P x = (x : P)$$

$$\Gamma_P (C\,p_1 \ldots p_n) = \Gamma_{A_1} p_1 \cdots \Gamma_{A_n} p_n \text{ if } C : A_1 \to \ldots \to A_n \to A$$

Pattern matching rules.

$$\_ <\!\!> v \qquad x <\!\!> (x := v) > v \qquad \frac{p_1 <\!\!> \gamma_1 > v_1 \quad \ldots \quad p_n <\!\!> \gamma_n > v_n}{C\,p_1 \ldots p_n <\!\!> \gamma_1 \ldots \gamma_n > C\,v_1 \ldots v_n}$$

Fig. 6. Patterns and pattern matching.

If the argument type P of a table t is known, the table can be expanded to eliminate patterns, by going through the list of all parameter values of P:

$$t = \textbf{table}\ \{1_P \Rightarrow t\,!\,1_P\ ;\ \ldots\ ;\ n_P \Rightarrow t\,!\,n_P\}$$

It is often handy to use **case** expressions as syntactic sugar for selections, as shown in Figure 5.

6.3.5 Functions, operation definitions, and local definitions

Concrete syntax uses the same rules for functions and function types as abstract syntax (Figure 2). Since functions are not part of canonical GF (Section 5.4), the $\beta\eta$ normal form plays no role in concrete syntax.

Functions are mostly introduced in *operation definitions*, judgements of the form

$$\textbf{oper}\ h\ :\ T\ =\ t$$

where T is any type (in the sense of concrete syntax) and $t\ :\ T$. *Local definitions* (**let** expressions) have a similar syntax, but they are local to expressions.

The **oper** definitions of a grammar may not be (mutually) recursive. A consequence of this is that a defined operation h can always be eliminated from a grammar by replacing it with its definition t; this procedure is known as *inlining*.

6.3.6 Type variables

GF does not have polymorphism. Explicit type variables and dependent types are used instead. The type of these variables is *Type*; there is for the time being no stratification. An example is the *flip* function (familiar from Haskell):

$$\textbf{oper}\ \mathit{flip}\ :\ (a, b, c : \mathit{Type}) \to (a \to b \to c) \to b \to a \to c = \lambda \_, \_, \_, f, x, y \to f\,y\,x$$

Another example is a local type definition, which is not possible in Haskell:

$$\textbf{let}\ \{S\ :\ \mathit{Type}\ =\ \{s\ :\ \mathit{Str}\}\}\ \textbf{in}\ S \to S \to S \to S$$

6.4 Concrete syntax for abstract syntax

A mathematical view of a GF grammar is that the abstract syntax defines a free algebra of syntax trees, and the concrete syntax defines a homomorphism from this algebra into a system of concrete-syntax objects. Mainly for the purpose of deriving parsers, we restrict concrete-syntax objects to certain special forms of records, captured by the notion of a linearization type.

6.4.1 Linearization types

A *linearization type* L is a record type usable as value type of linearization. It must have one or more fields whose types are Str-valued tables, the other fields having parameter types. More precisely,

- A string type is either Str or $P \Rightarrow S$ where S is a string type.
- If S is a string type, $\{s : S\}$ is a linearization type.
- If L is a linearization type, so is the type resulting from adding a field $r : T$ where T is a parameter type or a string type.

To simplify the generation of a parser, we require that all and only the string type valued fields are labelled $s_1, s_2, \ldots$, or s[23].

When giving a linearization rule to a function whose arguments have function types, we need to know what the linearization type of such a type is. The following clauses define this notion inductively for all types:

$$\begin{array}{lcl} (C\ a_1\ \ldots\ a_n)^\circ & = & L, \text{ if } \mathbf{lincat}\ C\ =\ L \\ ((x_1\ :\ A_1) \to (x_n\ :\ A_n) \to A)^\circ & = & \{v_1 : Str; \ldots ; v_n : Str\} ** A^\circ \end{array}$$

In the latter clause, we assume that the function type is in normal form, i.e. that A is a basic type. To form the linearization type of a function type, we thus add to the linearization type of the value type one field of type Str for each argument type. The idea is to introduce a field for each variable symbol. If $n = 1$, we follow the convention of adding $v : Str$ without a subscript. To avoid clashes with the system-generated labels $v, v_1, v_2, \ldots$, for bound variables, these labels are forbidden in user-defined record types.

6.4.2 Linearization type definitions and linearization rules

A linearization type definition for a category C is a judgement of the form

$$\mathbf{lincat}\ C\ =\ L$$

which presupposes that L is a linearization type. A notational convention allows us to omit the **lincat** judgement of a category C if the linearization type is $\{s : Str\}$.

A linearization rule for a function

$$\mathbf{fun}\ f\ :\ (x_1\ :\ A_1) \to (x_n\ :\ A_n) \to A$$

[23] An alternative would be to type-annotate the record fields.

Pattern variables in linearization rules.

$$\textbf{lin}\ f\ x_1\ \ldots\ x_m\ =\ t \equiv \textbf{lin}\ f\ =\ \lambda x_1, \ldots, x_m \to t$$

The **pattern** *rule format.*

$$\textbf{pattern}\ f\ x_1\ \ldots\ x_n = t_1\ ++\ \ldots\ ++\ t_m \equiv \textbf{lin}\ f\ x_1\ \ldots\ x_n = \{s\ =\ t'_1\ ++\ \ldots\ ++ t'_m\}$$

where $"foo"' = "foo"$ and $x'_i = x_i.s$.

Fig. 7. Syntactic sugar for linearization.

is a judgement of the form

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

which presupposes that

$$t\ :\ A_1^o \to \ldots \to A_n^o \to A^o$$

A concrete syntax is *complete* w.r.t. an abstract syntax, if it contains a **lincat** judgement for every **cat** judgement, and a **lin** judgement for every **fun** judgement.

The **pattern** notation for linearization rules (Section 2) can be used if the argument and value types of f all have the linearization type $\{s : Str\}$. A pattern element t_i is then either a token or one of the variables x_j (see Figure 7).

6.4.3 Default linearization

To linearize symbols not defined in the grammar (variables and metavariables), GF uses *default linearization*. It is a function that takes a string to an object of a linearization type. The default linearization of a category C is defined by the judgement

$$\textbf{lindef}\ C\ =\ t$$

which presupposes

$$t\ :\ Str \to C^o.$$

If the grammar does not contain a judgement of this form, a default default linearization is used: for a string t, it is a record where every field of a string type has uniformly the value t, and every parameter field of type P has as its value 1_P, i.e. the first value of type P.

7 Partial evaluation and parsing

While the linearization and typechecking algorithms follow straightforwardly from the semantics of GF, there are two other algorithms that are crucial for most practical applications of GF, and which are nontrivial: partial evaluation and parsing. Partial evaluation takes a GF grammar into a form by which linearization can be performed with the minimum of interpretational overhead. The same form is the basis of parsing, since it permits the derivation of a finite set of context-free rules.

7.1 Partial evaluation

Partial evaluation is evaluation at compile time, leaving evaluation at runtime less work to do (Jones *et al.*, 1993). A partial evaluator takes a program and computes it as far as it can. In general, the result is not fully evaluated, since some input of the program is unknown. Moreover, partial evaluation can be performed in different ways, e.g. optimizing either the time needed to run the runtime program or the space needed to store it.

What we do in GF is evaluate linearization rules into the canonical form of Section 5.4. This operation does not always optimize space: although it sometimes does reduce space, it may also do quite the contrary. However, the result is always a time-optimized runtime grammar.

Given a rule pair

$$\textbf{fun}\ f\ :\ (x_1 : A_1) \to \ldots \to (x_n : A_n) \to A\ ;\ \textbf{lin}\ f\ =\ t$$

we η-expand the term t with respect to a sequence of *argument variables*, that is, variables standing for arguments of the function f. We denote argument variables by pairs $<C, i>$, where C is the value category C of the argument type A_i, and i tells that it is the i'th argument. Showing the category will be useful in parser generation. The result is a linearization rule

$$\textbf{lin}\ f\ =\ \lambda <C_1, 1>, \ldots, <C_n, n> \ \to\ \{r_1\ =\ t_1\ ;\ \ldots\ ;\ r_m\ =\ t_m\}$$

where further η-expansions w.r.t. the linearization type of A have produced the record form. Each t_j $(j = 1, \ldots, m)$ is either an expression of a basic type (*Str* or a parameter type) or a fully expanded table

$$\textbf{table}\ \{1_P\ \Rightarrow\ u\ !\ 1_P\ ;\ \ldots\ ;\ n_P\ \Rightarrow\ u\ !\ n_P\}$$

where each term $u\ !\ k_P$ is evaluated further by inlining **oper** and **let** constants, η-expanding records and tables, and applying the evaluation rules for projections, selections, and function applications. Sometimes we also need transformations analogous to the elimination of maximal segments in proof theory (Prawitz, 1965). The most important such transformation is

$$(\textbf{table}\ \{p\ \Rightarrow\ f\ ;\ \ldots\ ;\ q\ \Rightarrow\ g\}\ !\ e)\ a\ \rhd\ \textbf{table}\ \{p\ \Rightarrow\ f\,a\ ;\ \ldots\ ;\ q\ \Rightarrow\ g\,a\}\ !\ e$$

pushing an application inside a table. The transformation is needed if the selection cannot be computed: such is the case if e depends on an argument variable. There are similar rules for projection and selection. The eliminability of function applications is analogous to the subformula property of intuitionistic propositional calculus: since the type of the linearization term consists solely of records, tables, strings, and parameters, no terms of function types need appear in it.

After partial evaluation, the only remaining unknown input in linearization rules are the argument variables. Because of compositionality, they can be treated as pointers to the linearizations of subtrees.

7.2 Parsing

In Sections 2.5 and 3.3, the parsing problem of simplified versions of GF were reduced to parsing in context-free grammars. We will now do the same to full GF.

We use the partially evaluated form of linearization functions, and consider the fields that are of string types. Each field is an n-place table ($n \geq 0$) that has finitely many possible values u of type *Str*. We call these values u the *productions* of the function f. We derive a finite set of context-free rules from every production: starting from a linearization rule

$$\textbf{lin}\ f\ =\ \lambda{<}C_1, 1{>}, \ldots, {<}C_n, n{>}\ \rightarrow\ r$$

consider an s_k-labelled (and thus *Str*-valued) field of r. Let u be a production coming from this field. The context-free rules generated from u have the form

$$f_p.\ C^k\ ::=\ c$$

where the value category C^k stands for the k'th part of C, and the right-hand side c is a sequence of context-free items. The sequence c is constructed from elements of the set u^*, which is defined inductively on the structure of the production u:

Terminal:	s^*	=	$\{s\}$
Nonterminal:	$(<C, i>.s_j)^*$	=	$\{C_i^j\}$
Binding:	$(<C, i>.v_j)^*$	=	$\{Var_i^j\}$
Concatenation:	$(a\ ++ b)^*$	=	$\{cc' \mid c \in a^*, c' \in b^*\}$
Selection:	$(a\ !\ p)^*$	=	a^*
Table:	$(\textbf{table}\ \{p_1\ \Rightarrow\ b_1; \ldots; p_k\ \Rightarrow\ b_k\})^*$	=	$\bigcup_{i=1}^{k} b_i^*$

Now, each sequence in the set u^* has the form

$$c_1 \ldots c_m$$

where each c_l is either a terminal s or a category symbol C_i^j indexed by the argument position i and the discontinuous-part number j. This is not yet a sequence of context-free items, because of the presence of the position numbers: these numbers are needed for profiles. However, the formation of context-free items

$$c_1' \ldots c_m'$$

is just simplification of nonterminals: each Var_j^j becomes Var, and each C_i^j becomes C^j. The profile p is constructed by collecting, from the subsequence of nonterminals in $c_1 \ldots c_m$, the list of positions for each argument place of f, in the same way as in Section 3.3.

Profiles p in rule labels f_p were earlier just an optimization removing the need to look up f in the grammar when postprocessing parse trees. When parametric variation is introduced, profiles become indispensable: the arguments of f may be placed to different positions when the tree is linearized under different parameters. Thus it would not be enough to look up f to restore the order of constituents.

When restoring syntax trees from parse trees, we do the same as in Sections 2.5

and 3.3. Discontinuous constituents bring nothing new to this, since they can be treated as reduplications. What typically happens in these reduplication instances is that different parts have metavariables in different places, so that no conflicts arise when the final result is formed by unifying the parts. For example, the German grammar of Section 5.3 generates the context-free grammar (with profiles that ignore bindings, which are empty)

$$\begin{aligned} Pred_{[[1],[2,3]]}.\ S &::= NP\ VP^1\ VP^2 \\ Pred_{[[2],[1,3]]}.\ S &::= VP^1\ NP\ VP^2 \\ Pred_{[[1],[2,3]]}.\ S &::= NP\ VP^2\ VP^3 \\ Compl_{[[1],[]]}.\ VP^1 &::= TV \\ Compl_{[[],[1]]}.\ VP^2 &::= NP \end{aligned}$$

The sentence *Johann liebt Maria* has the initial parse tree

$$Pred_{[[1],[2,3]]}\ Johann\ (Compl\ lieben\ ?)\ (Compl\ ?\ Maria)$$

which unifies to the final tree

$$Pred\ Johann\ (Compl\ lieben\ Maria)$$

Since the formation of context-free rules suppresses all parameters, the parser is over-tolerant. It could, for instance, recognize *they walks* as a valid English sentence. A strict parser is obtained by filtering away all those parse trees whose linearization does not match the input string. This arrangement of parsing has the disadvantage of being potentially inefficient: the number of rejectable parses can be exponential[24]. Its advantages are the simplicity of implementation and that it gives, as by-product, *grammar correction*: we can use tolerant parsing followed by linearization to correct *they walks* into *they walk*. Parsing via context-free grammars is known as *off-line parsing* in the context of unification grammars: the alternative is to perform unification at each construction step of the syntax tree.

7.3 The expressive power of GF

The expressive power of a grammar formalism is often characterized by its weak generative capacity—the class of sets of strings (in Chomsky hierarchy) it is capable of generating. Even though the focus in GF is on strong generative capacity (the trees it assigns to strings), its place in Chomsky hierarchy is a meaningful question.

In a trivial sense, GF is in the class 0 of unlimited languages, since we can define the universal language U of strings over any finite alphabet and encode any predicate P on U in the abstract syntax as a type of proofs. The rule pair

$$\mathbf{fun}\ f\ :\ (x\ :\ U) \to P\,x \to S\ ;\ \mathbf{lin}\ f\,x\,y\ =\ x$$

defines a string language S which is undecidable if P is. This construction is based

[24] Or even infinite, if cyclic rules are present.

on the suppression of an argument in linearization. However, since parsing of suppressed arguments is solved by introducing metavariables, it remains a meaningful question what happens if we consider GF without suppression. We have already seen that GF is more powerful than the class 2 of context-free languages (Section 2). GF is not just *mildly context-sensitive* like e.g. TAG (Joshi, 1985), since GF can define the double copy language $\{w\,e\,w\,e\,w \mid w \in \{a,b\}^*\}$. The precise location of GF-without-suppression is an open question.

8 The implementation of GF

Above we have described GF as a language of its own, independently of implementation. This description is partly an abstraction from earlier implementation work, partly a specification followed in later work. In this section, we will give an outline of the implementation and some problems that we have encountered in it.

8.1 Overview of the code

The Haskell implementation of GF (Version 1.0) has 12k lines of source code in 95 modules. The main parts of the code are the following:

- Grammar compiler: lexer, parser, type checker, partial evaluator, parser generator.
- Command line interpreter: functions to read grammar files and use grammars in batch mode.
- Syntax editor: functions to edit GF objects interactively.

The syntax editor is based on an abstract command language built upon a zipper, and it can be used through different user interfaces: we have a line-based editor, a graphical editor written in Fudgets (Carlsson & Hallgren, 1998), an experimental speech-based editor (Ranta & Cooper, 2001), and a Java GUI client communicating with a GF server via an XML-based protocol.

8.2 The use of Haskell

Haskell was chosen as implementation language for two reasons: we found it to be a good general-purpose programming language (particularly good for implementing compilers for functional languages), and we wanted to connect smoothly with some other programs written in Haskell, in particular, the proof editor Alfa (Hallgren, 2000). Some of the code was translated from earlier SML programs; in general, we did not want to exploit the laziness of Haskell in any essential way[25]. Neither did we use Haskell's impure features such as strictness flags. Monads (IO, error, state) are used heavily, and some classes are defined to simplify function names. GF conforms

[25] There is one single point where laziness would be useful: to treat infinite lists of parse trees arising in cyclic grammars (Section 2.5).

to the Haskell 98 standard (Peyton Jones & Hughes, 1999), and can be compiled with all standard compilers and interpreters, on all major operating systems[26].

8.3 Performance

There are two demanding components in GF: grammar compilation and object-language parsing. The parser of grammars was created using the Happy parser generator (Marlow, 2001), and it performs well. If the grammar is close to canonical form, type checking and partial evaluation together take less time than parsing[27]. However, if the grammar makes heavy use of functions and pattern matching, partial evaluation may take ten times longer than parsing. Some heuristics have helped considerably, such as topologically sorting all **oper** definitions and compiling them in dependency order, ignoring unused operations. Of course, once a grammar is ready, the compiled version can be saved in a file for rapid reuse.

The inefficiency of object-language parsing is partly due to the inherent complexity of general-purpose context-free parsing algorithms[28]. This can be helped in the special case of LALR(1) by using Happy parsers, which can be automatically generated from GF. More often, however, the bottleneck is postprocessing. To solve this problem, postprocessing would have to be integrated in the first parsing phase, using e.g. the semantic actions of Happy or some form of attribute or unification grammar. This is a research problem rather than an implementation issue.

8.4 Accessing GF

Users who do not write grammars themselves typically use GF via the graphical interactive editor. For grammar developers, and writers of batch programs, there is a command language and a shell, also permitting scripts. For instance, the following script imports an English and a French grammar, reads the file `enter.txt`, parses it as an English text, and linearizes the resulting tree in French:

```
i alarm.Eng.gf
i alarm.Fra.gf
rf enter.txt | p -lang=Eng | l -lang=Fra
```

Haskell programmers can access GF through an API (Application Programmer's Interface) module. It contains both default and customizable versions of parsing, linearization, and translation functions. This makes it possible to include GF functionalities and use GF grammars in other Haskell programs.

A library of macros is provided for creating GF grammars by Haskell programs. One way of using these macros is to define translations from other grammar formats

[26] Since the Fudgets library (Carlsson & Hallgren, 1998) requires the X window system, the Java GUI is the only graphical interface that works on Microsoft Windows.

[27] Parsing a 22k-line grammar with a Swedish resource lexicon takes 4 seconds on a 1.5 GHz Pentium 4 with RedHat Linux 7.1; the rest of the compilation of this close-to-canonical grammar takes 3 seconds.

[28] The time they take is cubic in the length of the input string.

to GF. For instance, BNF and EBNF can be used as input formats. Another use of code generation is to bypass the partial evaluator of GF: make all generalizations and abstractions in the Haskell code, and generate canonical GF directly. The next step from this idea would be to define GF as an embedded language (Hudak, 1996). However, we prefer to see GF as a language of its own, which can be used and reasoned about independently of implementation language. Moreover, since GF has dependent types, it is not possible to rely on code generated from Haskell: at least a type checker would in any case have to be written.

9 Some applications of GF

GF grammars have been written for fragments of at least 20 natural languages and many formal languages. Most of these grammars serve the theoretical purpose of verifying that GF can express a particularly intricate grammatical rule, or formalize the semantics of some specific application. The following list mentions some applications that have passed the level of first experiments and become independent projects.

Proof text editors. These are systems in which formal proofs are interactively constructed in type theory and at the same time viewed as texts in natural language. Via a parser, natural language input is also possible. The system is extensible to user-defined constants by means of user-defined linearization rules; if a rule is not given, a default linearization is generated. Users can also extend it to new new natural languages by writing GF grammars for the concrete syntax. Two implementations of proof text editors have been made in GF: one that works as a plug-in in the proof editor Alfa (Hallgren & Ranta, 2000), with support for English, French, and Swedish, and another one using the generic GF interface, also supporting Finnish, Italian, and Russian.

Software specifications. Formal and semi-formal software specification languages, such as OCL (Warmer & Kleppe, 1999), are widely used in industry, but still wider is the use of informal specifications in natural language. A project is going on to bridge this gap by building an abstract specification language in GF, with concrete syntaxes for OCL and English (Hähnle *et al.*, 2002). The goal is to enable simultaneous production of formal and informal specifications. The editor is being integrated in an industrial CASE tool.

Controlled language. This is the next step from mathematical proofs via software specifications towards non-mathematical language. Controlled languages are subsets of natural languages used for technical purposes such as instruction manuals for aircraft maintenance. Today's controlled languages (e.g. (The Boeing Company, 2001)) have neither formal grammars nor automatic checkers. But GF has been used to define prototypes where formal verification is applied to documents written in natural language. An example is a set of instructions for using an alarm system, generated in English, French, German, and Swedish, and equipped with a formal proof that the instructions preserve the system in a legal state (Johannisson & Ranta, 2001).

Dialogue systems. This is human-machine interaction where information is gathered by questions and answers. For instance, in a travel-agency dialogue system the machine asks where and when the customer wants to travel. The human answers all questions till enough information has been gathered to complete the booking. In order for the dialogue not to be too monotonous, the dialogue system should be flexible and e.g. accept answers to many questions at once. Several such criteria are identified in (Bohlin *et al.*, 1999). Somewhat surprisingly, it turned out that the metavariable-based model of interaction in proof editors readily fulfils most of these criteria, even adding extra functionality, e.g. a better control of the continuation of a dialogue via dependent types (Ranta & Cooper, 2001).

10 Related work

10.1 Montague grammar and categorial grammars

From the linguistic point of view, GF belongs to the tradition of *Montague grammar* (Montague, 1974). For Montague, a grammar was a set of rules linearizing logically interpreted analysis trees into strings of a natural language. The focus was on semantics rather than concrete syntax. A well-known problem in Montague's syntax is the use of so-called "quantifying in" rules to linearize variable-binding operations. Unlike other parts of Montague grammars, these rules cannot be directly formalized in GF, since they are not compositional. The rules can be circumscribed, however, partly by using combinators instead of variable binding (as suggested by Steedman (1988)), partly by means of discontinuous constituents.

The distinction between abstract and concrete syntax is seldom made by linguists. It was suggested, however, by the logician Haskell B. Curry, under the headings of *tectogrammatic* and *phenogrammatic* structure (Curry, 1963). For Curry, a tectogrammatic structure is similar to a term in combinatory logic, and it can show up as different phenogrammatic structures in different languages. Neither Curry nor Montague pursued the multilingual aspect, but there is a machine translation project, Rosetta (1994), based on Montague grammar.

Categorial grammar shares with Montague grammar the use of a type system to explain syntactic well-formedness. However, the idea is to explain not only abstract but also concrete syntax in terms of function application. To this end, Bar-Hillel (1953) made a distinction between prefix and postfix function types, β/α vs. $\alpha\backslash\beta$. His idea was developed further by Lambek (1958), resulting in a calculus that covered an impressive fragment of English, and was eventually proved equivalent to context-free grammars (Pentus, 1993). Extensions of Lambek calculus use richer sets of connectives (Morrill, 1994), or treat it as noncommutative linear logic (Abrusci, 1990).

In functional programming, some efforts have been made to implement logical and categorial grammars. A parser for a Montague-style grammar was implemented as a part of a database query system by Frost and Launchbury (1989), in Lazy ML. The grammar used in the system can also be defined in GF. A parser for the categorial grammar of Shaumyan was implemented by Jones and Hudak (1995) in

Haskell. This theory shares with Lambek calculus the use of typing rules to define concrete syntax.

10.2 Unification grammars

Unification grammars (Shieber, 1986) are a family of grammar formalisms where context-free categories are made dependent on *features*, which the parser tries to unify. Many grammar formalisms in computational linguistics belong to this family. Definite Clause Grammar (DCG) (Pereira & Warren, 1980) is perhaps the purest and simplest of them, and it has a built-in implementation in the Prolog programming language. It is also the most widely known, because it works well in education. The biggest grammars, however, have been written in Head Driven Phrase Structure Grammar (HPSG) (Pollard & Sag, 1994).

A typical example of DCG is the English predication rule,

$$S \longrightarrow NP(n)\ VP(n)$$

This rule expresses the condition that the subject and the verb must have the same number, n. In GF, the natural way to express the predication rule would be

$$\mathbf{fun}\ Pred\ :\ NP\ \rightarrow\ VP\ \rightarrow\ S\,;\ \mathbf{lin}\ Pred\,N\,V\ =\ \{s\ =\ N.s\ ++V.s\ !\ N.n\}$$

The traditional grammar view is closer to GF than to DCG: the subject and the verb are not in symmetric relation, but the verb depends on the subject. The subject *has* a number (as inherent feature), which it *gives* to the verb (as parameter).

The advantage of treating inherent features and parameters on a par is computational: it allows a direct implementation of parsing as unification. From the descriptive point of view, DCG appears as a low-level language, which moreover does not have types. A suggestive way of parsing in GF grammars would be to compile them into a DCG, and use local unification instead of off-line parsing and postprocessing.

HPSG inherits from PATR (Shieber, 1986) the use of records to express complex grammatical objects. In HPSG, these records contain both syntactic and semantic information. For instance, the English noun form *integers* could be described by the record (in GF notation)

$$\{cat = CN\,;\ sem = Int\,;\ phon = "integers"\ ;\ n = Pl\,;\ g = Neut\}$$

Records like this are called *signs* in HPSG. The information contained in a sign belongs partly to function declarations and partly to linearization rules in GF. We come close to a sign if we take a linearization record and add fields for the type and the syntax tree. However, the result is not quite the same: for *Int*, we get

$$\{cat = CN;\ sem = Int;\ s = \mathbf{table}\ \{Sg \Rightarrow "integer"; Pl \Rightarrow "integers"\}\,;\ g = Neut\}$$

The difference reflects the characteristic fact that HPSG records are obtained by analysing strings, whereas GF records are obtained by linearizing trees. The HPSG record is, in a sense, an *instance* of the GF record: it shows one branch of a table instead of the whole table.

Another interesting feature of HPSG is that it has a type system, which helps to detect errors at compile time. As regards records, the type system has much in common with GF. But there are no function types and thus no higher-order abstractions available for grammar writers.

10.3 Syntax editors

As a syntax editor, GF belongs to the tradition starting from Mentor (Donzeau-Gouge *et al.*, 1975) and the Cornell Program Synthesizer (Teitelbaum & Reps, 1981). These systems were initially not frameworks but had a hard-wired object language. Later on, the Cornell system used attribute grammars (Knuth, 1968) in the same rôle as the GF formalism is used in the GF editor, and Mentor was extended by the formalism framework Metal (Kahn *et al.*, 1983). As for concrete syntax, these systems of course only support unambiguous programming languages, one at a time. On the abstract level, they have advanced computational features, such as stepwise forward and backward execution of code.

Proof editors are a descendant of syntax editors, and the closest to GF are those that use dependent types. GF has inherited its type theory from ALF (Magnusson & Nordström, 1994), which uses metavariables, whereas NuPRL (Constable, 1986), Coq (The Coq Development Team, 1999), and LEGO (Luo & Pollack, 1992) use tactics. All these systems support some amount of user-defined syntactic sugar, such as infix declarations, but are of course far from natural language syntax. For Coq, a natural-language interface exists (Coscoy *et al.*, 1995). It works in the direction of linearization only and cannot be extended by the user; however, it has some built-in optimizations that are not possible in compositional linearization. Even more in this direction is the proof explanation system PRex (Fiedler, 2001), which uses AI methods to adapt proof texts for individual users.

WYSIWYM ("What you see is what you mean") is a multilingual authoring system for software manuals (Power & Scott, 1998). The user edits an abstract object which is reflected by "feedback texts" in English, French, and Italian. The grammars are hard-wired in the system and work in the direction of linearization only; the research emphasis is clearly on interaction rather than on grammars.

11 Conclusion

We have defined a grammar formalism GF on top of a logical framework with dependent types. The formalism is a special-purpose functional programming language, which adds the known advantages of functional languages (type checking, high abstraction level, succinctness of expression) to a simple computational model. GF grammars can be used for both parsing and generation of languages. The formalism is able to describe semantic conditions and intricate natural-language structures. It differs from earlier grammar formalisms by being based on functional programming and by having a powerful type system. The most important remaining problem is the inefficiency of the parsers generated from some GF grammars.

The main applications of GF are in domain-specific fragments of natural lan-

guage, which have a semantic model that can be described in type theory. GF grammars provide natural-language interfaces to such models and make it possible to translate domain-specific language reliably via the model. GF supports interaction: it can be used as a multilingual authoring system in which texts are created in many languages simultaneously. For future developments, an important task is to develop libraries of domain-independent resource grammars.

GF has been implemented in the functional language Haskell. The implementation follows the Haskell 98 standard and is portable to different operating systems. In addition to the separate program, GF functionalities can be accessed from other Haskell programs through an API module.

References

Abrusci, M. (1990). Noncommutative Intuitionistic Linear Propositional Logic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, **36**, 297–398.

Augustsson, L. (1998). Cayenne—a language with dependent types. *Proc. of ICFP'98.* ACM Press.

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

Bescherelle. (1997). *La conjugaison pour tous.* Hatier.

Bohlin, P., Bos, J., Larsson, S., Lewin, I., Matheson, C., & Milward, D. (1999). *Survey of existing interactive systems.* Trindi deliverable D1.3, Gothenburg University.

Bresnan, J. (ed). (1982). *The Mental Representation of Grammatical Relations.* MIT Press.

Carlsson, M., & Hallgren, T. (1998). *Fudgets—Purely Functional Processes with applications to Graphical User Interfaces.* Ph.D. thesis, Department of Computing Science, Chalmers University of Technology.

Constable, R. L. (1986). *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall.

Coquand, T. (1996). An algorithm for type checking dependent types. *Science of Computer Programming*, **26**, 167–177.

Coscoy, Y., Kahn, G., & Thery, L. (1995). Extracting text from proofs. *Pages 109–123 of:* Dezani-Ciancaglini, M., & Plotkin, G. (eds), *Proc. Second Int. Conf. on Typed Lambda Calculi and Applications.* LNCS, vol. 902.

Curry, H. B. (1963). Some logical aspects of grammatical structure. *Pages 56–68 of:* Jakobson, Roman (ed), *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics.* American Mathematical Society.

de Bruijn, N. G. (1994). Mathematical Vernacular: a Language for Mathematics with Typed Sets. *Pages 865–935 of:* Nederpelt, R. (ed), *Selected Papers on Automath.* North-Holland Publishing Company.

Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., & Levy, J. J. (1975). A structure-oriented program editor: a first step towards computer assisted programming. *International Computing Symposium (ICS'75).*

Dymetman, M. Lux, V. & Ranta, A.(2000). XML and multilingual document authoring: Convergent trends. *Pages 243–249 of: COLING, Saarbrücken, Germany.*

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, **13**(2), 94–102.

Fiedler, A. (2001). *User-Adaptive Proof Explanation.* Ph.D. thesis, Universität des Saarlandes.

Frost, R., & Launchbury, J. (1989). Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, **32**(2), 108–121.

Hallgren, T. (2000). *Home Page of the Proof Editor Alfa.* `http://www.cs.chalmers.se/~hallgren/Alfa/`

Hallgren, T. & Ranta, A. (2000). An extensible proof text editor. *Pages 70–84 of:* Parigot, M., & Voronkov, A. (eds), *LPAR-2000.* LNCS/LNAI, vol. 1955. Springer.

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

Hähnle, R. Johannisson, K. & Ranta, A. (2002). An authoring tool for informal and formal requirements specifications. *Pages 233–248 of:* Kutsche, R.-D., & Weber, H. (eds), *Fundamental Approaches to Software Engineering.* LNCS, vol. 2306. Springer.

Hopcroft, J., & Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys*, **28**(4).

Huet, G. (1997). The Zipper. *Journal of Functional Programming*, **7**(5), 549–554.

Huet, G. (2000). *Sanskrit site.* Program and documentation, `http://pauillac.inria.fr/~huet/SKT/`.

Johannisson, K., & Ranta, A. (2001). Formal verification of multilingual instructions. *The joint winter meeting of computing science and computer engineering.* Chalmers University of Technology.

Johnson, S. C. (1975). *Yacc — yet another compiler compiler.* Tech. rept. CSTR-32. AT & T Bell Laboratories, Murray Hill, NJ.

Jones, M., & Hudak, P. (1995). Using types to parse natural language. *Proceedings of the Glasgow Workshop on Functional Programming.* LNCS.

Jones, N.D., Gomard, C.K., & Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation.* Prentice-Hall.

Joshi, A. (1985). Tree-adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions. *Pages 206–250 of:* Dowty, D., Karttunen, L., & Zwicky, A. (eds), *Natural Language Parsing.* Cambridge University Press.

Kahn, G., Lang, B., Mélèse, B., & Morcos, E. (1983). Metal: a formalism to specify formalisms. *Science of Computer Programming*, **3**, 151–188.

Kay, M. (1997). The Proper Place of Men and Machines in Language Translation. *Machine Translation*, **12**(1–2), 3–23.

Knuth, D. (1965). On the translation of languages from left to right. *Information and Control*, **8**, 607–639.

Knuth, D. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, **2**, 127–145.

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

Luo, Z., & Callaghan, P. (1999). Mathematical vernacular and conceptual well-formedness in mathematical language. *Pages 231–250 of:* Lecomte, A., Lamarche, F., & Perrier, G. (eds), *Logical Aspects of Computational Linguistics (LACL).* LNCS/LNAI, vol. 1582.

Luo, Z., & Pollack, R. (1992). *LEGO Proof Development System.* Tech. rept. University of Edinburgh.

Magnusson, L., & Nordström, B. (1994). The ALF proof editor and its proof engine. *Pages 213–237 of: Types for Proofs and Programs.* LNCS 806. Springer.

Marlow, S. (2001). *Happy, The Parser Generator for Haskell.* `http://www.haskell.org/happy/`.

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

Mäenpää, P., & Ranta, A. (1999). The type theory and type checker of GF. *PLI-1999: Workshop on Logical Frameworks and Meta-languages, Paris, France.*

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

Morrill, G. (1994). *Type Logical Grammar.* Kluwer.

Necula, G. C. (1997). Proof-Carrying Code. *Pages 106–119 of: Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France.* ACM Press.

Nordström, B., Petersson, K., & Smith, J. M. (1990). *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press.

Paulson, L. (2002). *The Isabelle Reference Manual.* Available at the Isabelle homepage `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/` With contributions by T. Nipkow and M. Wenzel.

Pentus, M. (1993). Lambek grammars are context-free. *Pages 35–42 of: LICS, Utrecht, The Netherlands.*

Pereira, F., & Warren, D. (1980). Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, **13**, 231–278.

Peyton Jones, S., & Hughes, J. 1999 (February). *Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language.* Available from `http://www.haskell.org`

Pollard, C., & Sag, I. (1994). *Head-Driven Phrase Structure Grammar.* University of Chicago Press.

Power, R., & Scott, D. (1998). Multilingual authoring using feedback texts. *COLING-ACL.*

Prawitz, D. (1965). *Natural Deduction.* Stockholm: Almqvist & Wiksell.

Ranta, A. (1994). *Type Theoretical Grammar.* Oxford University Press.

Ranta, A. (2002). *Grammatical Framework Homepage.* `www.cs.chalmers.se/~aarne/GF/`.

Ranta, A., & Cooper, R. (2001). Dialogue systems as proof editors. *IJCAR/ICoS-3.*

Rosetta, M. T. (1994). *Compositional translation.* Dordrecht: Kluwer.

Shieber, S. (1986). *An Introduction to Unification-Based Approaches to Grammars.* University of Chicago Press.

Steedman, M. (1988). Combinators and grammars. *Pages 417–442 of:* Oehrle, R., Bach, E., & Wheeler, D. (eds), *Categorial Grammars and Natural Language Structures.* Dordrecht: D. Reidel.

Teitelbaum, T., & Reps, T. (1981). The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, **24**(9), 563–573.

The Boeing Company. (2001). *Boeing Simplified English Checker.* `http://www.boeing.com/assocproducts/sechecker/`

The Coq Development Team. (1999). *The Coq Proof Assistant Reference Manual.* `pauillac.inria.fr/coq/`.

Warmer, J., & Kleppe, A. (1999). *The Object Constraint Language: Precise Modelling with UML.* Addison-Wesley.