Translating between Language and Logic: What Is Easy and What Is Difficult

Aarne Ranta

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

Abstract. Natural language interfaces make formal systems accessible in informal language. They have a potential to make systems like theorem provers more widely used by students, mathematicians, and engineers who are not experts in logic. This paper shows that simple but still useful interfaces are easy to build with available technology. They are moreover easy to adapt to different formalisms and natural languages. The language can be made reasonably nice and stylistically varied. However, a fully general translation between logic and natural language also poses difficult, even unsolvable problems. This paper investigates what can be realistically expected and what problems are hard.

Keywords: Grammatical Framework, natural language interface

1 Introduction

Mature technology is characterized by *invisibility*: it never reminds the user of its existence. Operating systems are a prime example. Still a decade ago, you'd better be a Unix hacker to do anything useful with a Unix computer. Nowadays Unix is hidden under a layer of Mac OS or Ubuntu Linux, and it works so well that the layman user hardly ever notices it is there.

When will formal proof systems mature? Decades of accumulated experience and improvements have produced many systems that are sophisticated, efficient, and robust. But using them is often an expert task—if not for the same experts as the ones who developed the systems, then at least for persons with a special training. One reason (not always the only one, of course) is that the systems use formalized proof languages, which have to be learnt. The formalized language, close to the machine language of the underlying proof engine, constantly reminds the user of the existence of the engine.

Let us focus on one use case: a student who wants to use a proof system as an infatigable teaching assistant, helping her to construct and verify proofs. This case easily extends to a mathematician who needs help in proving new theorems, and to an engineer who needs to verify software or hardware systems with respect to informal specifications. Now, the student must constantly perform manual conversions between the informal mathematical language of her textbooks and the formalism used by the proof system.

We can imagine this to be otherwise. Computer algebra systems, such as Mathematica [1], are able to manipulate normal mathematical notations, for instance, $\sqrt{x}$ instead

of its formalized representation `Sqrt[x]`. The support for normal mathematical language is one (although not the only one) of the reasons why computer algebras, unlike formal proof systems, have become main-stream tools in mathematics education.

Now, what is the proof-system counterpart of algebraic formulas in computer algebra? It is mathematical text, which is a mixture of natural language and algebraic formulas. The natural language part cannot be replaced by formulas. Therefore it is only by allowing input and output in text that proof system can hide their internal technology and reach the same level of maturity as computer algebras.

The importance of informal language for proof systems has of course been noticed several times. It has motivated systems like STUDENT [2], Mathematical Vernacular [3], Mizar [4], OMEGA [5], Isar [6], Vip [7], Theorema [8], MathLang [9], Naproche [10], and FMathL [11]. These systems permit user interaction in a notation that resembles English more than logical symbolisms do. The notations are of course limited, and far from a full coverage of the language found in mathematics books. Their development and maintenance has required considerable efforts. What we hope to show in this paper is that such interfaces are now easy to build, that they can be ported to other languages than English, and that their language can be made fairly sophisticated.

In Section 2, we will give a brief overview of the language of mathematics. In Section 3, we will introduce GF, Grammatical Framework [12,13], as a tool that enables the construction of translation systems with the minimum of effort. Section 4 defines a simple predicate logic interface, which, while satisfying the grammar rules of natural language, is easy to build and to port to different notations of formal logic and to different natural languages. In this interface, the formula

$$(\forall x)(\mathit{Nat}(x) \supset \mathit{Even}(x) \vee \mathit{Odd}(x))$$

gets translated to English, German, French, and Finnish in the following ways:

for all x, if x is a natural number then x is even or x is odd
für alle x, wenn x eine natürliche Zahl ist, dann ist x gerade oder x ist ungerade
pour tout x, si x est un nombre entier alors x est pair ou x est impair
kaikille x, jos x on luonnollinen luku niin x on parillinen tai x on pariton

In Section 5, we will increase the sophistication of the language by well-known techniques from linguistics and compiler construction. For instance, the above formula then gets the translations

every natural number is even or odd
jede natürliche Zahl ist gerade oder ungerade
tout nombre entier est pair ou impair
jokainen luonnollinen luku on parillinen tai pariton

and the translation still works in both directions between the formula and the sentences. In Section 6, we will discuss some problems that are either open or positively undecidable. Section 7 summarizes some natural language interfaces implemented in GF, and Section 8 concludes. The associated web page [14] contains the complete code referred to in this paper, as well as a live translation demo.

Throughout this paper, we will use the terms *easy* and *difficult* in a special way. *Easy* problems are ones that can be solved by well-known techniques; this doesn't mean that it was easy to develop these techniques in the first place. Thus the *easy* problems in natural language interfaces don't require training in GF or linguistics but can reuse existing components and libraries. For the *difficult* problems, no out-of-the-box solution exists. This relation between easy and difficult has parallels in all areas of technology. For instance, in automatic theorem proving itself, some classes of formulas are easy to decide with known techniques; some classes are easy for humans but still impossible for computers; and some classes will remain difficult forever. An important aspect of progress in both natural language processing and automatic reasoning has been to identify and extend the classes of easy problems, instead of getting paralyzed by the impossibility of the full problem.

2 The Language of Mathematics

What is the ideal language for interaction with proof systems? If we take mathematic books as the starting point, the answer is clear: it is a natural language, such as English or Polish or French, containing some mathematical formulas as parts of the text and structured by headers such as "definition" and "lemma". There can also be diagrams, linked to the text in intricate ways; for instance, a diagram showing a triangle may "bind" the variables used for its sides and angles in the text.

Ignoring the diagrams and the structuring headers for a moment, let us concentrate on the text parts. Mathematical texts consist of two kinds of elements, *verbal* (natural language words) and *symbolic* (mathematical formulas). The distribution of these elements is characterized by the following principles:

- Each sentence is a well-formed natural language sentence.
- A sentence may contain symbolic parts in the following roles:
  - noun phrases, as in x^2 *is divisible by* $\sqrt{x}$;
  - subsentences formed with certain predicates, as in *we conclude that* $x^2 > \sqrt{x}$.
- A symbolic part may not contain verbal parts (with some exceptions, for instance, the notation for set comprehension).

Of particular interest here is that logical constants are never (at least in traditional style) expressed by formulas. Also most logically atomic sentences are expressed by informal text, and so are many noun phrases corresponding to singular terms. The last rule may mandate the use of verbal expression even when symbolic notation exists. For instance, the sentence *the square of every odd number is odd* could not use the symbolic notation for the square, because it would then contain the verbal expression *every odd number* as its part. The conversion to symbolic notation is a complex procedure, as it involves the introduction of a variable: *for every odd number x,* x^2 *is odd*. For this example, the purely verbal expression is probably the better one.

3 GF in a Nutshell

3.1 Abstract and Concrete Syntax

GF is based on a distinction between *abstract syntax* and *concrete syntax*. An abstract syntax defines a system of *trees*, and a concrete syntax specifies how the trees are realized as *strings*. This distinction is only implicit in context-free (BNF, Backus-Naur Form) grammars. To give an example, the BNF rule for multiplication expressions

```
Exp ::= Exp "*" Exp
```

is in GF analysed into a pair of rules,

```
fun EMul : Exp -> Exp -> Exp
lin EMul x y = x ++ "*" ++ y
```

The first rule belongs to abstract syntax and defines a *function* `EMul` for constructing trees of the form `(EMul x y)`. The second rule belongs to concrete syntax and defines the *linearization* of trees of the forms `(EMul x y)`: it is the linearization of `x` concatenated (++) with the token `*` followed by the linearization of `y`.

GF grammars are *reversible*: they can be used both for linearizing trees into strings and for *parsing* strings into trees. They may be *ambiguous*: the string `x * y * z` results in two trees, `(EMul (EMul x y) z)` and `(EMul x (EMul y z))`.

3.2 Parametrization

Avoiding ambiguity may or may not be a goal for the grammar writer. In the rule above, a natural way to avoid ambiguity is to *parametrize* the linearization on *precedence*. Each expression has a precedence number, which can be compared to an expected number; parentheses are used if the given number is lower than the expected number. Rather than showing the low-level GF code for this, we use the GF library function `infixl` (left associative infix). Then we get the correct use of parentheses and the precedence level 2 by writing

```
lin EMul x y = infixl 2 "*" x y
```

Since GF is a functional programming language, we can be even more concise by using *partial application*,

```
lin EMul = infixl 2 "*"
```

This rule is similar to a precedence declaration in languages like Haskell [15],

```
infixl 2 *
```

Thus syntactic conventions such as precedences are a special case of GF's concrete syntax rules, which also cover parameters used in natural languages (see 3.3). What is important is that concrete syntax can be parametrized in different ways without changing the abstract syntax.

3.3 Multilinguality

The most important property of GF in most applications is its *multilinguality*: one abstract syntax can be equipped with many concrete syntaxes. Thus the function `EMul` can also be given the linearization

```
lin EMul x y = x ++ y ++ "imul"
```

which generates the reverse Polish notation of Java Virtual Machine. Now, combining the parsing of `x * y` with linearization into `x y imul` makes GF usable as a *compiler*. Since the number of concrete syntaxes attached to an abstract syntax is unlimited, and since all grammars are reversible, GF can be seen as a framework for building *multi-source multi-target compiler-decompilers*.

Yet another aspect of GF is the expressive power of its concrete syntax, reaching beyond the context-free: GF is equivalent to *Parallel Multiple Context-Free Grammars* (PMCFG, [16]). Therefore GF is capable of dealing with all the complexities of natural languages. Sometimes this extra power is not needed; for instance, an English rule for `EMul` can be simply written

```
lin EMul x y = "the product of" ++ x ++ "and" ++ y
```

But in German, the equivalent phrase has to *inflect* in different cases; in particular, the operands have to be in the dative case required by the preposition *von*. This is written

```
lin EMul x y = \\c => defArt Neutr c ++ Produkt_N ! c ++
                      "von" ++ x ! Dat ++ "und" ++ y ! Dat
```

where a case variable `c` is passed to the definite article (`defArt`), which in the neuter form produces *das*, *dem* or *des* as function of the case. The noun *Produkt* ("product") likewise depends on case.

Without going to the details of the notation, let alone German grammar, we notice that case belongs to the concrete syntax of German without affecting the abstract syntax. French has a different parameter system, involving gender (*le produit* vs. *la somme*) and article-preposition contractions (*le produit du produit*), etc. All these variations can be defined in GF, which enables the use of one and the same abstract syntax for a wide variety of languages.

3.4 Grammar Engineering

That the expressive power of GF is sufficient for multilingual grammars does not yet mean that it is *easy* to write them. The German `EMul` rule above is really the tip of an iceberg of the complexity found in natural languages. Writing formal grammars of natural language from scratch is *difficult* and laborious. To this end, a considerable part of the GF effort has gone to the development of *libraries* of grammars, which encapsulate the difficulties [17]. The *GF Resource Grammar Library*, RGL [18] is a collaborative project, which has produced implementations of 18 languages ranging from English, German, and French to Finnish, Arabic, and Urdu.

The RGL function `app` builds noun phrases (`NP`) as function applications with relational nouns (`N2`). It has several overloaded instances, for example,

```
app : N2 -> NP -> NP        -- the successor of x
app : N2 -> NP -> NP -> NP -- the sum of x and y
```

We use `app` to write the linearization rules of `EMul` for English, German, French, and Finnish as follows (hiding the variables by partial application):

```
lin EMul = app (mkN2 (mkN "product"))
lin EMul = app (mkN2 (mkN "Produkt" "Produkte" Neutr))
lin EMul = app (mkN2 (mkN "produit"))
lin EMul = app (mkN2 (mkN "tulo"))
```

In most of these cases, it is enough to write just the dictionary form of the word equivalent to *product*; the RGL function `mkN` infers all grammatical properties from this form. In German, however, the plural form and the gender need to be given separately. The syntactic construction is the same in all languages, using the function `app`, which yields equivalents of *the product of x and y*.

The RGL enables a *division of labour* between two kinds of grammar writers: *linguists*, who write the resource grammar libraries, and *application programmers*, who use the libraries for their specific purposes. An application programmer has an access to the RGL via its API (*Application Programmer's Interface*), which hides the linguistic complexity and details such as the passing of parameters. What the application programmer has to know is the vocabulary of her domain. For instance, the translation of *product* as *tulo* in Finnish is specific to mathematics; in many other contexts the translation is *tuote*. This kind of knowledge may be beyond the reach of the linguist, which shows that also the application programmer's knowledge makes an essential contribution to the quality of a translation system.

4 Baseline Translation for the Core Syntax of Logic

Let us start with a simple but complete *core syntax* of predicate logic, specified by the following table:

construction	**symbolic**	**verbal**
negation	$\sim P$	*it is not the case that P*
conjunction	$P \,\&\, Q$	*P and Q*
disjunction	$P \vee Q$	*P or Q*
implication	$P \supset Q$	*if P then Q*
universal quantification	$(\forall x)P$	*for all x, P*
existential quantification	$(\exists x)P$	*there exists an x such that P*

How to write the grammar in GF is described below, with full details for the abstract syntax (4.1) and the concrete syntax of natural language (4.2). The "verbal" column is generalized from English string templates to RGL structures, which work for all RGL languages. The formation of atomic sentences is described in 4.3, and Section 4.4 improves the grammar by eliminating the ambiguities of the verbalizations.

4.1 Abstract Syntax

The abstract syntax of predicate calculus can be written as follows:

```
cat Prop ; Ind ; Var
fun
  And, Or, If   : Prop -> Prop -> Prop
  Not           : Prop -> Prop
  Forall, Exist : Var  -> Prop -> Prop
  IVar          : Var  -> Ind
  VStr          : String -> Var
```

This abstract syntax introduces three *categories* (types of syntax trees); `Prop` (proposition), `Ind` (individual), and `Var` (variable); the category `String` is a built-in category of GF. It addresses pure predicate calculus. It can be extended with domain-specific functions. For instance, in arithmetic such functions may include the use of built-in integers (`Int`), addition and multiplication, and the predicates *natural number*, *even*, *odd*, and *equal*:

```
fun
  IInt           : Int -> Ind
  Add, Mul       : Ind -> Ind -> Ind
  Nat, Even, Odd : Ind -> Prop
  Equal          : Ind -> Ind -> Prop
```

Now the sentence

for all x, if x is a natural number then x is even or x is odd

can be expected to have the abstract syntax

```
Forall (VStr "x") (If (Nat (IVar (VStr "x")))
  (Or (Even (IVar (VStr "x"))) (Odd (IVar (VStr "x")))))
```

4.2 Concrete Syntax

When the RGL library is used, the first step in concrete syntax is to define the linguistic categories used for linearizing the categories of the application grammar. In the case at hand, propositions come out as sentences (`S`) and individuals and variables as noun phrases (`NP`). Thus we set

```
lincat Prop = S ; Ind, Var = NP
```

and can then define

```
lin
  And = mkS and_Conj
  Or  = mkS or_Conj
```

```
If p q = mkS (mkAdv if_Subj p) (mkS then_Adv q)
Not = negS
Forall x p = mkS (mkAdv for_Prep (mkNP all_Predet x)) p
Exist  x p = mkS (existS (mkNP x (mkRS p)))
IVar x = x
VStr s = symb s
```

We refer to the on-line RGL documentation for the details of the API functions used here. All of them are general-purpose functions readily available for all RGL languages, except `negS` and `existS`, which are constructed by using the RGL in different ways in different languages. Thus a straightforward implementation for the negation in English is *it is not the case that P*, obtained by

```
negS p = mkS negative_Pol
  (mkCl it_NP (mkNP the_Det (mkCN (mkN "case") p)))
```

and similarly for German (*es ist nicht der Fall, dass P*) and French (*il n'est pas le cas que P*). This form of negation works for all kinds of propositions, complex and atomic alike. We will show later how to optimize this for atomic propositions and produce *x is not odd* instead of *it is not the case that x is odd*. Existence can likewise be straightforwardly linearized as *there exists x such that P* and its equivalents.

At the other end of the translation, we need a grammar for the symbolic notation of logic. By using precedences in the way shown in Section 3.2, this is straighforward for some notations; some others, however, may use devices such as lists (many-place conjunctions rather than binary ones) and domain-restricted quantifiers. We will return to these variations in Section 5.

4.3 The Lexicon

To extend the concrete syntax to arithmetic constants, we can write

```
lin
  IInt  = symb
  Add   = app  (mkN2 (mkN "sum"))
  Mul   = app  (mkN2 (mkN "product"))
  Nat   = pred (mkCN (mkA "natural") (mkN "number"))
  Even  = pred (mkA "even")
  Odd   = pred (mkA "odd")
  Equal = pred (mkA "equal")
```

and similarly in other languages, varying the adjectives and nouns used. A particularly useful RGL function for this purpose is `pred`, which is an overloaded function covering different kinds of predication with adjectives, nouns, and verbs:

```
pred : A  -> NP -> S        -- x is even
pred : A  -> NP -> NP -> S  -- x and y are equal
pred : CN -> NP -> S        -- x is a number
pred : V  -> NP -> S        -- x converges
pred : V2 -> NP -> NP -> S  -- x includes y
```

Since the main part of work in natural language interfaces has to do with the non-logical vocabulary, the main devices needed by most programmers will be the functions `app` (Section 3.4) and `pred`, as well as lexical functions such as `mkN` and `mkA`.

The lexical work can be further reduced by using a general-purpose multilingual mathematical lexicon. Such a lexicon was built for six languages within the WebALT project [19] to cover the content lexicons in the OpenMath project [20]. The WebALT lexicon is extended and ported to more languages in the MOLTO project [21].

4.4 Ambiguity

The baseline grammar of predicate logic has a narrow coverage and produces clumsy language. Its worst property, however, is that the language is ambiguous. As there is no concept of precedence, we have the following ambiguities:

P and Q or R :	$(P\&Q) \vee R$	vs.	$P\&(Q \vee R)$
it is not the case that P and Q :	$(\sim P)\&Q$	vs.	$\sim (P\&Q)$
for all x, P and Q :	$((\forall x)P)\&Q$	vs.	$(\forall x)(P\&Q)$

Introducing a precedence order by stipulation would not solve the problem. For instance, stipulating that *and* binds stronger than *or* would simply make $P\&(Q \vee R)$ inexpressible! Moreover, stipulations like this would cause the language no longer to be a real fragment of natural language, since the user would have to learn the artificial precedence rules separately in order really to understand the language.

The ambiguity problem will be revisited in Section 5, using techniques from natural language generation (NLG) to reduce ambiguities in optimized ways. However, even in the simple interface we can solve the problem by parametrizing the concrete syntax. We just need a Boolean parameter that indicates whether a proposition is complex (i.e. formed by a connective). Such parameters can in GF be attached to expressions by using *records*. Thus the linearization type of propositions becomes

```
lincat Prop = {s : S ; isCompl : Bool}
```

Following an idea from [22], we can use bulleted lists to structure sentences—a device that is more natural-language-like than parentheses would be. Thus we have the following unambiguous variants of *P and Q or R*:

$(P\&Q) \vee R$	$P\&(Q \vee R)$
either of the following holds:	both of the following hold:
• *P* and *Q*	• *P*
• *R*	• *Q* or *R*

The rule for generating this says that, if *none* of the operands is complex, the sentence conjunction can be used; otherwise, the bulleted structure must be used:

```
lin And p q = case <p.isCompl, q.isCompl> of {
  <False,False>
    => {s = mkS and_Conj p.s and q.s ; isCompl = True} ;
  _ => {s = bulletS Pl "both" p.s q.s ; isCompl = False}
  }
```

Unfortunately, it is not decidable whether a GF grammar (or even a context-free grammar) is ambiguous. Working on the high abstraction level of RGL can make it difficult even to see the effect of individual rules. Therefore, the only certain procedure is to test with the parser. When generating natural language from logic, the parser test can be applied to a set of alternative equivalent expressions to select, for instance, the shortest unambiguous one. The next section will show how such equivalent expressions are generated.

In general, syntactic disambiguation may be based on semantic considerations, which makes it *difficult* (Section 6.1). But one semantic method is *easy* to implement for fragment at hand: *binding analysis*. Thus, when parsing the sentence *for all x, x is even or x is odd*, the interpretation $((\forall x)\mathit{Even}(x)) \vee \mathit{Odd}(x)$ can be excluded, because it contains an unbound variable. A general and powerful approach to semantics-based disambiguation uses *dependent types* and *higher-order abstract syntax* ([13], Chapter 6). For instance, the universal quantifier can be declared

```
fun Univ : (A : Dom) -> (Var A -> Prop) -> Prop
```

The body of quantification now depends on a variable, which moreover is typed with respect to a domain. In this way, both binding analysis and the well-typedness of predicate applications with respect to domains is defined in GF, in a declarative way. However, this method is *difficult* (though not impossible) to scale up to the syntax extensions discussed in the next section.

5 Beyond the Baseline Translations: Easy Improvements

In this section, we will extend the abstract syntax of logic with structures available in natural language but not in standard predicate logic (Section 5.1). This extended syntax is still a conservative extension of the core syntax and can easily be translated to it (5.2). The most challenging part is the reverse: given a core syntax tree, find the best tree in the extended syntax. In addition to better style, the extended syntax provides new ways to eliminate ambiguity (5.3). Finally, Section 5.4 will show how to optimally divide the expressions into verbal and symbolic parts, as specified in Section 2.

For reasons of space and readability, we will no longer give the explicit GF code. Instead, we use a logical formalism extended with constructs that correspond to the desired natural language structures. At this point, and certainly with help from GF documentation, the reader should be able easily to reconstruct the GF code. The full code is shown on the associated web page [14], including a concrete syntax using RGL, and the conversions of Sections 5.2 and 5.3.

The core-to-extension conversions of Section 5.3 could not be implemented as linearization rules of core syntax. The reason is *compositionality*: every linearization rule is a mapping * such that

$$(f\, t_1 \ldots t_n)^* \;=\; h\, t_1^* \ldots t_n^*$$

where the function *h* operates on the *linearizations* of the immediate subtrees t_i. Thus it cannot analyse the subtrees t_i, as the conversions of Section 5.3 have to do. When two languages are related by an abstract syntax and compositional linearizations, they

are in a part-to-part correspondance; this is no longer true for core syntax formulas and their optimal natural language expressions.

Even though the conversions cannot be defined as linearizations, they could in principle be written in GF as *semantic actions* ([13], Chapter 6). But this would require advanced GF hackery and therefore be *difficult*; GF is a special-purpose language designed for multilingual grammars, and lacks the program constructs and libraries needed for non-compositional translations, such as list processing, state management, and so on. The conversions in [14] are therefore written in Haskell. This is an illustration of the technique of *embedded grammars*, where a GF grammars can be combined with host language programs ([13], Chapter 7). Thus the overall system written in Haskell provides linearization and parsing via the GF grammar, and abstract syntax trees can be manipulated as Haskell data objects. In particular, the technique of *almost compositional functions* [23] is available, which makes it *easy* for Haskell programmers to implement conversions such as the ones in Section 5.3. The same technique is available for Java as well, and provides the *easy* way for Java programmers to use GF for building natural language interfaces to Java programs.

5.1 Extended Abstract Syntax

The core logical language addressed in Section 4 has a minimal set of categories and one function for each logical constant. The extended language has a more fine-grained structure. The following table gives the extensions in symbolic logic and English examples:

construction	**symbolic**	**verbal (example)**
atom negation	$\overline{A}$	*x is not even*
conjunction of proposition list	$\&[P_1, \ldots, P_n]$	*P, Q and R*
conjunction of predicate list	$\&[F_1, \ldots, F_n]$	*even and odd*
conjunction of term list	$\&[a_1, \ldots, a_n]$	*x and y*
bounded quantification	$(\forall\, x_1, \ldots, x_n : K)P$	*for all numbers x and y, P*
in-situ quantification	$F(\forall\, K)$	*every number is even*
one-place predication	$F^1(x)$	*x is even*
two-place predication	$F^2(x, y)$	*x is equal to y*
reflexive predication	$\mathit{Refl}(F^2)(x)$	*x is equal to itself*
modified predicate	$\mathit{Mod}(K, F)(x)$	*x is an even number*

All constructs with $\&$ also have a variant for $\vee$, and so have the constructs with $\forall$ for $\exists$.

The new forms of expression involve some new categories. Obviously, we need categories of *lists* of propositions, predicates, variables, and individual terms. But we also introduce separate categories of one- and two-place predicates, and it is useful to have a separate category of atomic propositions.

It is moreover useful to distinguish a category of *kind predicates*, typically used for restricting the domain of quantification (the variable *K* in the table above). For instance, *natural number* is a kind predicate, as opposed to *odd*, which is an "ordinary" predicate. But we will also allow the use of *K* in a predication position, to say *x is a natural number*. Modified predicates combine a kind predicate with another predicate

into a new kind predicate. Very typically, kind predicates are expressed with nouns and other one-place predicates with adjectives (cf. [9]); but this need not be assumed for all predicates and all languages.

The new syntax is a proper extension of the core syntax of Section 4. It is, in particular, not necessary to force all concepts into the categories of one- or two-place predicates: they can still be expressed by "raw" propositional functions. But reclassifying the arithmetic lexicon with the new categories will give opportunities for better language generation via the extended syntax. Thus *Even* and *Odd* can be classified as one-place predicates, *Equal* as a two-place predicate, and *Nat* as a kind predicate. We can then obtain the sentence

every natural number is even or odd

as a compositional translation of the formula

$$\vee[\mathit{Even}, \mathit{Odd}](\forall\, \mathit{Nat})$$

It remains to see how this formula is converted to a core syntax formula (easy), and how it can be obtained as an optimization of the core formula (more tricky).

5.2 From Extended Syntax to Core Syntax

Mapping the extended syntax into the core syntax can be seen as *denotational semantics*, where the core syntax works as a *model* of the extended syntax. The semantics follows the ideas of Montague [24], which focused particularly on in-situ quantification. The question of quantification has indeed been central in linguistic semantics (see e.g. [25]); what we use here is a small, *easy* part of the potential, carefully avoiding usages that lead to ambiguities and other *difficult* problems.

The crucial rule is in-situ quantification. It requires that trees of type `Ind` are interpreted, not as individuals but as quantifiers, that is, as functions from propositional functions to propositions. Thus the type of the interpretation is

$$(\forall\, K)^* : (\mathit{Ind} \rightarrow \mathit{Prop}) \rightarrow \mathit{Prop}$$

The interpretation is defined by specifying how this function applies to a propositional function *F* (which need not be an atomic predicate):

$$(\forall\, K)^* F \;=\; (\forall\, x : K^*)(F\, x)$$

The rule introduces a bound variable *x*, which must be fresh in the context in which the rule is applied. Notice that we define the result as a proposition still in the extended syntax. It can be processed further by moving the kind *K* to the body, using the rule

$$((\forall\, x_1, \ldots, x_n : K)P)^* \;=\; (\forall\, x_1) \cdots (\forall\, x_n)((K^* x_1)\,\&\,\ldots\,\&\,(K^* x_n) \;\supset\; P^*)$$

But the intermediate stage is useful if the target logic formalism supports domain-restricted quantifiers. In general, various constructs of the extended syntax are available in extensions of predicate logic, such as TFF and THF [26].

Conjunctions of individuals are likewise interpreted as functions on propositional functions,

$$\&[a_1, \ldots, a_n]^*\, F \;=\; \&[a_1^*\, F, \ldots, a_n^*\, F]$$

The interpretation of predication is "reversed": now it is the argument that is applied to the predicate, rather then the other way round (interestingly, this shift of point of view was already known to Frege [27], §10: "one can conceive $\Phi(A)$ as a function of the argument Φ"!). Two-place predication requires lambda abstraction.

$$(F(a))^* \;=\; a^* F^*$$
$$(F(a,b))^* \;=\; a^*((\lambda x) b^*((\lambda y)(F^* x\, y)))$$

Atomic predicates, including simple kind predicates, can be interpreted as themselves, whereas the conjunction of predicates is a propositional function forming a conjunction of propositions:

$$\&[F_1, \ldots, F_n]^*\, x \;=\; \&[(F_1^* x), \ldots, (F_n^* x)]$$

Reflexive predicates expand to repeated application, whereas modified kind predicates are interpreted as conjunctions:

$$(\mathit{Refl}(F))^* x \;=\; F^* x\, x$$
$$(\mathit{Mod}(K, F))^*\, x \;=\; (K^* x) \& (F^* x)$$

The elimination of list conjunctions is simple folding with the binary conjunction:

$$\&[P_1, \ldots, P_n]^* \;=\; P_1^* \& \ldots \& P_n^*$$

Now we can satisfy one direction of the desired conversion: we can parse *every natural number is even or odd* and obtain the formula

$$\vee[\mathit{Even}, \mathit{Odd}](\forall\, \mathit{Nat})$$

whose interpretation in the core syntax is

$$(\forall x)(\mathit{Nat}(x) \;\supset\; \mathit{Even}(x) \vee \mathit{Odd}(x))$$

whose compositional translation is *for all x, if x is a natural number then x is even or x is odd*.

5.3 From Core Syntax to Extended Syntax

Finding extended syntax equivalents for core syntax trees is trickier than the opposite direction. It is a problem of *optimization*: find the "best" possible tree to express the same proposition as the original. Now, the "sameness" of propositions is defined by the interpretation shown in Section 5.2. But what does the "best" mean? Let us consider some conversions that clearly improve the proposition.

1. **Flattening**. Nested binary conjunctions can be flattened to lists, to the effect that *P and Q and R* becomes *P, Q and R*. This has many good effects: a syntactic ambiguity is eliminated; bullet lists become arbitrarily long and thus more natural; and opportunities are created for the next operation, aggregation.

2. **Aggregation**. This is a standard technique from NLG [28]. For the task at hand, its main usage is to share common parts of predications, for instance, to convert *x is even or x is odd* to *x is even or odd*. Thus the subject-sharing aggregation rule has the effect

$$\&[F_1(a), \ldots, F_n(a)] \implies \&[F_1, \ldots, F_n](a)$$

The predicate-sharing aggregation rule is, dually,

$$\&[F(a_1), \ldots, F(a_n)] \implies F(\&[a_1, \ldots, a_n])$$

Aggregation can be further strengthened by sorting the conjuncts by the predicate or the argument, and then grouping maximally long segments. Notice that aggregation reduces ambiguity: *x is even or x is odd and y is odd* is ambiguous, but the two readings are captured by *x is even or odd and y is odd* and *x is even or x and y are odd*.

3. **In-situ quantification**. The schematic rule is to find an occurrence of the bound variable in the body of the sentence and replace it with a quantifier phrase:

$$(\forall x : K)P \implies P((\forall K)/x)$$

If the starting point is a formula with domainless quantification, the pattern must first be found by the rules

$$(\forall x)(K(x) \supset P) \implies (\forall x : K)P$$
$$(\exists x)(K(x)\&P) \implies (\exists x : K)P$$

The in-situ rule is restricted to cases where *P* is atomic and has exactly one occurrence of the variable. Thus aggregation can create an opportunity for in-situ quantification:

for all natural numbers x, x is even or x is odd
$\implies$ *for all natural numbers x, x is even or odd*
$\implies$ *every natural number is even or odd*

This chain of steps shows why in-situ quantification is restricted to propositions with a single occurrence of the variable: perfoming it *before* aggregation would form the sentence *every number is even or every number is odd*, which has a different meaning.

4. **Verb negation**,

$$\sim A \implies \overline{A}$$

This has the effect of transforming *it is not the case that x is even* to *x is not even*. Verb negation is problematic if in-situ quantifiers are present, since for instance *every natural number is not even* has two parses, where either the quantifier or the negation has wider scope. If in-situ quantification is restricted to atomic formulas, the only interpretation is $\sim (\forall x : \mathit{Nat})\mathit{Even}(x)$. But relying on this is better avoided altogether, to remain on the safe side—and thus create verb negation only for formulas without in-situ quantifiers.

5. **Reflexivization**,

$$F(x, x) \implies \mathit{Refl}(F)(x)$$

which has the effect of converting *x is equal to x* to *x is equal to itself*. This can again create an opportunity for in-situ quantification: *every natural number is equal to itself*.

6. **Modification**, combining a kind and a modifying predicate into a complex kind predicate

$$K(x)\&F(x) \implies \mathit{Mod}(K,F)(x)$$

which has the effect of converting *x is a number and x is even* to *x is an even number*. As the number of occurrences of *x* is reduced, there is a new opportunity for in-situ quantification: *some even number is prime*.

5.4 Verbal vs. Symbolic

In Section 2, we suggested that symbolic expressions are preferred in mathematical text, whenever available. This creates a tension with the conversions of Section 5.3, which try to minimize the use of symbols. But there is no contradiction here: the conversions minimize the use of symbols for expressing the *logical structure*, whereas Section 2 suggests it should be maximized in *atomic formulas*. The symbols needed for logical structure are the bound variables, which are replaced by in-situ quantification whenever possible without introducing ambiguity.

Hence, the strategy is to perform the conversions of 5.3 first, and then express symbolically whatever is possible. Here are two examples of the two-step procedure, leading to different outcomes:

for all x, if x is a number and x is odd, then x^2 *is odd*
$\implies$ (*every odd number*)^2 *is odd* (**incorrect!**)
$\implies$ *the square of every odd number is odd*

for all x, if x is a number and x is odd, then the sum of x and the square of x is even
$\implies$ *for all odd numbers x, the sum of x and the square of x is even*
$\implies$ *for all odd numbers x,* $x + x^2$ *is even*

The latter could be improved by using pronouns to eliminate the variable: *the sum of every odd number and its square is even*. However, the analysis and synthesis of (non-reflexive) pronouns belongs to the problems we still consider *difficult*.

The choice between verbal and symbolic expressions is easy to implement, since it can be performed by linearization. All that is needed is a Boolean parameter saying whether an expression is symbolic. If all arguments of a function are symbolic, the application of the function to them can be rendered symbolically; if not, the verbal expression is chosen. For example, using strings rather than RGL to make the idea explicit, the square function becomes

```
lin Square x = {
  s = case x.isSymbolic of {
        True  => x.s ++ "^2" ;
        False => "the square of" ++ x.s
        } ;
  isSymbolic = x.isSymbolic
  }
```

Thus the feature of being symbolic is inherited from the argument. Variables and integers are symbolic, whereas in-situ quantifiers are not.

6 The Limits of Known Techniques

6.1 The Dynamicity of Language

The Holy Grail of theorem proving in natural language is a system able to formalize any mathematical text automatically. In one sense, this goal has already been reached. The Boxer system [29] is able to parse *any* English sentence and translate it into a formula of predicate calculus. Combined with theorem proving and model checking, Boxer can moreover solve problems in open-domain textual entailment by formal reasoning [30].

The problem that remains with Boxer is that the quality of formalization and reasoning is far from perfect. It is useful for information retrieval that goes beyond string matching, but not for the precision task of checking mathematical proofs. While the retrieval aspect of reasoning is an important topic, no-one seriously claims that the technique would reach the precision needed for mathematics. What we have here is the classical trade-off in natural language processing: we cannot maximize both *coverage* and *precision* at the same time, but have to choose.

Our target in this paper has obviously been to maintain precision while extending the coverage step by step. What is more, our perspective has mainly been that of *generation*: we have started from an abstract syntax of predicate calculus and seen how it is reflected in natural language. The opposite perspective of *analysis* is only implicit, via the reversibility of GF grammars. If our grammar does manage to parse a sentence from a real text, it happens more as a lucky coincidence than by design. (This is not so bad as it might sound, since parsing in GF is *predictive*, which means that the user input is guided by word suggestions and completions; see [31].)

The task of formalizing real mathematical texts is thoroughly analysed by Ganesalingam [32]. With samples of real texts as starting point, he shows that mathematical language is not only complex but also *ambiguous* and *dynamically changing*. Even things like operator precedence can be ambiguous. For instance, addition binds stronger than equality in formulas like $2 + 2 = 4$. But [32] cites the expression

$$\lambda + K = S$$

which in [33] stands for the theory λ enriched with the axiom $K = S$, and should hence be parsed $\lambda + (K = S)$. The solution to this ambiguity is to look at the *types* of the expression. The $+$ and $=$ operators are *overloaded*: they work for different types, including numbers and theories. Not only does their semantics depend on the type, which is common in programming languages, but also their syntactic properties.

Because of the intertwining of parsing and type checking, the usual pipe-lined techniques are insufficient for automatically formalizing arbitrary mathematical texts. This is nothing new for natural language: overload resolution is, basically, just an instance of *word sense disambiguation*, which is needed in tasks like information retrieval and machine translation. One thing shown in [32] is that, despite its believed exactness, the informal language of mathematics inherits many of the general problems of natural language processing.

As suggested in Section 4.4, parsing intertwined with type checking can be theoretically understood via the use of *dependent types* (cf. also [34,35]). It is also supported by GF. But it is not yet a piece of technology that can be called *easy*; building a natural

language interface that uses GF's dependent types is still a research project. [36] and [37] are pioneering examples. They also use dependent types to deal with pronouns and definite descriptions, as suggested in [34].

6.2 The Structure of Proof Texts

Given that the analysis of arbitrary mathematical text is beyond the reach of current technology, how is it with generation? The task of *text generation from formal proofs* has a great potential, for instance, in industry, where theorem provers are used for verifying large systems. The users are not always happy with just the answer "yes" or "no", but want to understand why. [38] was a pioneering work in generating text from Coq proofs. Even though it worked reasonably for small proofs, the text generated from larger proofs turned out to be practically unreadable.

The problems in proof generation are analogous to the problems in sentence generation: the structure has to be changed by techniques such as aggregation (Section 5.3). Some such techniques, inspired by code generation in compilers, were developed in [39]. But there is also another aspect: the hiding of uninteresting steps. Natural proofs typically take much longer steps than formalized proofs. Thus we also have the question which steps to show. [40] suggests a promising approach, trying to identify the decisive branching points in proofs as the ones worth showing. This idea resembles the methods of *dataflow analysis* in compiler construction [41], in particular the technique of *basic blocks*. But, in analogy to the "full employment theorem for compiler writers", which Appel [41] attributes to Rice [42], it seems that text generation involves undecidable optimization problems that have no ultimate automatic solution.

Even though the techniques shown in this paper don't scale up to proof texts, they do easily extend beyond individual propositions. [43] shows that texts consisting of definition and theorem block can be made natural and readable. [36] likewise succeeds with software specifications where the texts describe invariants and pre- and postconditions.

7 Projects

GF was first released in 1998, in a project entitled "Multilingual Document Authoring" at Xerox Research Centre Europe in Grenoble [44]. The project built on earlier work on natural language proof editors in [45,46]. The earlier work had focused on the multilingual rendering of formal mathematics, based on constructive type theory [47] and the ALF system [48]. The Xerox project shifted the focus from mathematics to more "layman" applications such as user manuals, tourist phrasebooks, and spoken dialogue systems (see [13] for a survey). But the translation between logic and language has been a recurring theme in GF projects. Here are some of them:

– **Alfa**, a type-theoretical proof editor was equipped with a GF grammar and a system of annotations for defining a multilingual mathematical lexicon [43]. The system was aimed to be portable to different applications, due to the generality of Alfa itself. As an early design choice, dependent types were not employed in GF, but type checking was left to Alfa. The lack of resource grammars made it difficult to define the translations of new concepts.

- **KeY**, a software verification system [49] was equipped with a translation of specifications between OCL (Object Constraint Language, [50]), English, and German [36]. Dependent types were used to guide the author to write meaningful specifications. This was the first large-scale application of the resource grammar library.
- **FraCaS** [51], a test suite of textual entailment in English, was parsed with an extension of RGL, translated to the TPTP format, and fed to automated reasoning tools [52].
- **WebALT** (Web Advanced Learning Technology), a European project aiming to build a repository of multilingual math exercises [19] using formalizations from the OpenMath project [20]. The work is continued in the MOLTO project [21], and the lexicon is used in the web demo of this article [14]
- **Attempto** (Attempto Controlled English), a natural language fragment used for knowledge representation and reasoning [53]. The fragment was implemented in GF and ported to five other languages using the RGL [54].
- **OWL** (Web Ontology Language, [55]), interfaced with English and Latvian via Attempto [53], but without a previously existing Latvian RGL [56].
- **SUMO** (Suggested Upper Merged Ontology, [57]). This large knowledge base and lexicon was reverse-engineered in GF, with improved natural language generation for three languages using RGL [37]. The abstract syntax uses dependent types to express the semantics of SUMO.
- **MathNat**. An educational proof system implemented in GF and linked to theorem proving in the TPTP format [58].
- **Nomic**. A computer game where new rules can be defined by the players in English [59]. Aimed as a pilot study for a controlled natural language for contracts.
- **MOLTO KRI** (Knowledge Representation Infrastructure). A query language with a back end in ontology-based reasoning [60].

8 Conclusion

Accumulated experience, the growth of the resource grammar libraries, and the improvement of tools make it *easy* to build translators between formal and informal languages in GF. The translators can produce reasonably understandable language and are portable to all languages available in the grammar library (currently 18). The effort can be brought down to the level of a few days' engineering or an undergraduate project; still some years ago, it was a research project requiring at least a PhD student.

The scope for improvements is endless. Parsing natural language is still restricted to very small fragments. Increasing the coverage is not just the matter of writing bigger grammars, but new ideas are needed for disambiguation. Generating natural language, especially from complex formal proofs, still tends to produce texts that are unreadable in spite of being grammatically correct. New ideas are needed even here.

Acknowledgements I am grateful to Wolfgang Ahrendt, Olga Caprotti, Ramona Enache, Reiner Hähnle, Thomas Hallgren, and Jordi Saludes for suggestions and comments. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n:o FP7-ICT-247914.

References

1. Wolfram Research, I.: Mathematica Homepage. http://www.wolfram.com/products/mathematica/ (2000)
2. Bobrow, D.G.: Natural Language Input for a Computer Problem Solving System. PhD thesis, Massachusetts Institute of Technology (1964)
3. de Bruijn, N.G.: Mathematical Vernacular: a Language for Mathematics with Typed Sets. In Nederpelt, R., ed.: Selected Papers on Automath. North-Holland Publishing Company (1994) 865–935
4. Trybulec, A.: The Mizar Homepage (2006) http://mizar.org/.
5. Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, Kerber, M., Kohlhase, M., Konrad, K., Melis, E., Meier, A., Schaarschmidt, W., Siekmann, J., Sorge, V.: Omega: Towards a mathematical assistant. In: Proceedings of the 14th Conference on Automated Deduction, Springer (1997)
6. Wenzel, M.: Isar - a generic interpretative approach to readable formal proof documents. In Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L., eds.: TPHOLs. Volume 1690 of LNCS., Springer (1999) 167–184
7. Zinn, C.: Understanding Informal Mathematical Discourse. PhD thesis, Department of Computer Science, University of Erlangen-Nürnberg (2010)
8. Buchberger, B., Craciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M.: Theorema: Towards computer-aided mathematical theory exploration. J. Applied Logic **4**(4) (2006) 470–504
9. Kamareddine, F., Wells, J.B.: Computerizing Mathematical Text with MathLang. Electr. Notes Theor. Comput. Sci. **205** (2008) 5–30
10. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts. In Fuchs, N.E., ed.: CNL. Volume 5972 of LNCS., Springer (2009) 170–186
11. Neumaier, A.: FMathL - Formal Mathematical Language (2009) http://www.mat.univie.ac.at/~neum/FMathL.html.
12. Ranta, A.: Grammatical Framework: A Type-Theoretical Grammar Formalism. The Journal of Functional Programming **14(2)** (2004) 145–189 http://www.cse.chalmers.se/~aarne/articles/gf-jfp.pdf.
13. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford (2011) ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).
14. Ranta, A.: Grammatical Framework: A Hands-On Introduction. CADE-23 Tutorial, Wroclaw (2011) http://www.grammaticalframework.org/gf-cade-2011/.
15. Peyton Jones, S.: Haskell 98 language and libraries: the Revised Report (2003) http://www.haskell.org/haskellwiki/Language_and_library_specification.
16. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science **88** (1991) 191–229
17. Ranta, A.: Grammars as Software Libraries. In Bertot, Y., Huet, G., Lévy, J.J., Plotkin, G., eds.: From Semantics to Computer Science. Essays in Honour of Gilles Kahn, Cambridge University Press (2009) 281–308 http://www.cse.chalmers.se/~aarne/articles/libraries-kahn.pdf.
18. Ranta, A.: The GF Resource Grammar Library. Linguistics in Language Technology **2** (2009) http://elanguage.net/journals/index.php/lilt/article/viewFile/214/158.
19. Caprotti, O.: WebALT! Deliver Mathematics Everywhere. In: Proceedings of SITE 2006. Orlando March 20-24. (2006) http://webalt.math.helsinki.fi/content/e16/e301/e512/PosterDemoWebALT_eng.pdf.

20. Abbott, J., Díaz, A., Sutor, R.S.: A report on OpenMath: a protocol for the exchange of mathematical information. SIGSAM Bull. **30** (March 1996) 21–24
21. Saludes, J., Xambó, S.: MOLTO Mathematical Grammar Library (2010) http://www.molto-project.eu/node/1246.
22. Burke, D.A., Johannisson, K.: Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In P. Blache and E. Stabler and J. Busquets and R. Moot, ed.: Logical Aspects of Computational Linguistics (LACL 2005). Volume 3492 of LNCS/LNAI., Springer (2005) 51–66 http://www.springerlink.com/content/?k=LNCS+3492.
23. Bringert, B., Ranta, A.: A Pattern for Almost Compositional Functions. The Journal of Functional Programming **18(5–6)** (2008) 567–598
24. Montague, R.: Formal Philosophy. Yale University Press, New Haven (1974) Collected papers edited by Richmond Thomason.
25. Barwise, J., Cooper, R.: Generalized quantifiers and natural language. Linguistics and Philosophy **4**(2) (1981) 159–219
26. Sutcliffe, G., Benzmüller, C.: Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. Journal of Formalized Reasoning **3** (2010)
27. Frege, G.: Begriffsschrift. Louis Nebert, Halle A/S (1879)
28. Reiter, E., Dale, R.: Building Natural Language Generation Systems. Cambridge University Press (2000)
29. Bos, J., Clark, S., Steedman, M., Curran, J.R., Hockenmaier, J.: Wide-Coverage Semantic Representations from a CCG Parser. In: Proceedings of the 20th International Conference on Computational Linguistics (COLING '04), Geneva, Switzerland (2004) 1240–1246
30. Bos, J., Markert, K.: Recognising Textual Entailment with Robust Logical Inference. In Candela, J.Q., Dagan, I., Magnini, B., d'Alché Buc, F., eds.: MLCW. Volume 3944 of LNCS., Springer (2005) 404–426
31. Angelov, K.: Incremental Parsing with Parallel Multiple Context-Free Grammars. In: Proceedings of EACL'09, Athens. (2009)
32. Ganesalingam, M.: The Language of Mathematics. PhD thesis, Department of Computer Science, University of Cambridge (2010) http://people.pwf.cam.ac.uk/mg262/.
33. Barendregt, H.: The Lambda Calculus. Its Syntax and Semantics. North-Holland (1981)
34. Ranta, A.: Type Theoretical Grammar. Oxford University Press (1994)
35. Ljunglöf, P.: The Expressivity and Complexity of Grammatical Framework. PhD thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University (2004) http://www.cs.chalmers.se/~peb/pubs/p04-PhD-thesis.pdf.
36. Johannisson, K.: Formal and Informal Software Specifications. PhD thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University (2005)
37. Angelov, K., Enache, R.: Typeful Ontologies with Direct Multilingual Verbalization. In Fuchs, N., Rosner, M., eds.: CNL 2010, Controlled Natural Language. (2010)
38. Coscoy, Y., Kahn, G., Thery, L.: Extracting text from proofs. In Dezani-Ciancaglini, M., Plotkin, G., eds.: Proc. Second Int. Conf. on Typed Lambda Calculi and Applications. Volume 902 of LNCS. (1995) 109–123
39. Coscoy, Y.: Explication textuelle de preuves pour le calcul des constructions inductives. PhD thesis, Université de Nice-Sophia-Antipolis (2000)
40. Wiedijk, F.: Formal Proof Sketches. In: Types for Proofs and Programs. LNCS 3085, Springer (2004) 378–393
41. Appel, A.: Modern Compiler Implementation in ML. Cambridge University Press (1998)
42. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2) (1953) pp. 358–366

43. Hallgren, T., Ranta, A.: An Extensible Proof Text Editor. In Parigot, M., Voronkov, A., eds.: LPAR-2000. Volume 1955 of LNCS/LNAI., Springer (2000) 70–84 http://www.cse.chalmers.se/~aarne/articles/lpar2000.pdf.
44. Dymetman, M., Lux, V., Ranta, A.: XML and multilingual document authoring: Convergent trends. In: Proc. Computational Linguistics COLING, Saarbrücken, Germany, International Committee on Computational Linguistics (2000) 243–249
45. Ranta, A.: Context-relative syntactic categories and the formalization of mathematical text. In Berardi, S., Coppo, M., eds.: Selected papers from TYPES'95: Int. Workshop on Types for Proofs and Programs, Trento, Italy. Volume 1158 of LNCS., Springer-Verlag (1996) 231–248
46. Ranta, A.: Structures grammaticales dans le français mathématique. Mathématiques, informatique et Sciences Humaines **138/139** (1997) 5–56/5–36
47. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis, Napoli (1984)
48. Magnusson, L., Nordström, B.: The ALF proof editor and its proof engine. In: Types for Proofs and Programs. LNCS, Nijmegen, Springer-Verlag (1994) 213–237
49. Beckert, B., Hähnle, R., Schmitt, P.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer-Verlag (2006)
50. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modelling with UML. Addison-Wesley (1999)
51. Kamp, H., Crouch, R., van Genabith, J., Cooper, R., Poesio, M., van Eijck, J., Jaspars, J., Pinkal, M., Vestre, E., Pulman, S.: Specification of linguistic coverage (1994) FRACAS Deliverable D2.
52. Bringert, B.: Semantics of the GF Resource Grammar Library. Report, Chalmers University (2008) http://www.cse.chalmers.se/alumni/bringert/darcs/mosg/.
53. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto Controlled English for Knowledge Representation. In Baroglio, C., Bonatti, P.A., Małuszyński, J., Marchiori, M., Polleres, A., Schaffert, S., eds.: Reasoning Web, Fourth International Summer School 2008. Number 5224 in LNCS, Springer (2008) 104–124
54. Ranta, A., Angelov, K.: Implementing Controlled Languages in GF. In: Proceedings of CNL-2009, Athens. Volume 5972 of LNCS. (2010) 82–101
55. Dean, M., Schreiber, G.: OWL Web Ontology Language Reference (2004) http://www.w3.org/TR/owl-ref/.
56. Gruzitis, N., Barzdins, G.: Towards a More Natural Multilingual Controlled Language Interface to OWL. In: 9th International Conference on Computational Semantics (IWCS). (2011) 335–339 http://www.aclweb.org/anthology/W/W11/W11-0138.pdf.
57. Niles, I., Pease, A.: Towards a standard upper ontology. In: Proceedings of the international conference on Formal Ontology in Information Systems - Volume 2001. FOIS '01, New York, NY, USA, ACM (2001) 2–9
58. Humayoun, M., Raffalli, C.: MathNat - Mathematical Text in a Controlled Natural Language. Journal on Research in Computing Science **66** (2010)
59. Camilleri, J.J., Pace, G.J., Rosner, M. In: Playing Nomic Using a Controlled Natural Language. (2010) http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-622/paper7.pdf.
60. Mitankin, P., Ilchev, A.: Knowledge Representation Infrastructure (2010) MOLTO Deliverable D4.1. http://www.molto-project.eu/sites/default/files/D4.1_0.pdf.