# The GF Grammar Compiler

Aarne Ranta

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

**Abstract.** GF (Grammatical Framework) is a grammar formalism based on the distinction between abstract and concrete syntax. An abstract syntax is a free algebra of trees, and a concrete syntax is a mapping from trees to nested records of strings and features. These mappings are naturally defined as functions in a functional programming language; the GF language provides the customary functional programming constructs such as algebraic data types, pattern matching, and higher-order functions, which enable productive grammar writing and linguistic generalizations. Given the seemingly transformational power of the GF language, its computational properties are not obvious. However, all grammars written in GF can be compiled into a simple and austere core language, Canonical GF (CGF). CGF is well suited for implementing parsing and generation with grammars, as well as for proving properties of GF. This paper gives a concise description of both the core and the source language, the algorithm used in compiling GF to CGF, and some back-end optimizations on CGF.

## 1 Introduction

Grammar formalisms are *formal systems* used for *defining languages.* As formal systems, they can be reasoned about, so that their mathematical properties such as worst-case parsing complexity can be determined. To make rigorous mathematical reasoning feasible, a grammar formalism should be *austere*, i.e. have as few constructs as possible.

At the same time, grammar formalisms are also *programming languages* used for *writing grammars.* For this task, austerity is no more a virtue. If we think of general-purpose programming languages, such as C or Haskell, we can easily establish them as Turing-complete—thus they are, in a way, grammar formalisms in class 0 of the Chomsky hierarchy. But no-one would like to write programs in a Turing-complete language with as few constructs as possible, such as pure lambda calculus or Böhm's P" [3].

Practical programming languages have constructs that are *redundant*, i.e. not strictly necessary for writing programs. Typical redundant features are high-level control structures that can squeeze several lines of code to just one, and rich type systems, which help the programmer to keep the code consistent. Redundancies are also involved in various *abstractions* that human programmers want to do but which will get lost in the austere machine representation.

In the research on grammar formalisms, the formal system aspects have been more prominent than the programming language aspects. One reason is certainly that many mathematical properties are still an open question—for instance, what complexity class is appropriate for a grammar formalism. Working on austere formalisms is essential to keep track of these aspects. However, when actually writing grammars, redundant constructs that help grammarians are welcome. What is more, linguists have always strived after abstractions and generalizations. To support this, those grammar formalisms that are actually used by linguists typically provide mechanisms such as the following:

– *Reducible language extensions* For example, EBNF extends BNF by regular expressions over BNF items, which can always be eliminated but make grammar writing more compact.
– *Macros.* For example, the finite-state scripting language XFST [2] can be made to look almost like a functional language by the use of carefully chosen macros.

Such facilities can raise the abstraction level of grammars without sacrificing their mathematical properties, and they are straightforward to implement: only a syntax-based preprocessor is needed, rather than a real compiler that has to analyse the code semantically. In general-purpose programming languages, the compiler has to do more work. The following constructs are found in many modern languages, but seldom in grammar formalisms:

– *Type systems.* Grammar formalisms usually operate in an untyped universe of strings or atoms in the sense of Prolog or LISP.
– *Functions.* Grammar formalisms usually rely on macros, a kind of “poor man’s functions”. Replacing macros by proper functions contributes to type safety, but also permits the powerful technique of higher-order functions.
– *Module systems.* Grammar formalisms usually rely on inclusions of files, rather than on separately compiled modules.

These constructs are characteristic of GF, Grammatical Framework [19]. GF a grammar formalism first and foremost designed as a programming language. It is modelled after the typed functional languages ML and Haskell (types, functions, pattern matching), but has also been inspired by C++ (overloading, multiple inheritance). GF aims to be easy for ordinarily trained programmers (non-linguists) to use, but at the same time to give linguists a tool by which they can enjoy of powerful abstractions. GF supports the development of grammars in collaborative projects, where resource grammars written by linguists are used as libraries in application grammars written by programmers. Since the first implementation in 1998, hundreds of programmers have used GF to create linguistic resources, user applications, and of course toy programs testing the limits of GF. Grammars have been written for fragments of at least 100 languages, and 12 languages have extensive resource grammars [18].

The linguists’ feel of GF is exemplified by a quote from Robin Cooper (personal communication): “using GF feels like having the power of a transformational grammar”. The key to this power is functional programming. A syntactic rule can use "transformations", i.e. functions that manipulate syntax trees, and even higher-order functions that take such functions as arguments. However, GF does not in the end have the transformational power: GF grammars reside in the class of polynomially parsable languages, as shown by Ljunglöf [10].

The "secret" of the controlled power of GF is the same thing that makes general-purpose programming languages work: compilation. "The GF grammar formalism" is actually two formalisms:

– Source GF, the rich language in which grammars are written.
– Canonical GF (CGF), the austere language to which grammars are compiled.

It would be hopeless for most humans to write grammars in CGF: it would feel like programming in machine code. At the same time, it would be hopeless to reason about source GF. There are too many language constructs to keep track of—a rough measure for this is that the BNF grammar used for parsing GF files has 247 productions. At the same time, the CGF format can be defined with less than 20 constructs.

The compiler that converts GF to CGF follows the compilation phases familiar from most modern compilers [1]:

1. Dependency analysis of the GF grammar to be compiled, determining what modules need compilation.
2. Lexing and parsing the GF code.
3. Type checking the parsed GF code with respect to the type system partly built within the code itself.
4. Simplifying the GF code so that it fits in a fragment of GF corresponding to CGF.
5. Generating the CGF code from the GF code.
6. Linking the grammar modules together into run-time grammar objects.
7. Optimizing the run-time grammar objects.

In this paper, we will first present the CGF formalism (Section 2) and give and outline of source GF (Section 3). The type checking phase is briefly discussed in Section 4, the simplifying phase in Section 5, the effect of simplification in grammar specialization in Section 6, generation of CGF in Section 7, and CGF optimizations in Section 8. A full presentation of source GF and its type system is given in [19]. The module system is introduced in [20].

## 2 Canonical GF

GF follows an architecture that divides a grammar into an *abstract syntax* and a *concrete syntax*. This division is commonplace in computer science, as a way of organizing compilers of programming languages. In linguistics, the same distinction was suggested by Curry [6] under the headings of *tectogrammatical* and *phenogrammatical* structure, respectively. It was implicitly followed by Montague [11] as observed by Dowty [9], but it has gained popularity only since the 1990's [17, 12, 19, 7, 15, 13].

A *run-time grammar* in GF is a grammar used for parsing and generation, as well as for type checking abstract syntax trees. It consists of one abstract syntax and one or more concrete syntaxes. The different concrete syntaxes typically model different languages, and the abstract syntax defines a common structure of those languages. GF grammars are thus *multilingual*.

To give a simplest possible example of a multilingual grammar, we first show a CGF grammar for constructing abstract greetings and linearizing them into two languages, English (*hello world*) and Italian (*ciao mondo*):

```
abstract Hello
  cat Greeting ; Addressee ;
  fun Hello : Addressee -> Greeting ;
  fun World : Addressee ;
concrete HelloEng
  lin Hello = [("hello",($0!0))] ;
  lin World = [("world")] ;
concrete HelloIta
  lin Hello = [("ciao",($0!0))] ;
  lin World = [("mondo")] ;
```

The abstract syntax has `cat` judgements defining the *categories* of the grammar, and `fun` judgementes definining the *functions* that construct *trees* of the categories. The functions can take zero or more arguments. For instance, `(Hello World)` is a tree formed by applying the one-argument function `Hello` to the zero-argument function `World`.

Each concrete syntax has `lin` judgements defining the *linearizations* of all functions. The linearization of a function with arguments may contain pointers to linearizations of the corresponding subtrees, denoted `$0, $1, $2`, etc. Linearization is thus a *compositional* operation, since it builds the linearization of a complex tree from the linearizations of its immediate subtrees.

The domain of linearization is a free algebra of syntax trees, inductively defined by an abstract syntax. The range of linearization is a universe whose elements are tokens, token lists, integers, and tuples. More formally, the universe $T$ of tuples is built as follows:

– Tokens: `"foo"` : $T$.
– Token lists: `(`$t_1, \ldots, t_n$ `)` : $T$ for $t_1, \ldots, t_n$ : $T$, $n \geq 0$.
– Integers: `0,1,2,...` : $T$.
– Tuples: `[` $t_1, \ldots, t_n$ `]` : $T$ for $t_1, \ldots, t_n$ : $T$, $n \geq 0$.

Obviously, the universe $T$ could be given a more discriminating type system, which would guarantee properties such as the impossibility to include integers in token lists. However, such restrictions are not needed in CGF, as long as we have a static type system for the GF source language and can thereby guarantee that the generated CGF expressions are meaningful.

Expressions for objects of the universe $T$ also include *variables*, i.e. pointers to subtree linearizations, and *projections*, which select numbered components from tuples:

– Variables: `$0, $1, $2,...` : $T$.
– Projections: `(` $t$ `!` $u$ `)` : $T$ if $t$ : $T$ and $u$ : $T$.

Notice that projections are only needed because of the presence of variables. The values of variables are not known at compile time; but as soon as they get known, a well-formed projection can be brought into a form from which it can be eliminated:

$$(\texttt{[}\ t_1, \ldots, t_n\ \texttt{]}\ \texttt{!}\ i\ ) \Longrightarrow t_i$$

In fact, this is the only computation rule needed in CGF, together with the rule of replacing variables with subtree linearizations. For this replacement, we maintain an array of these linearizations. The variable replacement rule is

$$\$i\{t_0, \ldots, t_n\} \implies t_i$$

The top-level linearization $t^*$ is defined compositionally as follows:

$$f\, t_1 \ldots t_n \implies f^*\, \{t_1^*, \ldots, t_n^*\}$$

where $f^*$ is the term defining the linearization of $f$ in the grammar.

The tuple structure is exploited to express linguistic combinations that involve more than just string concatenation. Actually, tuples are just an austere representation of *feature structures*, in which all atomic features are encoded as integers. Tuples can moreover contain more than one string component, which gives a model of *discontinuous constituents*. The computational advantage of the integer representation of features is that tuples can be implemented as *arrays*, which can be stored compactly and have efficient lookup.

Here is an example showing how to deal with number agreement in English. A verb phrase (`VP`) is a tuple that contains both a singular and a plural form. A noun phrase (`NP`) is a tuple that contains both a string and a parameter, the latter indicating the grammatical number.

```
abstract Predic
  cat S ; NP ; VP ;
  fun Pred : NP -> VP -> S ;
  fun He : NP ;
  fun They : NP ;
  fun Talk : VP ;
  fun Walk : VP ;
concrete PredicEng of Predic
  lin Pred = [(($0!0),(($1!0)!($0!1)))] ;
  lin He = ["he",0] ;
  lin They = ["they",1] ;
  lin Talk = [["talks","talk"]] ;
  lin Walk = [["walks","walk"]] ;
```

The linearization of the tree `(Pred They Walk)` is computed as follows:

```
    [(($0!0),(($1!0)!($0!1)))] {["they",1], [["walks","walk"]]}
  = [((["they",1]!0),(([["walks","walk"]]!0)!(["they",1]!1)))]
  = [("they",(["walks","walk"]!1))]
  = [("they","walk")]
```

Again, it is crucial for the computation that the values held by the variables are of certain types. This is guaranteed when CGF is generated from GF source, where every category in the abstract syntax is assigned a *linearization type* in the concrete syntax. Here, for instance, it is assumed that every `NP` is a tuple holding a string and a parameter in range {`0,1`}, and that every `VP` is a tuple holding a tuple with two strings.

To reach the expressive power of GF, in the language-theoretic sense, nothing more is needed than CGF as presented above. For this formalism, we can define a linearization algorithm that is linear in the size of the tree, and a parsing algorithm that is polynomial in the size of the string. The parsing algorithm is obtained via a reduction to PMCFG [22], which indeed is another grammar formalism based on tuples. (This result, in [10], does not hold in general if dependent types appear in the abstract syntas; using them collapses GF to level-0 formalisms.)

In addition to reasoning, CGF is a good format for implementation. Interpreters for CGF have been built in C++, Haskell, Java, and Prolog [18], and make it possible to embed grammar components in those languages. An even more intimate embedding is achieved when CGF grammars are translated into a programming language; such translators have been written for C and JavaScript [18]. Moreover, CGF grammars can be translated (via context-free or finite-state approximations) to various formats used for defining language models in speech recognizers [4].

## 3 Source GF

Writing grammars in CGF manually is neither safe nor productive. The reasons are similar to the reasons why machine code is not a good medium for general-purpose programming:

- The lack of type checking makes it difficult to avoid errors when writing grammars.
- The selection of elements from tuples by position is error-prone.
- The use of integers to represent grammatical features charges human memory and makes it possible to confuse features of different types.
- Writing explicit tuples without any abstractions makes the grammars bulky.

The GF source formalism solves these problems by providing a higher-level language. A strict, static type system makes the language failure-safe, and abstraction mechanisms make grammars concise. The range of concrete syntax mappings is changed from the austere all-encompassing universe of tuples to systems of *records* and *tables*, whose layout can be defined by the programmer and is thus not given once and for all.

In the GF source language—from now on, just GF—linearizations are built from the following ingredients:

– Tokens: `"foo" : Str` (also used as one-element token lists).
– Empty token list: `[]`
– Concatenation of token lists: `s ++ t : Str` if `s,t : Str`.
– User-defined parameter types: `param Number = Sg | Pl`.
– Labelled records: $\{\ r_1 = t_1; \ldots; r_n = t_n\ \} : \{\ r_1 : T_1; \ldots; r_n : T_n\ \}$.
– Finite functions, i.e. tables: `table {Sg => "walks" ; Pl => "walk"} : Number => Str`.

Each category in an abstract syntax is given a linearization type (`lincat`) in the concrete syntas, and all linearization judgements are type-checked with respect to these types. Here is the second example from the previous section rewritten in GF.

```
abstract Predic = {
  cat S ; NP ; VP ;
  fun Pred : NP -> VP -> S ;
  fun He, They : NP ;
  fun NP ;
  fun Talk, Walk : VP ;
}
concrete PredicEng of Predic = {
  param Number = Sg | Pl ;
  lincat S  = {s : Str} ;
  lincat NP = {s : Str ; n : Number} ;
  lincat VP = {s : Number => Str} ;
  oper regVP : Str -> VP = \v -> {
    s = table {Sg => v + "s" ; _ => v}
    } ;
  lin Pred np vp = {s = np.s ++ vp.s ! np.n} ;
  lin He   = {s = "he" ; n = Sg} ;
  lin They = {s = "they" ; n = Pl} ;
  lin Talk = regVP "talk" ;
  lin Walk = regVP "walk" ;
}
```

The presence of variables means, as in CGF, that more expression forms are needed:

– Projections from records: $t.r$ returns the field labelled $r$ from the record $t$.
– Selections from tables: $t$ *!* $v$ returns the value assigned to $v$ in the table `t`.
– Gluing of tokens: `"walk" + "s"` is computed to `"walks"`.

Note that the `case` expressions familiar from functional programming languages can be defined as syntactic sugar for table selections:

$$\texttt{case}\, t \,\texttt{of}\, \{\ldots\} \equiv \texttt{table}\, \{\ldots\}\, !\, t$$

This form of expression is convenient for programmers, and will also be used for presentation purposes below.

An important way in which GF is stronger than CGF is that variables are no longer only used for the arguments of linearization rules, but can appear anywhere in the concrete syntax:

- Auxiliary functions (`oper`) whose definitions bind variables.
- Local anonymous functions (lambda abstracts).
- Local definitions (`let` expressions).
- Pattern variables in table expressions used for case analysis.

It is the task of compilation from GF to CGF is to eliminate these "superfluous" variables. This can be done because their values are known at compile time. A technique based on *partial evaluation* is used. The usual evaluation rules of type theory are in this process applied to terms containing run-time variables.

Because of the way parsing algorithms work, the set of tokens must be known at compile-time (although it need not be finite). A token expression may thus not depend on run-time variables, which means that all instances of gluing tokens (`s + t`) are eliminated at compile time.

Let us conclude the presentation of GF with an example of parameter types and pattern matching. Parameter type definitions in GF are like algebraic datatype definitions in ML and Haskell, except that recursion is not allowed. They are thus not just enumerations of atomic features, but may introduce constructors that take arguments. An example is given by the following system, defining French agreement features as combinations of gender, number, and person:

```
param Gender = Masc | Fem ;
param Number = Sg | Pl ;
param Person = P1 | P2 | P3 ;
param Agr    = Ag Gender Number Person ;
```

The linearization of verb phrases depends on a parameter of type `Agr`. In actual pattern matching, separate branches are not necessarily needed for all values, but pattern variables can be passed to the right-hand side. This is what happens when verb phrases are formed from adjectival phrases by using the copula.

```
cat VP ; AP ;
lincat VP = {s : Agr => Str} ;
lincat AP = {s : Gender => Number => Str} ;
fun CompAP : AP -> VP ;
lin CompAP ap =
  {s = table {Ag g n p => copula n p ++ ap.s ! g ! n}} ;
oper copula : Number -> Person -> Str = ...
```

## 4 Type checking of GF expressions

The role of type checking in compilers is not just to verify the consistency of the code and report on errors. It also informs later compilation phases by adding *type annotations* to expressions. The most important annotations in GF are the following:

- Annotate tables with their argument types, to enable eta expansion.
- Annotate projections with integers indicating the positions of the projected fields.

The information needed for these operations is available at type checking time and would require the duplication of much of type checking work if performed later.

A recent addition to the type system is *overloading* of functions, which has proved useful in the presentation of large grammar libraries. In type checking, overloaded functions are replaced by their instances determined by the *overloading resolution algorithm.* The way overloading works in GF is inspired by C++ [23]. In comparison with C++, the possibility of partial applications makes the problem more complex, whereas the absence of type casts simplifies the problem.

## 5 Simplification of GF expressions

Simplification is performed as partial evaluation, which has several ingredients: eta expansion (5.1); application of evaluation rules (5.2); generalized reductions needed to guarantee the subformula property (5.3); elimination of variables from complex parameter expressions (5.4).

### 5.1 Eta expansion

Eta expansion is step that converts a term to the data form required by the type of the term. For function types, this form is that of a lambda abstract. For record types, it is a record with explicit fields appearing in the type. For table types, it is a table that has explicitly enumerated cases for each value of the argument type. Thus the expansion rules are as follows:

- $t \,:\, A \rightarrow B \implies \backslash x \,\rightarrow\, (t\,x)$
- $t \,:\, \{\ r_1 : T_1; \ldots ; r_n : T_n\ \} \implies \{\ r_1 = t.r_1; \ldots ; r_n = t.r_n\ \}$
- $t \,:\, P \Rightarrow T \implies \texttt{table}\ \{\ V_0 \Rightarrow t\,!\,V_0; \ldots ; V_n \Rightarrow t\,!\,V_n\ \}$

Eta-expanded forms correspond directly to CGF expressions of the corresponding types.

### 5.2 Evaluation rules

The evaluation rules of GF are completely standard: beta conversion of lambda abstracts, projection from records, selection from tables by pattern matching, and concatenation of tokens.

- $(\backslash x \,\rightarrow\, b)\,a \implies b\{x := a\}$
- $\{\ldots ; r = t; \ldots\}.r \implies t$

– $\texttt{table}\,\{\ldots;p \Rightarrow t;\ldots\}\,!\,u \implies t\gamma$ for the first $p$ that matches $u$ with $\gamma$
– `"foo" + "bar"` $\implies$ `"foobar"`

Pattern matching is similar to ML and Haskell, scanning the patterns from left to right. Its result is a term $t\gamma$ where $\gamma$ is a *substitution* by which the pattern $p$ matches the value $u$. The matched term $u$ may not in general contain variables. If variables occur in $u$, the selection must be postponed until the variables receive values; for run-time variables this means that the selection is passed to the generated CGF code.

### 5.3 The subformula property

Eta-expanded records and tables can be easily converted to CGF, but functions can be converted only if they appear as top-level linearization terms; no terms of a function type may occur inside those terms.

Now, a linearization term $t$ in

$$\texttt{fun}\,f\;:\;A_1 \to \cdots \to A_n \to A\;;\;\texttt{lin}f\;=\;t$$

is a function from the linearization types of $A_1,\ldots,A_n$ to the linearization type of $A$. These types are built from strings, features, records, and tables: no functions appear in these types. By a version of Curry-Howard isomorphism, the term $t$ can be seen as a *proof* of $A$ depending on the hypotheses $A_1,\ldots,A_n$. In this isomorphism, record types correspond to conjunctions and parameter types to disjunctions. No function types occur in the hypotheses or the conclusion. That no terms of a function type need occur in $t$ is an instance of the *subformula property* of intuitionistic propositional calculus in proof theory.

To prove the subformula property, it is not enough to use the ordinary evaluation rules. The problem is that the elimination of a function can be blocked by a variable. A case in point is a term of the form

$$(\texttt{case}\,x\,\texttt{of}\{A \Rightarrow f\;;\;B \Rightarrow g\})\,c$$

where the function application cannot be performed because of $x$. This term is isomorphic to a proof in which disjunction elimination is followed by a modus ponens. For this constellation, Prawitz [16] introduced a generalized reduction rule:

$$\frac{\dfrac{A \vee B \quad \overset{(A)}{C \to D} \quad \overset{(B)}{C \to D}}{C \to D} \qquad C}{D} \quad\Longrightarrow\quad \frac{A \vee B \quad \dfrac{\overset{(A)}{C \to D} \quad C}{D} \quad \dfrac{\overset{(B)}{C \to D} \quad C}{D}}{D}$$

Now the modus ponens has moved closer to the introductions of the implication, and can, as Prawitz proved, eventually be reduced away. The corresponding term transformation in the GF compiler is

$$(\texttt{case}\,x\,\texttt{of}\{A \Rightarrow f\;;\;B \Rightarrow g\})\,c \implies \texttt{case}\,x\,\texttt{of}\{A \Rightarrow f\,c\,;\;B \Rightarrow g\,c\}$$

### 5.4 Parameter constructors with variables in arguments

Parameter type definitions can introduce constructors that takes arguments, e.g. the constructor type `Ag` of French agreement features in Section 3. Now, an argument of `Ag` can be a run-time variable, as in

```
Ag Fem np.n P3
```

This expression has no translation in CGF. But we can first translate it to a case expression,

```
case np.n of {
  Sg => Ag Fem Sg P3 ;
  Pl => Ag Fem Pl P3
  }
```

which can then be translated to CGF by applying the compilation schemes of Section 7. This transformation has of course has be performed recursively, since there can be several occurrences of run-time variables in a constructor application.

## 6 Compilation and grammar specialization

A typical GF application is built on top of a *domain grammar*, whose abstract syntax is a type-theoretical model of a domain semantics. In the beginning, all GF grammars were such domain grammars implemented by writing concrete syntaxes from scratch. But gradually the evolution of the GF compiler permitted writing large-scale *resource grammars* and using them as libraries. The use of resource grammars in writing domain grammars is also known as *grammar specialization.*

As resource grammars are large and complex, they have high demands on both time and space. Properly implemented grammar specialization should eliminate all run-time penalty potentially caused by resource grammars. To show how this happens in GF, let us trace through the compilation of a simple domain grammar rule implemented using the GF resource grammar library [18]. The purpose of the application is to cover voice commands such as *I want to hear this song.* This is an example of how GF was used in the TALK project for building dialogue systems [14].

The abstract syntax covering the voice command is

```
cat Command ; Kind ;
fun want_hear_this : Kind -> Command ;
fun Song, Record, Singer : Kind ;
```

The concrete syntax assigns resource grammar categories as linearization types to the domain categories. Thus commands are utterances (`Utt`), kinds are common nouns (`CN`):

```
lincat Command = Utt ; Kind = CN ;
```

The linearization of the function `want_hear_this` is built by using constructor functions from the resource API. These constructors are either syntactic, having the form `mk`*C* for a function whose value category is *C*, or lexical, having the form *word*`_C` for a lexical unit of category *C*.

```
lin want_hear_this x = mkUtt (mkCl i_Pron
  (mkVP want_VV (mkVP hear_V2 (mkNP this_Quant x))))
```

The syntactic constructors are overloaded, for instance `mkVP` is here used for both `V2` (NP-complement verbs) and `VV` (VP-complement verbs).

In the resource grammar, a clause like our example has forms for different tenses and polarities and, e.g. in the case of German, also for different word orders. So the German variation includes 48 sentence forms, in the range

```
Pres Simul Pos Main: ich will dieses x hören
Pres Simul Pos Inv:  will ich dieses x hören
...
Cond Anter Neg Sub:  ich dieses x nicht würde hören wollen haben
```

Moreover, the form of `dieses` ("this") varies according to the gender of `x`: *diesen Sänger, dieses Lied, diese Platte.* All this variation gives initially 3*48 = 144 forms in the expanded tables. However, the linearization type `Utt` of the top-level category `Command` is a plain string, with no variation. The constructor `mkUtt` without explicit tense and polarity uses the values present, positive, and main clause. The category `Kind` is linearized to `CN`, which has a 3-valued gender parameter, so that the variation *diesen, dieses, diese* cannot be eliminated. So we have 3 forms that remain in the generated CGF grammar:

```
ich will diesen x hören
ich will dieses x hören
ich will diese  x hören
```

Further optimizations on the CGF code compactify this grammar by e.g. removing the repetitions of word strings (Section 8).

## 7 Translating GF to CGF

Once the GF grammar has been type-annotated and partial-evaluated, translation to CGF can be performed by compositional compilation schemes:

- $\texttt{lin}\, f\, \$0 ... \$n \, = \, t \implies \texttt{lin}\, f \, = \, t$
- $\{r_1 = t_1; \ldots ; r_n = t_n\} \implies [t_1, \ldots , t_n]$
- $\texttt{table}\{V_1 \Rightarrow t_1; \ldots ; V_n \Rightarrow t_n\} \implies [t_1, \ldots , t_n]$
- $s \mathbin{++} t \implies (s, t)$
- $[] \implies (s, t)$
- $t.r[i] \implies (t \,!\, i)$

– $t!u \implies (t!u)$
– $C\, v_1 \ldots v_n \implies \#(C\, v_1 \ldots v_n)$

The expression $t.r[i]$ is a projection annotated by the position $i$ of the label $r$. The expression $\#(v)$ denotes the integer value of the parameter value $v$.

The other forms of judgement—`param` and `lincat`—are not needed in run-time grammars and are hence omitted. However, when library grammars are separately compiled into CGF, these judgements are needed for type checking modules that use them.

Abstract syntax must be present in CGF, but it needs not be translated, apart from eliminating some syntactic sugar provided by the GF source language [19].

## 8 Back-end optimizations on CGF

Even though partial evaluation eliminates unnecessary rules from grammars (Section 6), the code bloat resulting from eta expansion can be significant. While eta expansion is indispensable for the compilation to work, some of its drawbacks can be relieved by back-end optimizations on CGF. This section gives a summary of two such optimizations, each of which uses a new expression form and thereby makes CGF less austere.

### 8.1 Common subexpression elimination

One and the same CGF term can appear several times in a grammar. Such terms can be captured by a standard technique of *common subexpression elimination*, which replaces the subterms with new constants and adds the definitions of those constants into the grammar. These expressions can contain run-time variables, and they are computed by simple syntactic replacement. The computation can be performed off-line in the whole grammar, but usually it is better to keep them in the run-time grammar and look them up at need.

Automatic subexpression elimination is often more powerful than hand-devised code sharing in the source code, because it catches all subexpressions that are used at least twice in the code. It is moreover iterated so that it takes into account the subexpressions in the definitions of global constants. The shrinkage of code size is typically by an order of magnitude.

### 8.2 Prefix-suffix tables

Suppose we have in a grammar the rule

```
Walk = [["walk", "walks", "walked", "walking"]]
```

The *prefix-suffix table* representation divides an array of words to the longest common prefix and an array of suffixes.

```
Walk = [[("walk" + ["","s", "ed", "ing"])]]
```

The power of this representation comes from the fact that suffix arrays tend to be repeated in a language, and can therefore be collected by common subexpression elimination. After this, a grammar may look as follows:

```
__a18 = ["","s", "ed", "ing"]
Destroy = [[("destroy" + __a18)]]
Talk = [[("talk" + __a18)]]
Walk = [[("walk" + __a18)]]
```

Thus this optimization in fact identifies a set of concatenative inflection paradigms of the language.

## 9 Implementation

The GF grammar compiler is a central component of the GF grammar development system, i.e. the program called `gf` and available from [18] as open-source software. The austere CGF format described in this paper is a second-generation target language for GF. At the moment, the GF system still uses a richer target language called GFC. GFC is a compromise between run-time simplicity and the concerns of separate compilation. The current GF system does produce CGF as a back-end format (under the name GFCC), but separate compilation and parser generation still depend on GFC.

## 10 Related work

Static type checking and library specialization are not very common in grammar formalisms. HPSG has type checking of typed feature structures [5], and Regulus [21] uses a technique called *explanation-based learning* for specializing large resource grammars to domain-specific run-time grammars. Regulus moreover compiles high-level unification grammars into lower-level context-free grammars by expanding rules depending on finite feature sets into sets of context-free rules.

In the new wave of grammar formalisms inspired by Curry [6], ACG has an implementation where generation is treated as term rewriting and parsing as higher-order linear matching [7]. These techniques are direct and elegant applications of the metatheory of ACG, but they do not give narrow complexity bounds. However, the equivalence results for fragments of ACG and classes such as context-free and mildly context-sensitive grammars [8] could serve as basis of efficient implementations via compilation.

## References

1. A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools. Second Edition.* Addison-Wesley, 2006.
2. K. Beesley and L. Karttunen. *Finite State Morphology.* CSLI Publications, 2003.

3. C. Böhm. On a family of Turing machines and the related programming language. *ICC Bulletine*, 3:185–194, 1964.
4. B. Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*, 2007.
5. A. Copestake and D. Flickinger. An open-source grammar development environment and broad-coverage English grammar using HPSG. *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000)*, 2000.
6. H. B. Curry. Some logical aspects of grammatical structure. In Roman Jakobson, editor, *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society, 1963.
7. Ph. de Groote. Towards Abstract Categorial Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Toulouse, France*, pages 148–155, 2001.
8. Ph. de Groote. Tree-Adjoining Grammars as Abstract Categorial Grammars. In *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 145–150. Università di Venezia, 2002.
9. D Dowty. *Word Meaning and Montague Grammar.* D. Reidel, Dordrecht, 1979.
10. P. Ljunglöf. *The Expressivity and Complexity of Grammatical Framework.* PhD thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University, 2004.
11. R. Montague. *Formal Philosophy.* Yale University Press, New Haven, 1974. Collected papers edited by Richmond Thomason.
12. R. Muskens. *Meaning and Partiality.* PhD thesis, University of Amsterdam, 1989.
13. R. Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.
14. N. Perera and A. Ranta. Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*, 2007.
15. C. Pollard. Higher-Order Categorial Grammar. In M. Moortgat, editor, *Proceedings of the Conference on Categorial Grammars (CG2004), Montpellier, France*, pages 340–361, 2004.
16. D. Prawitz. *Natural Deduction.* Almqvist & Wiksell, Stockholm, 1965.
17. A. Ranta. *Type Theoretical Grammar.* Oxford University Press, 1994.
18. A. Ranta. Grammatical Framework Homepage, 2002. `www.cs.chalmers.se/~aarne/GF/`.
19. A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.
20. A. Ranta. Modular Grammar Engineering in GF. *Research on Language and Computation*, 2007. To appear.
21. M. Rayner, B. A. Hockey, and P. Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler.* CSLI Publications, 2006.
22. H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.
23. B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1998.