Multilingual Syntax Editing in GF

Janna Khegai, Bengt Nordström, and Aarne Ranta

Department of Computing Science
Chalmers University of Technology and Gothenburg University
SE-41296, Gothenburg, Sweden
{janna, bengt, aarne}@cs.chalmers.se

Abstract. GF (Grammatical Framework) makes it possible to perform multilingual authoring of documents in restricted languages. The idea is to use an object in type theory to describe the common abstract syntax of a document and then map this object to a concrete syntax in the different languages using linearization functions, one for each language. Incomplete documents are represented using metavariables in type theory. The system belongs to the tradition of logical frameworks in computer science. The paper gives a description of how a user can use the editor to build a document in several languages and also shows some examples how ambiguity is resolved using type checking. There is a brief description of how GF grammars are written for new domains and how linearization functions are defined.

1 Introduction

1.1 Multilingual authoring

We are interested in the problem of editing a document in several languages simultaneously. In order for the problem to be feasible, we use a restricted language. The idea is to use a mathematical structure (an object in type theory) as the basic representation of the document being edited. This structure describes the abstract syntax of the document. Concrete representations of the document in the various languages are expressed using linearization functions, one function for each language. The process of producing a concrete representation from the abstract object is thus deterministic, each abstract object has only one concrete representation in each language. The reverse problem (parsing) is not deterministic, a given concrete representation may correspond to many abstract objects (ambiguity). The way we resolve ambiguity is by having an interactive system, an ambiguity results in an incomplete abstract object which has to be completed by the user. For instance, in the phrase *Dear friend* it is not clear whether the friend is male of female, thus making a translation into Swedish impossible. In the corresponding abstract object there is a field for gender which has to be filled in by the user before the editing is complete.

Type theory is a completely formal language for mathematics developed by Martin-Löf in the 70's [9]. Versions of it are extensively used under the title of Logical Frameworks in various implementations of proof editors like Coq [19], Alf [8], and Lego [7]. The type system of type theory is not only used to express syntactic well-formedness, but also semantic well-formedness. This means that a syntactically well-formed term also has a meaning. Moreover, it is often possible to use type checking to resolve ambiguities that a weaker grammatical description cannot resolve.

1.2 The GF Niche - Meaning-Based Technique

Grammatical Framework (GF) is a grammar formalism built upon a Logical Framework (LF). What GF adds to LF is a possibility to define concrete syntax, that is, notations expressing formal concepts in user-readable ways. The concrete syntax mechanism of GF is powerful enough to describe natural languages: like PATR [18] and HPSG [13], GF uses features and records to express complex linguistic objects. Although GF grammars are bidirectional like PATR and HPSG, the perspective of GF is on generation rather than parsing. This implies that grammars are built in a slightly different way, and also that generation is efficient enough to be performed in real time in interactive systems. Another difference from usual grammar formalisms is the support for multilinguality: it is possible to define several concrete syntaxes upon one abstract syntax. The abstract syntax then works as an interlingua between the concrete syntaxes. The development of GF as an authoring system started as a plug-in to the proof editor ALF, to permit natural-language rendering of formal proofs [5]. The extension of the scope outside mathematics was made in the Multilingual Document Authoring project at Xerox [3]. In continued work, GF has been used in areas like software specifications [4], instruction texts [6], and dialogue systems [17]. In general, GF works for any domain of language that permits a formal grammar. Since LF is more general than specific logical calculi, it is more flexible to use on different domains than, for instance, predicate calculus.

1.3 The Scope of the Paper

The GF program implementing the GF grammar formalism is a complex system able to perform many NLP tasks. For example, it can do the morphological analysis of French verbs, construct a letter in several languages, and even greet you in the morning using a speech synthesizer. In this paper, however, we choose to restrict the topic and only look at GF as a multilingual authoring tool. For a more elaborated description of the system, we refer to [16, 15].

The GF users can be divided into three competence levels:

- Author level
- Grammarian level
- Implementor level

On the author level all we can do is to work with pre-existing grammars. This level is described in Section 2. On the grammarian level we write grammars describing new language fragments. This, of course, requires acquaintance with the GF formalism. Examples of work on this level are given in Section 3. On both of these levels, we have some control, for example, over parsing algorithms to be used. However, the full control of parsing, linearization, graphics and other algorithms, is only accessible on the implementor level. Since GF is open source software, any user who wants can also become an implementor; but describing the implementor level is outside the scope of this paper.

2 The GF Syntax Editor

The graphical user interface implemented in the GF Syntax Editor hides the complexity of the system from the naive user. It provides access to the system functionality without requiring knowledge of the GF formalism. In this section we will show a simple example of GF syntax editing procedure.

When you start the GF editor you choose the topic and the languages you want to work with. For instance, we decide to work within the LETTER topic and want to have translations in four languages: English, Swedish, French and Finnish. You can create a new editing object by choosing a category from the New list. For example, to construct a letter, choose the Letter category (Fig. 1). In Fig. 2 you can see the created object in the tree form in the left upper part as well as linearizations in the right upper part. The tree representation corresponds to the GF language-independent semantic representation, the GF abstract syntax or interlingua. The linearizations area displays the result of translation of abstract syntax representation into the corresponding language using the GF concrete syntax.

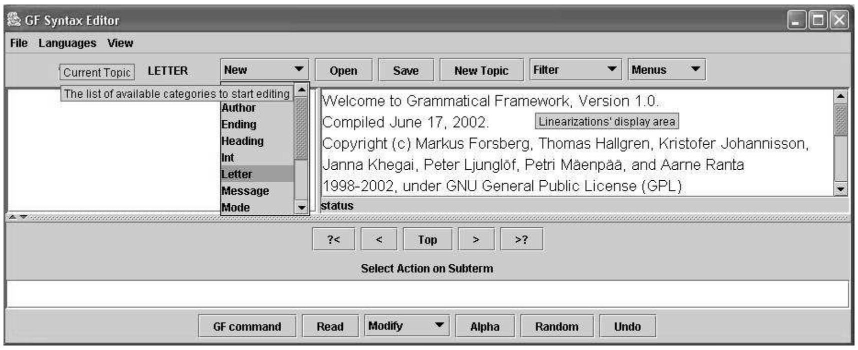

Fig. 1. The New menu shows the list of available categories within the current topic LETTER. Choosing the category Letter in the list will create an object of the corresponding type. The linearizations area contains a welcome message when the GF Editor has just been started.

According to the LETTER grammar a letter consists of a Heading, a Message and an Ending, which is reflected in the tree and linearizations structures.

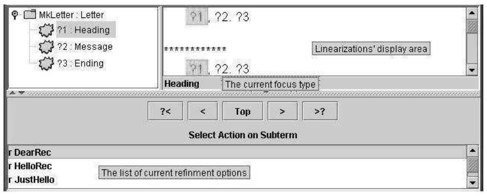

Fig. 2. The Abstract Syntax tree represents the letter structure. The current editing focus, the metavariable ?1 is highlighted. The type of the current focus is shown below the linearizations area. The context-dependent refinement option list is shown in the bottom part.

However, the exact contents of each of these parts are not yet known. Thus, we can only see question marks, representing metavariables, instead of language phrases in the linearizations.

Editing is a process of step-wise refinement, i.e. replacement of metavariables with language constructions. In order to proceed you can choose among the options shown in the refinement list. The refinement list is context-dependent, i.e. it refers to the currently selected focus. For example, if the focus is Heading, then we can choose among four options. Let us start our letter with the DearRec structure (Fig. 3(a)).

Now we have a new focus - metavariable ?4 of the type Recipient and a new set of refinement options. We have to decide what kind of recipient the letter has. Notice that the word *Dear* in Swedish and French versions is by default in male gender and, therefore, uses the corresponding adjective form. Suppose we want to address the letter to a female colleague. Then we choose the ColleagueShe option (Fig. 3(b)).

Notice that the Swedish and French linearizations now contain the female form of the adjective *Dear* , since we chose to write to a female recipient. This refinement step allows us to avoid the ambiguity while translating from English to, for example, a Swedish version of the letter.

Proceeding in the same fashion we eventually fill all the metavariables and get a completed letter like the one shown in Fig. 4(a).

A completed letter can be modified by replacing parts of it. For instance, we would like to address our letter to several male colleagues instead. We need first to move the focus to the Header node in the tree and delete the old refinement. In Fig. 5(a), we continue from this point by using the Read button, which invokes an input dialog, and expects a string to parse. Let us type *colleagues*.

The parsed string was ambiguous, therefore, as shown in Fig. 5(b), GF asks further questions. Notice that after choosing the ColleaguesHe option, not only the word *colleague* , but the whole letter switches to the plural, male form, see Fig. 4(b). In the English version only the noun *fellow* turns into plural, while in the other languages the transformations are more dramatic. The pronoun *you* turns into plural number. The participle *promoted* changes the number in the Swedish and French versions. The latter also changes the form of the verb *have*.

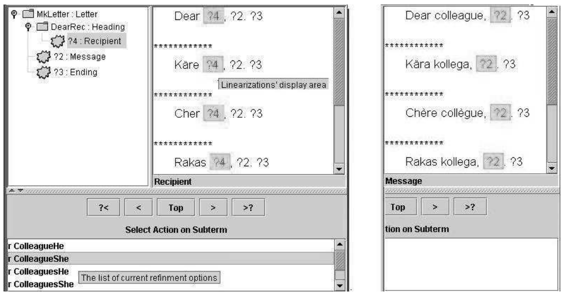

Fig. 3. (a) The linearizations are now filled with the first word that corresponds to *Dear* expression in English, Swedish, French and Finnish. The refinement focus is moved to the Recipient metavariable. (b) The Heading part is now complete. The adjective form changes to the corresponding gender after choosing the recipient.

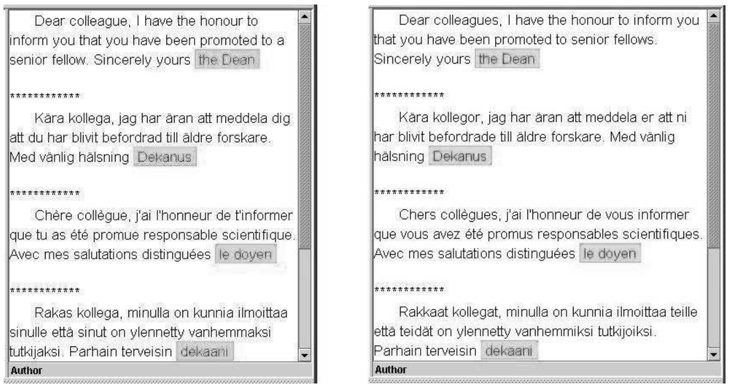

Fig. 4. (a) The complete letter in four languages. (b) Choosing the plural male form of the Recipient causes various linguistic changes in the letter as compared to (a).

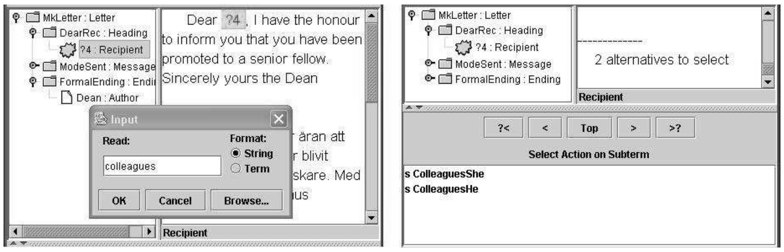

Fig. 5. (a) A refinement step can be done by using the Read button, which asks the user for a string to parse. (b) When the parsed string in (a) is ambiguous GF presents two alternative ways to resolve the ambiguity.

Both the gender and the number affect the adjective *dear* in French, but only the number changes in the corresponding Finnish adjective. Thus, the refinement step has led to substantial linguistic changes.

3 The GF grammar formalism

The syntax editor provided by GF is generic with respect to both subject matters and target languages. To create a new subject matter (or modify an old one), one has to create (or edit) an *abstract syntax*. To create a new target language, one has to work on a *concrete syntax*. Target languages can be added on the fly: if a new language is selected from the Language menu, a new view appears in the editor while other things remain equal, including the document that is being edited. Fig. 6 shows the effect of adding Russian to the above example.

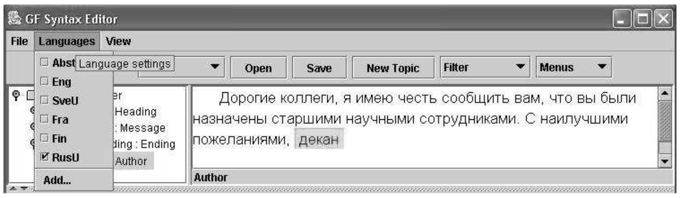

Fig. 6. Now we are able to translate the letter into Russian

The syntax editor itself is meant to be usable by people who do not know GF, but just something of the subject matter and at least one of the target languages. Authoring GF grammars requires expertise on both the subject matter and the target languages, and of course some knowledge of the GF grammar formalism.

A typical GF grammar has an abstract syntax of 1–3 pages of GF code, and concrete syntaxes about the double of that size. The use of resource grammars (Section 3.5) makes concrete syntaxes much shorter and easier to write.

3.1 Abstract syntax: simple example

An abstract syntax gives a structural description of a domain. It can be semantically as detailed and rigorous as a mathematical theory in a Logical Framework. It can also be less detailed, depending on how much semantic control of the document is desired. Asq an example, consider a fragment of the abstract syntax for letters:

```
cat Letter ; Recipient ; Author ; Message ;
    Heading ; Ending ; Sentence ;

fun MkLetter : Heading -> Message -> Ending -> Letter ;
fun DearRec : Recipient -> Heading ;
```

```
fun PlainSent : Sentence -> Message ;
fun ColleagueHe, ColleaguesShe : Recipient ;
```

The grammar has a set of categories `cat` and functions `fun`. The functions are used for building abstract syntax trees, and each tree belongs to a category. When editing proceeds, the system uses the abstract syntax to build a menu of possible actions: for instance, the possible refinements for a metavariable of type C are those functions whose value type is C.

3.2 Concrete syntax

Concrete syntax maps abstract syntax trees into linguistic objects. The objects can be simply strings, but in general they are records containing inflection tables, agreement features, etc. Each record has a type, which depends on the category of the tree, and of course on the target language. For instance, a part of the English category system for letters is defined as follows:

```
param Num = Sg | Pl ;
param Agr = Ag Num ;

lincat Letter = {s : Str} ;
lincat Recipient, Author = {s : Str ; a : Agr} ;
lincat Message = {s : Agr => Agr => Str} ;
lincat Heading, Ending = {s : Str ; a : Agr} ;
```

Both the author and the recipient have inherent agreement features (number), which are passed from them to the message body, so that right forms of verbs and nouns can be selected there:

```
lin MkLetter head mess end = {s =
      head.s ++ "," ++
      mess.s ! end.a ! head.a ++ "." ++
      end.s} ;
```

Different languages have different parameter systems. French, for instance, has gender in addition to number in the agreement features:

```
param Gen = Masc | Fem ; Num = Sg | Pl ; Agr = Ag Gen Num ;
```

3.3 Semantic control in abstract syntax

The main semantic control in the letter grammar is the structure that is imposed on all letters: it is, for instance, not possible to finish a letter without a heading. This kind of control would be easy to implement even using a context-free grammar or XML. Semantic control of more demanding kind is achieved by using the *dependent types* of type theory. For instance, the abstract syntax

```
cat Text ; Prop ; Proof (A : Prop) ;
fun ThmWithProof, ThmHideProof : (A : Prop) -> Proof A -> Text ;
```

defines mathematical texts consisting of a proposition and a proof. The type of proofs depends on propositions: the type checker can effectively decide whether a given proof really is a proof of a given theorem. Type checking also helps the author of the proof by only showing menu items that can possibly lead to a correct proof. Proof texts are linearized by to the following rules:

```
lin ThmWithProof A P =
      {s = "Theorem." ++ A.s ++ "Proof." ++ P.s ++ "Q.E.D."} ;
lin ThmHideProof A P =
      {s = "Theorem." ++ A.s ++ "Proof." ++ "Omitted."} ;
```

The latter form omits the proof, but the author is nevertheless obliged to construct the proof in the internal representation.

Mathematical texts with hidden proofs are a special case of *proof-carrying documents*, where semantic conditions are imposed by using dependent types in abstract syntax (cf. the notion of *proof-carrying code* [12]). Consider texts describing flight connections:

> *To get from Gothenburg to New York, you can first fly SK433 to Copenhagen and then take SK909.*

There are three conditions: that SK433 flies from Gothenburg to Copenhagen, that SK909 flies from Copenhagen to New York, and that change in Copenhagen is possible. These conditions are expressed by the following abstract syntax:

```
cat City ; Flight (x,y : City) ;

fun Connection :
      (x,y,z : City) -> (a : Flight x y) -> (b : Flight y z)
            -> Proof (PossibleChange x y z a b) -> Flight x z ;
fun PossibleChange :
      (x,y,z : City) -> Flight x y -> Flight y z -> Prop ;
```

The linearization rule for `Connection` produces texts like the example above, with internal representation that includes the hidden proof. We have left it open how exactly to construct proofs that a change is possible between two flights: this involves a proof that the departure time of the second flight lies within a certain interval from the arrival time of the first flight, the minimum length of the interval depending on the cities involved. In the end, the proof condition reduces to ordinary mathematical concepts.

3.4 Semantic disambiguation

An important application of semantic control is *disambiguation*. For instance, the English sentence

> *there exists an integer x such that x is even and x is prime*

has two French translations,

il existe un entier x tel que x soit pair et que x soit premier
il existe un entier x tel que x soit pair et x est premier

corresponding to the trees

```
Exist Int (\x -> Conj (Even x) (Prime x))
Conj (Exist Int (\x -> Even x)) (Prime x)
```

respectively. Both analyses are possible by context-free parsing, but type checking rejects the latter one because it has an unbound occurrence of x.

Another example of semantic disambiguation is the resolution of pronominal reference. The English sentence

if the function f has a maximum then it reaches it at 0

has two occurrences of *it*. Yet the sentence is not ambiguous, since it uses the predicate *reach*, which can only take the function as its first argument and the maximum as its second argument: the dependently typed syntax tree uses a pronominalization function

```
fun Pron : (A : Dom) -> Elem A -> Elem A
```

making the domain and the reference of the pronoun explicit. Linearization rules of `Pron` into languages like French and German use the domain argument to select the gender of the pronoun, so that, for instance, the German translation of the example sentence uses *sie* for the first *it* and *es* for the second:

wenn die Funktion f ein Maximum hat, dann reicht sie es bei 0

3.5 Application grammars and resource grammars

GF is primarily geared for writing specialized grammars for specialized domains. It is possible to avoid many linguistic problems just by ignoring them. For instance, if the grammar only uses the present tense, large parts of verb conjugation can be ignored. However, always writing such grammars from scratch has several disadvantages. First, it favours solutions that are linguistically *ad hoc*. Secondly, it produces concrete syntaxes that are not reusable from one application to another. Thirdly, it requires the grammarian simultaneously to think about the domain and about linguistic facts such as inflection, agreement, word order, etc. A solution is to raise the level of abstraction, exploiting the fact that GF is a functional programming language: do not define the concrete syntax as direct mappings from trees to strings, but as mappings from trees to structures in a *resource grammar*.

A resource grammar is a generic description of a language, aiming at completeness. From the programming point of view, it is like a *library module*, whose proper use is via type signatures: the user need not know the definitions of module functions, but only their types. This modularity permits a division of labour between programmers with different expertises. In grammar programming, there is typically a domain expert, who knows the abstract syntax and wants to map it into a concrete language, and a linguist, who has written the resource grammar and provided a high-level interface to it.

For instance, the resource grammar may contain linguistic categories, syntactic rules, and (as a limiting case of syntactic rules) lexical entries:

```
cat S ; NP ; Adj ;              -- sentence, noun phrase, adjective
fun PredAdj : Adj -> NP -> S ; -- "NP is Adj"
fun Condit : S -> S -> S ;     -- "if S then S"
fun adj_even : Adj ;           -- "even"
```

The author of a grammar of arithmetic proofs may have the following abstract syntax with semantically motivated categories and functions:

```
cat Prop ; Nat ;                 -- proposition, natural number
fun If : Prop -> Prop -> Prop ; -- logical implication
fun Ev : Nat -> Prop ;          -- the evenness predicate
```

The concrete syntax that she writes can exploit the resource grammar:

```
lincat Prop = S ; Nat = NP ;
lin If = Condit ;
lin Ev = PredAdj adj_even ;
```

Experience with GF has shown that the abstract interfaces to resource grammars can largely be shared between different languages. Thus a German resource grammar can have the same type signatures as the English one, with the exception of lexical rules. In this case we have

```
lin Ev = PredAdj adj_gerade ;
```

Yet the effects of these rules are language-dependent. German has more agreement and word order variation in the conditional and predication rules. For instance, the syntax tree `If (Ev n1) (Od n3)` is linearized as follows:

English: *if* 1 *is even then* 2 *is odd*
German: *wenn* 1 *gerade ist, dann ist* 2 *ungerade*

Of course, there are also cases where different linguistic structures must be used in the abstract syntax. For instance, the two-place predicate saying that x misses y is expressed by a two-place verb construction in both English and French, but the roles of subject and object are inverted (x *misses* y vs. y *manque à* x):

English: `lin Miss x y = PredVP x (ComplV2 verb_miss y)`
French: `lin Miss x y = PredVP y (ComplV2 verb_manquer x)`

4 Discussion

4.1 Comparison to other systems

In computer science, one of the earliest attempt of generating a syntax editor from a language description was the Mentor [2] system at INRIA. Another early example is the Cornell program synthesizer [20], which uses an attribute grammar formalism to describe the language.

The idea of using a strictly formalized language for mathematics to express the abstract syntax of natural language was proposed by Curry [1] and used by Montague [11] in his denotational semantics of English. The followers of Montague, however, usually ignore the abstract syntax and define relations between natural language and logic directly. This makes generation much harder than when using an abstract syntax tree as the primary representation.

The WYSIWYM system [14] by Power and Scott has many similarities with our system. It is also a system for interactive multi-lingual editing. WYSIWYM does not use a mathematical language to express abstract syntax and it seems not to be possible for the user to change the structure of what they call the knowledge base (our abstract syntax).

Processing natural language within restricted domains makes GF related to the KANT translation system [10]. Kant Controlled English (KCE) put constraints on vocabulary, grammar and document structure in order to reduce the amount of ambiguity in the source text in the pre-processing phase. The remaining ambiguities are resolved via interaction with the author in the authoring environment. The only source language in the KANT system is English. KANT does not use any formal semantic representation.

4.2 Future work

We would like to see the system as a (structured) document editor. This has many implications. The major part of the screen will in the future be devoted to the different documents and not to the various menus. The parts of the document which is not filled in – now indicated by metavariables – will have a meaningful label expressed in the language being presented. The natural action of refining a metavariable by parsing a text from the keyboard is to put the cursor on the metavariable and start typing. In the future it will also be possible to use completions, so when the user enters for instance a tab character the system responds with the longest unique possible continuation of the input together with a list of alternatives. Completion in GF can be controlled both by the application grammar and by statistics of the interaction history.

A grammarian-level user needs an advanced editor for editing grammar. So a natural idea is to extend GF to make it possible to edit the GF formalism itself.

Creating resource grammars is an important part in the development of GF, corresponding to the development of standard libraries for programming languages. The current libraries (end 2002) contain basic morphology, phrase structure, and agreement rules for English, French, German, and Swedish.

GF is generic program capable of using any set of grammars in an. Hardwired grammars, however, permit more light-weight implementations. We use the name *gramlets* for Java programs implementing such light-weight special-purpose editors. Gramlets can be used in PDAs and as web applets, and they can be automatically compiled from GF grammars.

References

1. 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.
2. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy. A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*, 1975.
3. M. Dymetman, V. Lux, and A. Ranta. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249, 2000.
4. R. Hähnle, K. Johannisson, and A. Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.
5. T. Hallgren and A. Ranta. An extensible proof text editor. In M. Parigot and A. Voronkov, editors, *LPAR-2000*, volume 1955 of *LNCS/LNAI*, pages 70–84. Springer, 2000.
6. K. Johannisson and A. Ranta. Formal verification of multilingual instructions. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2001.
7. Z. Luo and R. Pollack. LEGO Proof Development System. Technical report, University of Edinburgh, 1992.
8. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS 806, pages 213–237. Springer, 1994.
9. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.
10. T. Mitamura and E. H. Nyberg. Controlled English for Knowledge-Based MT: Experience with the KANT system. In *TMI*, 1995.
11. R. Montague. *Formal Philosophy.* Yale University Press, New Haven, 1974. Collected papers edited by R. Thomason.
12. G. C. Necula. Proof-Carrying Code. In *Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 106–119. ACM Press, 1997.
13. C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, 1994.
14. R. Power and D. Scott. Multilingual authoring using feedback texts. In *COLING-ACL*, 1998.
15. A. Ranta. GF Homepage, 2002. `www.cs.chalmers.se/~aarne/GF/`.
16. A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, to appear.
17. A. Ranta and R. Cooper. Dialogue systems as proof editors. In *IJCAR/ICoS-3*, Siena, Italy, 2001.
18. S. Shieber. *An Introduction to Unification-Based Approaches to Grammars.* University of Chicago Press, 1986.
19. Coq Development Team. Coq Homepage. `http://pauillac.inria.fr/coq/`, 1999.
20. T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.