The type theory and type checker of GF

Petri Mäenpää
Nokia Telecommunications
pmaenpaa@cs.hut.fi

Aarne Ranta
Chalmers & Gothenburg University
aarne@cs.chalmers.se

Abstract

GF (Grammatical Framework) is a Logical Framework enriched with concrete syntax specifications. Ordinary type-theoretical judgements of typing and definitional equality specify a theory. In addition, a judgement carries a description of how to produce a string in concrete syntax. The intended principal application area of GF is natural languages. It describes formal languages just as well, although less general tools exist that are optimized for them. Natural and formal languages can be combined in an interface to a proof editor of a logical framework by means of GF. Indeed, such an experimental interface has been made for Agda as an application of GF. This presentation focuses on matters familiar from syntax-directed editors of logical frameworks, although GF also has other applications: in semantically precise multilingual natural language documentation, and in compiling the programming language Shines.

1 Overview of GF

This paper presents the type-theoretical structure and type checking principles of GF (Grammatical Framework). GF is a variant of Martin-Löf's higher-level type theory with metavariables and rules for concrete syntax, based on type-theoretical grammar (Ranta 1994). We shall use the term type theory for Martin-Löf's higher-level type theory. Section 2 contains the basic rules of type theory that GF uses.

Consider the following specification of a fragment of arithmetic in type theory extended with definitions by pattern equations, as in ALF.

```
Nat, Prop : set
zero : Nat
succ : (x:Nat)Nat
sum : (x:Nat)(y:Nat)Nat
  sum(x,zero) = x : Nat
  sum(x,succ(y)) = succ(sum(x,y)) : Nat
EqNat : (x:Nat)(y:Nat)Prop
```

We would usually like to use a less formal *concrete syntax* instead of the above *abstract syntax.* Logical frameworks do not help here, although many have a user interface with *layout conventions.* They might allow a concrete syntax where `0` stands for `zero`, `x'` for `succ(x)`, `(x+y)` for `sum(x,y)`, and `x=y` for `EqNat(x,y)`.

In GF one may replace these layout conventions by logically rigorous concrete syntax definitions in the framework itself. This is done by extending the above type-theoretical *theory* into the GF *grammar*

```
Nat, Prop : cat
zero : Nat - "0"
succ : (x:Nat)Nat - x "'"
sum : (x:Nat)(y:Nat)Nat - "(" x "+" y ")"
  sum(x,zero) = x : Nat
  sum(x,succ(y)) = succ(sum(x,y)) : Nat
EqNat : (x:Nat)(y:Nat)Prop - x "=" y
```

The grammar first defines the *syntactic categories* (or categories for short) `Nat` and `Prop`. They are types of expressions, the syntactic counterpart of the "semantic" notion of type. The functions `zero`, `succ`, `sum` and `EqNat` now form *abstract syntax trees*. Their rules are annotated syntactically with *linear patterns*, which specify the string that corresponds to a syntax tree. The string is the *linearization* of the tree, and the tree is a *parse tree* of the string.

A linear pattern specifies the concrete syntax of an expression generated by a grammatical rule. The term *expression* is in fact ambiguous: it can refer to syntax trees as well as the corresponding strings. We shall let the context disambiguate which is meant.

The grammar produces for example the strings `0`, `0''` and `(0''+0)`, which denote natural numbers, as well as the string `0 = 0''`, which denotes a proposition. The pattern equations imply that the strings `(0''+0)` and `(0+0'')` denote the same natural number. This is because the corresponding syntax trees compute into the same value `succ(succ(zero))`.

Moreover, GF contains a generic *parser*, *generator* and *type checker*. They operate on an arbitrary GF grammar, just as the type checker of a logical framework operates on an arbitrary theory.

A GF grammar can be seen as a syntactically annotated theory. A form of this idea appears already in Curry (1963). It contrasts to attribute grammar (Knuth 1968) and its relatives, for example Yacc, that consist of syntactic rules with semantic annotations. Mäenpää (1998) is an earlier account of formal languages in terms of type-theoretical grammar, with a systematic comparison to attribute grammar.

One may also want to interface a proof editor with English or some other natural language. GF allows this on a par with its treatment of formal languages. It is instructive to compare a natural language grammar to the one above with respect to syntax and semantics. We replace the syntactic annotations with very simple ones for mathematical English, disregarding deeper English grammatical structure (see the GF documentation for that). Only the following definitions change.

```
zero : Nat - "zero"
succ : (x:Nat)Nat - "the successor of" x
sum : (x:Nat)(y:Nat)Nat - "the sum of" x "and" y
EqNat : (x:Nat)(y:Nat)Prop - x "is equal to" y
```

To illustrate the descriptive power of GF further, we enrich the grammar with another domain of individuals besides natural numbers, namely non-negative rationals. This is done by representing `Nat` now as a set expression, letting `Rat` be another, and declaring a new category `Elem` parametrized over `Set`. To define expressions for proofs of propositions, we furthermore declare a category `Proof`, parametrized over `Prop`.

```
Set, Prop : cat ;    Elem(X) : cat (X:Set) ;   Proof(X) : cat (X:Prop)
```

These categories represent the basic types of type theory. `Elem` must be parametric, because the judgement Elem : (set)type is malformed, and similarly for `Proof`.

The rules for natural numbers now take the following form, with some numerals defined by pattern matching.

```
Nat : Set - "N"
zero : Elem(Nat) - "0"
succ : (x:Elem(Nat))Elem(Nat) - x "'"
sum : (x:Elem(Nat))(y:Elem(Nat))Elem(Nat) - "(" x "+" y ")"
  sum(x,zero) = x : Elem(Nat)
  sum(x,succ(y)) = succ(sum(x,y)) : Elem(Nat)
EqNat : (x:Elem(Nat))(y:Elem(Nat))Prop - x "=" y
two : Elem(Nat) - "2"
  two = succ(succ(zero)) : Elem(Nat)
four : Elem(Nat) - "4"
  four = sum(two,two) :  Elem(Nat)
eight : Elem(Nat) - "8"
  eight = sum(four,four) :  Elem(Nat)
```

Rationals are represented as pairs of natural numbers that denote fractions. The denumerator of a fraction is positive by construction. They also have an equality predicate.

```
Rat : Set - "Q"
zeroRat : Elem(Rat) - "0"
frac : (x:Elem(Nat))(y:Elem(Nat))Elem(Rat) - x "/" y "'"
EqRat : (x:Elem(Rat))(y:Elem(Rat))Prop - x "=" y
```

This grammar shows that GF is able to describe *overloading*, whereas ordinary logical frameworks support it only by layout conventions. For instance, the string `0` is *ambiguous*. It can denote either a natural number or a rational number. The ambiguity is resolved on the level of syntax trees: `zero` differs from `zeroRat`. Deciding which syntax tree an ambiguous string corresponds to requires type-checking in addition to parsing.

Another example of overloading is the string `=`, which corresponds to the functions `EqNat` and `EqRat`, as well as to the general equality predicate

```
Eq :  (A:Set)(x:Elem(A))(y:Elem(A))Prop - x "=" y
```

This rule furthermore illustrates overloading combined with *type argument hiding*, a layout convention familiar from logical frameworks. The first argument of `Eq` is hidden by its *omission* in the linear pattern.

Proof arguments can be omitted as well. The following definition of the inverse function illustrates this. It also requires defining a non-zero predicate, a predecessor function, and a rule for proving that positive rationals differ from zero.

```
notZero : (x:Elem(Rat))Prop - x "<>" "0"
pred : (x:Elem(Nat))Elem(Nat) - "(" x "- 1 )"
  pred(zero) = zero : Elem(Nat)
  pred(succ(x)) = x : Elem(Nat)
positiveIsNotZero : (x:Elem(Nat))(y:Elem(Nat))Proof(notZero(frac(succ(x),y))) -
                                                         x "'" "/" y "'" "<>" "0"
inv : (z:Elem(Rat))(h:Proof(notZero(z)))Elem(Rat) - "1" "/" "(" z ")"
  inv(frac(x,y),h) = frac(succ(y),pred(x)) : Elem(Rat)
```

The `inv` rule states that the inverse of a rational number `z` exists only `z` is not zero. The proof `h` of this condition is omitted in the concrete syntax `1 / (z)` for the inverse, although it is a constituent of the syntax tree `inv(z,h)`. This kind of omission of proofs is a commonplace in informal mathematical language.

Yet another phenomenon illustrated by the grammar is *permutation* of arguments of a syntax tree in the concrete syntax. This occurs in the rule

```
sumList : (a:Elem(Nat))(d:Elem(Nat))(f:(x:Elem(Nat))Elem(Nat))Elem(Nat) -
                        "sum" "[" f "|" x "<-" "[" a ".." a "+" d "]" "]"
  sumList(a,zero,f) = f(a) : Elem(Nat)
  sumList(a,succ(d),f) = sum(f(sum(a,succ(d))),sumList(a,d,f)) : Elem(Nat)
```

where the arguments `a`, `d` and `f`, in left-to-right order, are permuted to `f`, `a`, `d` in the linear pattern. (The concrete syntax is that of a list comprehension.)

The `sumList` rule also illustrates *reduplication* of arguments. The argument `a` occurs twice in the linear pattern. Moreover, the linear pattern contains an occurrence of `x`, which is a *bound variable* of the argument `f`. This argument is a function from natural numbers to natural numbers. Section 2 describes the linearization of bound variables in more detail.

The grammar illustrates how a linear pattern contains the arguments of the corresponding syntax tree, with possible omissions, permutations and reduplications, and strings inserted in between. GF allows all of these operations in a logically rigorous way. They go beyond the expressive power of context-free grammar. The omission of arguments raises the question of how GF grammars can be parsed. As Section 3 explains, parsing is successful in virtue of metavariables.

Also some other logical frameworks support forms of syntactic annotation that are more restricted than those of GF. Thus for example Isabelle (Paulson 1998, chapter 7) allows the use of *priority grammars* and a *mixfix notation* to specify concrete syntax. An example is

```
  "plus" : [exp, exp] -> exp ("_ + _" [0,1] 0)
```

Here `0` and `1` are *precedence* values. The left argument of `plus` is expected to have precedence `0` and the right argument `1`. The whole expression receives the precedence value `0`. Parentheses are inserted into produced strings according to this precedence specification.

GF also supports specifying precedence information and corresponding parentheses. This comes out as a special case of *morphological* information. Morphological variation, although ubiquitous in natural languages, is scant in formal languages: precedence is often the only example of it. Section 2 describes the formal rules of morphological variation, while the below GF definition of the Isabelle `plus` illustrates its use to define precedences:

```
  plus : (x:exp)(y:exp)exp -
    lpar(0,Prec(x)) x rpar(0,Prec(x)) "+" lpar(1,Prec(y)) y rpar(1,Prec(y)) - 0
```

This definition uses a basic category `exp` of expressions, and the morphological operations `lpar` and `rpar`. They operate on a *morphological parameter* for precedence, whose possible values are (at least) `0` and `1`. By comparing the precedences `m` and `n`, `lpar(m,n)` produces either a left parenthesis or the empty string, and `rpar(m,n)` produces either the right parenthesis or the empty string. (Cf. Appendix B for more examples.)

In contrast to GF, the mixfix notation of Isabelle does not support argument hiding, permutation or reduplication. The arguments of the function are represented by the character `_` in the linear pattern, and this makes sense only if the arguments occur exactly once and in exactly the same order as in the function declaration. Isabelle does allow overloading, which can be resolved by type inference. Its use of precedences is a special case of the morphological parameters of GF.

Some logical frameworks do support argument hiding. For example, in ALF one may specify a number of arguments to be hidden in a function application, starting the count from the left. This is a special case of argument omission in GF. The `Eq` function above is an example with one argument hidden.

One can also compare GF to LaTeX (Lamport 1985), whose macros correspond to GF linear patterns without parameters. The definition of a LaTeX macro allows for arbitraty permutations, repetitions, and omissions of arguments. For instance, the inverse function could be represented thus:

```
  \newcommand{\inv}[3]{#1 / (#2)}
```

with the omission of the third argument. An important difference between GF and LaTeX is, of course, that the latter only has one type of expressions: any piece of LaTeX code may appear at any argument place of any macro.

Finally, from the point of view of presenting proofs, GF can be compared with the text generation functionality of Coq (Coscoy, Kahn, and Théry 1995). In Appendix C, some of the text formats from that work are used in linearization rules for proof constants. In Coq, natural language generation stops at the level of proof structure, so that the propositions contained in those proofs remain in formal notation. Moreover, the text generation rules are hard-wired rather than user-definable. In fact, the emphasis of text generation in Coq has been on "optimizing" proof texts so as to make them short and idiomatic. This aspect of Coq has no counterpart in GF yet.

2 Forms of judgement and rules of inference

The starting point of GF was the higher-level type theory of Martin-Löf and the way in which it was used for linguistic analysis in Ranta (1994; see chapter 8 for the version of type theory used). Besides syntactic annotations, GF differs from Martin-Löf's type theory in two ways: it has no predefined basic types, and, like ALF (Magnusson 1994), it has metavariables to stand for undefined constructions.

Metavariables are used in GF for type checking and user interaction, and they will be discussed in more detail in Section 3. As for basic types, Martin-Löf's two rules

$$\mathrm{set} : \mathrm{type}, \quad \mathrm{elem}(A) : \mathrm{type} \;\; (A : \mathrm{set})$$

are replaced by a *scheme for basic type declarations*:

$$C(x_1, \ldots, x_n) : \mathrm{type} \;\; (x_1 : \alpha_1, \ldots, x_n : \alpha_n),$$

that is, $C(x_1, \ldots, x_n)$ is a type depending on the variables $x_1 : \alpha_1, \ldots, x_n : \alpha_n$. Types are then formed by instantiations of variables in basic types and by dependent function type formation,

$$\frac{\alpha : \mathrm{type} \quad \begin{array}{c}(x : \alpha)\\ \beta : \mathrm{type}\end{array}}{(x : \alpha)\beta : \mathrm{type}}.$$

From the linguistic point of view, the basic types of GF play the role of syntactic categories. Basic type declarations are annotated by definitions of *linearization types*, which define the behaviour of objects of those types in linearization:

$$\begin{cases} C(x_1, \ldots, x_n) : \mathrm{type} \;\; (x_1 : \alpha_1, \ldots, x_n : \alpha_n), \\ C^{\mathrm{o}} \;=\; L : \mathrm{lintype}. \end{cases}$$

Linearization types have a structure of their own, definable in simple type theory. There is a type str of *strings*, and user-definable *parameter types*, which are finite sets of parameter values. Examples of parameter types are the English (or French, German, or Latin) number, the German (or Latin) gender, and the Latin case:

$$\mathrm{Num} = \{\mathrm{sg}, \mathrm{pl}\}, \quad \mathrm{Gen} = \{\mathrm{masc}, \mathrm{fem}, \mathrm{neut}\}, \quad \mathrm{Cas} = \{\mathrm{nom}, \mathrm{acc}, \mathrm{gen}, \mathrm{dat}, \mathrm{abl}\}.$$

Parametre types can be associated with syntactic categories either as *variable features* or as *inherent features*. These correspond respectively to the inherited and synthesized attributes of attribute grammar (Knuth 1968, see Mäenpää 1998 for a discussion of the distinction in terms of type-theoretical grammar). For instance, German and Latin common nouns have variable number and case, and inherent gender: *verbum* ("word") has forms for different numbers and cases, such as *verbum* (sg. nom. and acc.) and *verborum* (pl. gen.), but not for different genders—it is inherently of neuter gender. Adjectives have both gender, number, and case variable. Verbs in many languages have variable number, mode, and tense, but no case.

The general form[1] of the linearization type of a category α is

$$((\mathrm{Var}_\alpha)\mathrm{str}, \mathrm{Inh}_\alpha)$$

that is, a pair whose first component is a function from the variable features of α to strings, and the second component is a tuple of inherent features of α. For instance, Latin common nouns and adjectives have the following linearization types:

$$\mathrm{CN}^{\mathrm{o}} \;=\; ((\mathrm{Num}, \mathrm{Cas})\mathrm{str}, (\mathrm{Gen})), \;\; \mathrm{Adj}^{\mathrm{o}} \;=\; ((\mathrm{Gen}, \mathrm{Num}, \mathrm{Cas})\mathrm{str}, ()).$$

The *grammatical rules* of GF are *function declarations* annotated by *linearization functions*.

$$\begin{cases} F \;:\; (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha, \\ F^{\mathrm{o}} \;=\; l : (x_1 : \alpha_1^{\mathrm{o}}) \cdots (x_n : \alpha_n^{\mathrm{o}})\alpha^{\mathrm{o}}. \end{cases}$$

[1]The actual definition of linearization types is slightly more general than this, replacing strings by token lists, and recognizing tuples of token lists as linearizations of so-called discontinuous constituents.

Thus the linearization function F^{o} corresponding to a function F is a function from the linearization types of the arguments to the linearization type of the value.

A function type $(x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha$ can always be seen in the form in which α is a basic type (that is, not a function type itself). GF uses this view to impose the rule of *full application*, that is, application whose value is an object of a basic type. Such a function application can always be linearized into a string (which possibly depends on variable features), and its inherent features can be calculated. The full application rule is completed by the generic linearization rule of GF:

$$\frac{F \;:\; (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha \quad a_1 : \alpha_1 \quad \ldots \quad a_n : \alpha_n(x_1 = a_1, \ldots, x_{n-1} = a_{n-1})}{\left\{ \begin{array}{l} F(a_1, \ldots, a_n) : \alpha_n(x_1 = a_1, \ldots, x_n = a_n) \\ F(a_1, \ldots, a_n)^{\mathrm{o}} \;=\; F^{\mathrm{o}}(a_1^{\mathrm{o}}, \ldots, a_n^{\mathrm{o}}) : \alpha^{\mathrm{o}} \end{array} \right.}$$

Linearization is thus *compositional*, in the sense that the linearization of a function application is a function of the linearizations of its immediate constituents.

For example, the rule of adjectival modification in Latin is formalized as follows:

$$\left\{ \begin{array}{l} \mathrm{Mod} : (A : \mathrm{CN})(B : \mathrm{Adj}(A))\mathrm{CN} \\ \mathrm{Mod}^{\mathrm{o}}((A, (g)), (B, ())) = ((n)(c)(A(n, c) \mathbin{++} B(g, n, c)), (g)) \end{array} \right.$$

As for the function declaration, notice the dependence of the category of adjectives on common nouns: semantically speaking, a common noun expresses a set, and an adjective expresses a propositional function over a set. As for the linearization function, we have used pattern matching against the linearization types of the arguments of the function. The result is a concatenation of the noun with the adjective, both of which receive the number and the case of the whole construction. The adjective also needs a gender, and it receives the inherent gender g of the noun. The inherent gender of the whole construction is likewise inherited from the noun. Now, given appropriate rules for the simple nouns *verbum* ("word") and *vita* ("life"), and the adjective *æternus* ("eternal"; assume it applies both to word and to life), we can form, for instance

vitam æternam, "eternal life", feminine noun in the singular accusative,

verborum æternorum, "of eternal words", neuter noun in the plural genitive.

The concrete GF notation that has been implemented is slightly different from the rule shown above, although it must, of course, contain the same information. What we would in fact write for Latin adjectival modification is

```
Mod : (A:CN)(B:Adj(A))CN - case (n,c) -> A(n,c) B(Gen(A),n,c) - Gen(A)
```

A GF *grammar* is a sequence of category and function declarations syntactically annotated in the way explained above. Given the parallel structures of ordinary ("semantic") types and linearization types, it is always possible to check the well-formedness of a GF grammar, both in the semantic and in the syntactic sense. Semantic type checking, which we inherit from Martin-Löf's type theory, is explained in Section 3. Syntactic type checking is easier, since it is just a special case of Hindley-Milner type checking without polymorphism. But it is vital for the proper functioning of a GF grammar in its algorithmic use, which comprises linearization and its inverse, *parsing*. The language of syntactic annotations is so designed that it is always possible to generate a parsing algorithm corresponding to a GF grammar.

In addition to category and function declarations, GF grammars can also contain definitions of parameter types and of string-valued functions on them (such as verb conjugations). Furthermore, there can be *semantic definitions*, which are judgements of the form

$$a = b : \alpha \ (x_1 : \alpha_1, \ldots, x_n : \alpha_n)$$

just like in Martin-Löf's type theory. Semantic definitions are the basis of notions like *computation*, *normal form*, and *paraphrase* with respect to GF grammars. It should be noted that linearization is not invariant under computation: the semantic equality of a and b does not guarantee the equality of the strings a^{o} and

b^{o}. This should not be surprising: just recall that $2+2$, 2×2, and 4 are different as expressions even though equal as numbers.

It remains to say a word about the linearization of bound variables. As an intuitive example, take the type-theoretical declaration of the universal quantifier,

$$\forall : (A : \mathrm{set})(B : (x : \mathrm{elem}(A))\mathrm{prop})\mathrm{prop}$$

and the ordinary expression of a universally quantified proposition,

$$(\forall x : A)B(x).$$

There is a systematic connection between the two, a connection that GF annotations should be able to make precise. In actual GF, the declaration of the universal quantifier with its syntactic annotation is

```
  forall : (A:Set)(B:(x:Elem(A))Prop)Prop - "( \forall" x@B ":" A ")" B
```

In this declaration, the bound variable `x` is used in the linearization rule alongside with the ordinary arguments `A` and `B`. The symbol `B` is used, not for an expression of the function type `(x:Elem(A))Prop`, but for an expression of the type `Prop`, where the variable `x` may occur free—the "body" of the function. In order for linearization rules like this to work, we need to require that all functions appear in their η-expanded form. Then it makes sense to distinguish syntactically between the variables and the body of a functional expression. The requirement of η-expansion is closely associated with the previously explained rule of full application.

One of the central ideas of the higher-level type theory of Martin-Löf is to localize all variable bindings in the abstraction rule. If a function declaration in GF has a function type among its argument types, the function is likewise treated as a variable-binding operator, and the syntactic annotation has to specify the way in which the variable binding is shown in the concrete expression. We have already seen two examples of this: the $\forall$ rule above, and the `sumList` rule in Section 1. In order to make the annotation language completely precise, we still have to define the linearization types of function types, as well as the linearization of function abstracts:

$$\frac{\alpha\ :\ \mathrm{type}\qquad \begin{array}{c}(x:\alpha)\\ \beta\ :\ \mathrm{type}\end{array}}{\left\{\begin{array}{l}(x:\alpha)\beta\ :\ \mathrm{type}\\ ((x:\alpha)\beta)^{\mathrm{o}} = (\mathrm{str}, \beta^{\mathrm{o}})\end{array}\right.},\qquad \frac{\begin{array}{c}(x:\alpha)\\ b:\beta\end{array}}{\left\{\begin{array}{l}(x)b : (x:\alpha)\beta\\ ((x)b)^{\mathrm{o}} = (x^{\mathrm{o}}, b^{\mathrm{o}})\end{array}\right.}.$$

In other words, the linearization type of a function type (if used as an argument type of a syntactic rule) is the type of strings paired with the linearization type of the value type. The linearization of an abstract is the abstracted variable symbol paired with the linearization of the body. We have used the notation x^{o} to denote the variable symbol x.

The rule of universal quantifier formation can now be formalized

$$\left\{\begin{array}{l}\forall : (A : \mathrm{set})(B : (x : \mathrm{elem}(A))\mathrm{prop})\mathrm{prop}\\ \forall^{\mathrm{o}}(A, (x, B)) = \text{"("} +\!\!+ \text{"}\forall\text{"} +\!\!+ x +\!\!+ \text{" : "} +\!\!+ A +\!\!+ \text{")"} +\!\!+ B\end{array}\right.$$

assuming there are no morphological parameters for set and elem, and simplifying the notation for their linearizations by ignoring empty tuples accordingly.

3 Type checking in GF

The starting point of GF's type checker is Coquand's (1996) algorithm for type checking dependent types. GF extends this algorithm by a treatment of metavariables and constraints. Furthermore, GF replaces the axiom type : type by the parametric axiom scheme for basic types presented in the Section 2. GF's type checker is thus generic over languages with arbitrary basic types. Appendix A presents the type checker's main functions, in Haskell code.

Following Coquand's algorithm, the type checker operates in terms of a distinction between (semantic) values and (syntactic) terms (the datatypes `Val` and `Term`). Values carry local *explicit substitutions*, wrapped together with a term into a *closure* (formed by the constructor `VClos`). An example is the closure value $x(x = 1)$, which consists of the variable term x with the local explicit substitution $(x = 1)$.

Now we describe the main differences to Coquand's algorithm. One that is also present in the implementation of Agda's proof checker, although in a different form, is the introduction of metavariables. In checking whether a metavariable ? has a type T, GF first looks up ? in the list of open goals of the current proof state. If it occurs there, with the type U, the algorithm adds the constraint $T = U$ to the proof state. Otherwise, it adds the new open goal $? : T$ to the proof state. In the former case, the local context $x_1 : \alpha_1, \ldots, x_m : \alpha_m$ of U is matched with the type environment $y_1 : \beta_1, \ldots, y_n : \beta_n$ of the proof state. The constraints $\alpha_i = \beta j$ for all syntactically equal variables $x_i = y_j$ are added to the proof state.

Perhaps the most significant difference to Coquand's algorithm is type checking basic types. Coquand's algorithm uses the basic type type, whereas GF has the parametric scheme for basic types. To check whether a basic type $C(a_1, \ldots, a_n)$ is a valid type, GF looks up C in the list of declared categories of a grammar. A category declaration contains a parameter list $x_1 : \alpha_1, \ldots, x_n : \alpha_n$. GF type checks the actual parameters $a_1 : \alpha_1, \ldots, a_n : \alpha_n(x_1 = a_1, \ldots, x_{n-1} = a_{n-1})$ of a category instantiation in turn (the function `check_args`).

Category and constant declarations make up the data type `Theory`. It is a component of the type checking environment of type `REnv`, in addition to the value environment, type environment, goals and constraints.

The last main difference to Coquand's algorithm is that GF type inference and type checking return a list of new open goals and constraints (the functions `infer_type` and `check_type`). Type inference of course returns a type as well. Coquand's type inference algorithm returns just a type upon success, and his type checking algorithm returns a boolean value. This difference is due to GF's introduction of metavariables and constraints into the algorithm.

Interactive proof editing in GF represents the proof state in terms of values and terms. However, it displays values to the user by first transforming them into terms, because he works in the context of a grammar that contains only terms. A value is displayed by carrying out the local explicit substitutions in the term. Agda uses a similar approach, although it resolves constraints in a different way (Coquand and Coquand 1999).

An important case of converting a value into a term in interactive proof editing is when a closure consists of a metavariable with local explicit substitutions. An example is $?(x = 1)$, where the substitution $x = 1$ has no direct effect on the metavariable ?.

GF displays a metavariable without substitutions to the user. They have to be retained implicitly in the proof state however, because metavariables are constants yet to be defined. Even if substitutions have no direct effect on a metavariable, because it is a constant, they may have an indirect effect through the constraints that determine the possible values that a metavariable may come to have. For instance, ? may be an expression depending on x, and the variable may get instantiated before ? itself is resolved.

Type checking in GF has a special function not present in ordinary logical frameworks: the type-theoretical control of overloading by resolving *ambiguities.* An ambiguous string may have several parse trees, whereas a tree is always linearized in a determinate way. Parsing a string defined by means of argument omission yields a syntax tree with metavariables in the omitted argument places. Type checking then determines their value, if possible. Sometimes ordinary unification does the job, but in general user interaction is required.

A situation where unification suffices is parsing the string `0=0`. Six possible syntax trees result: `EqNat(zero, zero)`, `EqRat(zeroRat,zeroRat)`, `Eq(?,zero,zero)`, `Eq(?,zeroRat,zeroRat)`, `Eq(?,zero,zeroRat)` and `Eq(?,zeroRat,zero)`. Type checking rules out the last two, because the second and third arguments of `Eq` are not of the same type. The use of dependent types in the grammar allows enforcing this constraint. It enables the type checker to produce the unsolvable constraint `Nat = Rat`. The unification component of the type checker automatically instantiates the third and fourth alternative syntax trees to `Eq(Nat,zero,zero)` and `Eq(Rat,zeroRat,zeroRat)`, respectively, so there are four type correct parses altogether.

Parsing ambiguous strings into syntax trees with metavariables also illustrates another crucial property of parsing in GF. Namely, the *completeness of parsing* is ensured by letting it terminate in syntax trees that may contain metavariables. Finding their instantiations requires finding proofs in general, and can hence be arbitrarily difficult. Ensuring the completeness of the parsing algorithm makes metavariables *necessary* in GF, whereas ordinary logical frameworks may choose to do without them. This is because they lack overloading and make proofs always explicit in expressions.

The use of dependent types in a grammar rules out semantically malformed expressions like `Eq(Nat,zero, zeroRat)` in a straightforward way. They simply cannot be constructed according to the typing rules. Such restrictions on forms of syntactic combination by means of dependent types can be contrasted to approaches like Higher Order Abstract Syntax (Pfenning and Elliott 1988) that use simple types instead, and discard such semantically malformed expressions by checking whether separate semantical well-formedness predicates hold.

4 Conclusion

We have presented Grammatical Framework's system of syntactic annotations for Martin-Löf's higher-level type theory. Its type-checker is generic with respect to grammars, rather than theories as in ordinary logical frameworks. It also has a parsing and a linearization algorithm, generic over grammars as well. A GF grammar is a theory with syntactic annotations, together with a scheme for introducing basic types instead of the single basic type of sets or propositions of type theory.

This system of syntactic annotations is sufficient for a wide variety of notations. For example, the layout conventions in implementations of logical frameworks, the priority grammars and mixfix notation of Isabelle, and the macros of LaTeX are special cases of concrete syntax annotations in GF. In fact, the GF annotations suffice for a "mathematical vernacular", a full natural language for mathematics.

GF can also be used as a prototyping tool for programming languages, as is done for Shines (Shines programming language). A GF grammar for a language is capable of describing its type-theoretical semantics, both static and dynamic, at the same time as its syntax, both abstract and concrete. Thus one GF file is sufficient for building the entire prototype of a new language.

References

[1] Agda home page. `http://www.cs.chalmers.se/ catarina/agda/`

[2] GF home page. `http://www.xrce.xerox.com/research/mltt/gf/gf-index/index.html`

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

[4] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26:167–177, 1996.

[5] Catarina and Thierry Coquand. Structured type theory. Preliminary version, June 1999. Available at `http://www.cs.chalmers.se/ coquand/type.html`

[6] Yanne Coscoy, Gilles Kahn, and Laurent Théry. "Extracting text from proofs", *Typed Lambda Calculus and Applications*, *Lecture Notes in Computer Science* 902, Springer-Verlag, Heidelberg, 1995.

[7] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968. Errata 5:95–96, 1971.

[8] Leslie Lamport. LaTeX. *A Document Preparation System. User's Guide and Reference Manual.* Addison-Wesley, Reading, 1985.

[9] Petri Mäenpää. Semantical BNF. In E. Gimenez and C. Paulin-Mohring (eds.), *Types for Proofs and Programs, International Workshop TYPES'96*, LNCS 1512, pp. 196–215, Springer, 1998.

[10] Lena Magnusson. *An implementation of ALF, a proof editor based on Martin-Löf's monomorphic type theory with explicit substitutions.* PhD thesis, Chalmers University of Technology, 1995.

[11] Per Martin-Löf. *Intuitionistic Type Theory.* Notes by G. Sambin of a series of lectures given in Padua, June 1980. Bibliopolis, Napoli, 1984.

[12] Lawrence C. Paulson. *Isabelle Reference Manual.* With contributions by Tobias Nipkow and Markus Wenzel, 1998. Available at
`http://sunsite.doc.ic.ac.uk/pub/0-Most-Packages/smlnj/isabelle/` .

[13] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pp. 199–208, Atlanta, Georgia, June 1988.

[14] Aarne Ranta. *Type-Theoretical Grammar.* Oxford University Press, 1994.

[15] Shines programming language. Documentation available at HiBase project home page
`http://hibase.cs.hut.fi/hibase/hibase.html` .

Appendix A: The type checker

```
type Ident = String
data Fun  = Fun  Ident
data Symb = Symb Ident
data Cat  = Cat  Ident
type MetaSymb = (Cat,Ident)

data Term = Cons Fun
         | Var  Symb
         | Meta MetaSymb
         | App Term Term
         | Abs Symb Term
         | Ground Cat [Term]       - basic types
         | Prod Symb Term Term     - dependent function types
         | Predef (Ident,Ident)    - predefined constants

data Val = VGen Int | VApp Val Val | VCons Fun | VClos Env Term | VType
        | VPredef (Ident,Ident)

type Theory  = ([(Cat, Env)], [(Fun, Val)])
type Env     =  [(Symb, Val)]
type Goals   = [((Cat, Ident), (Val, Env))]
type REnv    = (Theory, Env, Env, Goals, Constrs)
type Constrs = [(Val, Val)]

update :: Env -> Symb -> Val -> Env
update env x u = (x, u) : env

app :: Val -> Val -> Val
eval :: Env -> Term -> Val

app u v =
 case u of
  VClos env (Abs x e) -> eval (update env x v) e
  _                   -> VApp u v
```

```
eval env e =
 case e of
  Cons c    -> VCons c
  Var x     -> case lookup x env of
                     Just a -> a
                     _      -> error ("Unknown identifier: " ++ prTerm (Var x))
  App e1 e2 -> app (eval env e1) (eval env e2)
  Predef d  -> VPredef d
  _         -> VClos env e

whnf :: Val -> Val
whnf v =
 case v of
  VApp u w    -> app (whnf u) (whnf w)
  VClos env e -> eval env e
  _           -> v

infer_type :: Int -> REnv -> Term -> Err (Val, Goals, Constrs)
infer_type k env@((_, con), rho, co, qs, cs) e =
 case e of
  Cons c ->
   case lookup c con of
    Just a -> Ok (a, [], [])
    _      -> Bad ("Unknown constant: " ++ prTerm (Cons c))
  Predef (c,_) ->
   case lookup c predefRules of
    Just (a,_,_) -> Ok (term_val (predeftype2type a), [], [])
    _            -> Bad ("Unknown predefined constant: " ++ prTerm e)
  Var x  ->
   case lookup x co of
    Just a -> Ok (a, [], [])
    _      -> Bad ("Unknown variable: " ++ prTerm (Var x))
  Meta s@(i, n) ->
   case (lookup s [(s1, t) | (s1, (t, c)) <- qs]) of
    Just a -> Ok (a, [], [])
    _      -> Bad ("Unknown subgoal: " ++ prMeta (i, n))
  App f c ->
   case (infer_type k env f) of
    Ok (t, qs1, cs1)  -> case whnf t of
                          VClos env1 (Prod x a b) ->
                           let (qs2, cs2) = check_type k env c (VClos env1 a)
                           in Ok (VClos (update env1 x (VClos rho c)) b,
                                  qs2 ++ qs1, cs2 ++ cs1)
                          _ -> Bad "Product expected in application"
    Bad s -> Bad s
  _ -> Bad "Cannot infer type"

check_type :: Int -> REnv -> Term -> Val -> (Goals, Constrs)
check_type k env@((cat, con), rho, co, qs, cs) e v =
 case e of
  Meta s  ->
   case lookup s qs of
    Just (a, t) ->
     ([], (a, v) : [(t1, t2) | (x1, t1) <- co, (x2, t2) <- t, x1 == x2])
    _           -> ([(s, (v, co))], [])
  Abs x n ->
   case whnf v of
    VClos env1 (Prod y a b) ->
     let w = VGen k
     in check_type (k + 1)
                   ((cat,con), update rho x w, update co x (VClos env1 a), qs, cs)
                   n
                   (VClos (update env1 y w) b)
    _      -> error "Product expected in abstraction"
  Ground c es ->
   case whnf v of
    VType ->
     case lookup c cat' of
       Just as -> if length as == length es
                  then check_args k env [] es as
                  else ([],arityConstrs (length as) (length es))
       _       -> error ("Unknown type identifier in: " ++ prTerm e)
      where cat' = cat ++ [(Cat c, []) | (c,_) <- predefCats]
    _     -> error "Type expected"
  Prod x a b ->
   case whnf v of
    VType ->
     let (qs1, cs1) = check_type k env a VType
         (qs2, cs2) = check_type (k + 1) ((cat, con),
                                          update rho x (VGen k),
                                          update co x (VClos rho a),
                                          qs,
                                          cs)
                                b VType
     in (qs2 ++ qs1, cs2 ++ cs1)
    _ -> error "Type expected"
  c -> case infer_type k env e of
         Ok (a, qs1, cs1)  -> (qs1, (a, v) : cs1)
         Bad s -> (typeProblem s, [])

check_args :: Int -> REnv -> Env -> [Term] -> Env -> (Goals, Constrs)
check_args k env@(_, rho, _, _, _) env1 es cont =
 case (es, cont) of
  ([], []) -> ([], [])
  (a : l, (x, VClos [] t1) : m) ->
     let (qs1, cs1) = check_type k env a (VClos env1 t1)
         (qs2, cs2) = check_args k env (update env1 x (VClos rho a)) l m
     in (qs2 ++ qs1, cs2 ++ cs1)
  _         -> error "Malformed family of types"
```

Appendix B: A GF grammar of Mini ML

The following is a piece of real GF code, describing a functional programming language with the basic types of non-negative decimal integers and Boolean values, with function types and cartesian products, and with `let` expressions. To give an example, the syntax tree

```
  Prg(Prod(Bool,Int),
      App(Int,Prod(Bool,Int),
          Abs(Int,Prod(Bool,Int),
              (x)Pair(Bool,Int,True,RP(Int,Int,Pair(Int,Int,Zero,x)))),
          UsePos(AddDig(UseDig(Dig2),Dig3))))
```

of type `Prog` is linearized into the string

```
  (\ x -> (true, snd (0, x)))23 : Bool * Int
```

and can be computed into `(true, 23) : Bool * Int`, by using the definitions of the grammar.

The grammar starts with the specification of a `Tokenizer`, followed by a declaration of morphological `Parametres` and their values, definitions of morphological `Operations`, declarations of `Categories` with their dependencies and linearization types, declarations of explicit `Variables`, and, finally, `Rules` and `Definitions`. For full details of the grammar format, we refer to the documentation on GF home page.

```
Tokenizer code Tokens "->" ;

Parametres Prec(p1,p2,p3) ; (* precedence *)

Categories Prog ; Type - - Prec ; Exp(Type) - - Prec ; Dig ; Pos ;

Variables x, y, z, X, Y, Z (Exp) ; (* variables of type Exp *)

Operations      (* parentheses as function of precedence *)
par(Prec,Prec) =
 case (p1,p)     -> ("",""),
      (p2,p1)   -> ("(",")"), (p2,p2)   -> ("",""), (p2,p3) -> ("",""),
      (p3,p1) -> ("(",")"), (p3,p2) -> ("(",")"), (p3,p3) -> ("","") ;

Rules
Dig1.    Dig -> 1 ; (* decimal integers, using context-free notation *)
Dig2.    Dig -> 2 ;
Dig3.    Dig -> 3 ;
Dig4.    Dig -> 4 ;
Dig5.    Dig -> 5 ;
Dig6.    Dig -> 6 ;
Dig7.    Dig -> 7 ;
Dig8.    Dig -> 8 ;
Dig9.    Dig -> 9 ;
UseDig.  Pos -> Dig ;
AddDig.  Pos -> Pos Dig ;
AddZero. Pos -> Pos 0 ;

Prg  : (A:Type)(a:Exp(A))Prog - a ":" A ; (* program with its type *)

Int    : Type - "Int"    - p3 ;   (* the type of integers *)
Zero   : Exp(Int)        - "0" - p3 ;
UsePos : (p:Pos)Exp(Int) - p - p3 ;

Bool  : Type - "Bool" - p3 ;    (* the type of truth values *)
True  : Exp(Bool) - "true" - p3 ;
False : Exp(Bool) - "false" - p3 ;

Fun  : (A:Type)(B:Type)Type -      (* the function type *)
  par.1(p2,Prec(A)) A par.2(p2,Prec(A)) "->" B - p1 ;
Abs  : (A:Type)(B:Type)(b:(x:Exp(A))Exp(B))Exp(Fun(A,B)) -
```

```
  "\\" x "->" b - p1 ;
App  : (A:Type)(B:Type)(c:Exp(Fun(A,B)))(a:Exp(A))Exp(B) -
  par.1(p2,Prec(c)) c par.2(p2,Prec(c)) par.1(p3,Prec(a)) a par.2(p3,Prec(a)) - p2 ;

Prod : (A:Type)(B:Type)Type -       (* the cartesian product *)
  par.1(p2,Prec(A)) A par.2(p2,Prec(A)) "*" par.1(p2,Prec(B)) B par.2(p2,Prec(B)) -
  p2 ;
Pair  : (A:Type)(B:Type)(a:Exp(A))(b:Exp(B))Exp(Prod(A,B)) -
  "(" a "," b ")" - p3 ;
LP    : (A:Type)(B:Type)(c:Exp(Prod(A,B)))Exp(A) -
  "fst" par.1(p3,Prec(c)) c par.2(p3,Prec(c)) - p2 ;
RP    : (A:Type)(B:Type)(c:Exp(Prod(A,B)))Exp(B) -
  "snd" par.1(p3,Prec(c)) c par.2(p3,Prec(c)) - p2 ;

Let : (A:Type)(B:Type)(a:Exp(A))(b:(x:Exp(A))Exp(B))Exp(B) -
  "let" x "=" a "in" b - p1 ;  (* local definition *)

Definitions
Int  = {Zero,UsePos} : Type ;
Bool = {True,False} : Type ;
App(A,B,Abs(A,B,b),a) = b(a) : Exp(B) ;
LP(A,B,Pair(A,B,a,b)) = a    : Exp(A) ;
RP(A,B,Pair(A,B,a,b)) = b    : Exp(B) ;
Let(A,B,a,b)          = b(a) : Exp(B) ;
```

Appendix C: English and French GF grammars for mathematical proofs

The following two grammars give rules for a fragment of many-sorted predicate calculus, such as defined in Martin-Löf (1984). The grammars have both proposition formation and proof rules. An example syntax tree is

```
  ThmWithProof(Neg(Abs),NegI(Abs,(x)Hypo(Abs,x)))
```

of the type `Text`. Its English and French linearizations are

> Theorem. It is not the case that we have a contradiction.
> Proof. Assume we have a contradiction. By hypothesis, we have a contradiction. Hence, it is not the case that we have a contradiction.
>
> Théorème. Il n'est pas vrai que nous ayons une contradiction.
> Démonstration. Supposons que nous avons une contradiction. Par hypothèse, nous avons une contradiction. Donc, il n'est pas vrai que nous ayons une contradiction.

The structure of GF grammars is summarized in the Appendix B above. We start with the English grammar.

```
Tokenizer text ;

Parametres Num(sg,pl) ;

Operations nomReg(Num) = case (sg) -> "", (pl) -> "s" ;

Categories Text ; Set - Num ; Prop ; Elem(Set) ; Proof(Prop) ;
```

```
Variables x,y,z,k,l,m,n,a,b,c (Elem) ; h,r,t (Proof) ;

Rules
ThmWithProof : (A:Prop)(a:Proof(A))Text -    (* theorem with a proof *)
  "Theorem ." A ". Proof ." a "." ;               (* shows the proof *)
ThmWithTrivialProof : (A:Prop)(a:Proof(A))Text -
  "Theorem ." A ". Proof . Trivial ." ;           (* hides the proof *)
Conj : (A:Prop)(B:Prop)Prop -
  A "and" B ;
Univ : (A:Set)(B:(x:Elem(A))Prop)Prop -
  "for all" A(pl) x@B "," B ;
Abs : Prop -
  "we have a contradiction" ;
Neg : (A:Prop)Prop -
  "it is not the case that" A ;
ConjI  : (A:Prop)(B:Prop)(a:Proof(A))(b:Proof(B))Proof(Conj(A,B)) - (* proofs *)
  a "." b ". Hence" A "and" B ;
ConjEl : (A:Prop)(B:Prop)(c:Proof(Conj(A,B)))Proof(A) -
  c ". A fortiori," A ;
ConjEr : (A:Prop)(B:Prop)(c:Proof(Conj(A,B)))Proof(B) -
  c ". A fortiori," B ;
NegI : (A:Prop)(b:(x:Proof(A))Proof(Abs))Proof(Neg(A)) -
  "assume" A "." b ". Hence, it is not the case that" A ;
NegE : (A:Prop)(c:Proof(Neg(A)))(a:Proof(A))Proof(Abs) -
  a ". But" c ". We have a contradiction" ;
UnivI : (A:Set)(B:(x:Elem(A))Prop)(b:(x:Elem(A))Proof(B(x)))Proof(Univ(A,B)) -
  "consider an arbitrary" A(sg) x@b "." b ". Hence, for all" A(pl) x@B "," B ;
UnivE : (A:Set)(B:(x:Elem(A))Prop)(c:Proof(Univ(A,B)))(a:Elem(A))Proof(B(a)) -
  c ". Hence" B "for" x@B "set to" a ;
AbsE : (C:Prop)(c:Proof(Abs))Proof(C) -
  c ". We may conclude" C ;
Hypo : (A:Prop)(a:Proof(A))Proof(A) -
  "by hypothesis," A ;

Definitions
ConjEl(A,B,ConjI(A,B,a,b)) = a : Proof(A) ;
ConjEr(A,B,ConjI(A,B,a,b)) = b : Proof(B) ;
NegE(A,NegI(A,b),a) = b(a) : Proof(Abs) ;
UnivE(A,B,UnivI(A,B,b),a) = b(a) : Proof(B(a)) ;
Neg(A) = Impl(A,Abs) : Prop ;
NegI(A,b) = ImplI(A,Abs,b) : Prop ;
```

The following French grammar has exactly the same type-theoretical (abstract) part as the previous English grammar. In the linearization (concrete) part, it has a much richer morphology, even for this very limited fragment of language.

```
Tokenizer text ;

Parametres Gen(masc,fem) ; Num(sg,pl) ; Mod(ind,subj) ; Cas(nom,aa,dd) ;

Operations
nomReg(Num) = case (sg) -> "", (pl) -> "s" ;
adjReg(Gen,Num) = case (masc,n) -> nomReg(n), (fem,n)  -> "e" + nomReg(n) ;
elision = "e", "'" ("a", "e", "i", "o", "u", "y", "\\'e") ;
ne  = "n"  + elision ;
que = "qu" + elision ;
```

```
indef(Gen) = case (g) -> "un" + adjReg(g,sg) ;
tout(Gen,Num) = case (masc,sg) -> "tout", (masc,pl) -> "tous",
                     (fem,n) -> "tout" + adjReg(fem,n) ;
etre(Num,Mod) = case (sg,ind)  -> "est", (sg,subj) -> "soit",
                     (pl,ind)  -> "sont", (pl,subj) -> "soient" ;

Categories
Text ; Set - Num - Gen ; Prop - Mod ; Elem(Set) - Cas - Gen ; Proof(Prop) ;

Variables x,y,z,k,l,m,n,a,b,c (Elem) ; h,r,t (Proof) ;

Rules
ThmWithProof  : (A:Prop)(a:Proof(A))Text -             (* theorem with a proof *)
  "Th\\'eor\\`eme ." A(ind) ". D\\'emonstration ." a "." ;  (* shows the proof *)
ThmWithTrivialProof : (A:Prop)(a:Proof(A))Text -
  "Th\\'eor\\`eme ." A(ind) ". D\\'emonstration . Triviale ." ; (* hides proof *)
Conj : (A:Prop)(B:Prop)Prop -                          (* logical constants *)
  case (m) -> A(m) "et" B(m) ;
Univ : (A:Set)(B:(x:Elem(A))Prop)Prop -
  case (m) -> "pour" tout(Gen(A),pl) "les" A(pl) x@B "," B(m) ;
Abs  : Prop -
  case (ind)  -> "nous avons une contradiction",
       (subj) -> "nous ayons une contradiction" ;
Neg  : (A:Prop)Prop -
  case (m) -> "il" ne etre(sg,m) "pas vrai" que A(subj) ;
ConjI  : (A:Prop)(B:Prop)(a:Proof(A))(b:Proof(B))Proof(Conj(A,B)) - (* proofs *)
  a "." b ". Donc" A(ind) "et" B(ind) ;
ConjEl : (A:Prop)(B:Prop)(c:Proof(Conj(A,B)))Proof(A) -
  c ". A fortiori," A(ind) ;
ConjEr : (A:Prop)(B:Prop)(c:Proof(Conj(A,B)))Proof(B) -
  c ". A fortiori," B(ind) ;
NegI : (A:Prop)(b:(x:Proof(A))Proof(Abs))Proof(Neg(A)) -
  "supposons" que A(ind) "." b ". Donc , il n'est pas vrai" que A(subj) ;
NegE : (A:Prop)(c:Proof(Neg(A)))(a:Proof(A))Proof(Abs) -
  a ". Mais" c ". Nous avons une contradiction" ;
UnivI : (A:Set)(B:(x:Elem(A))Prop)(b:(x:Elem(A))Proof(B(x)))Proof(Univ(A,B)) -
  "consid\\'erons" indef(Gen(A)) A(sg) x@b "arbitraire." b ". Donc , pour"
  tout(Gen(A),pl) "les" A(pl) x@B "," B(ind) ;
UnivE : (A:Set)(B:(x:Elem(A))Prop)(c:Proof(Univ(A,B)))(a:Elem(A))Proof(B(a)) -
  c ". Donc" B(ind) "avec" x@B "remplac\\'e par" a(nom) ;
AbsE : (C:Prop)(c:Proof(Abs))Proof(C) -
  c ". Nous concluons" que C(ind) ;
Hypo : (A:Prop)(a:Proof(A))Proof(A) -
  "par hypoth\\`ese," A(ind) ;

(* Definitions as in English *)
```