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

CADE-23, Wrocław, 4 August 2011

Introduction: why natural language matters

Mature technology

Mature technology is **invisible**: the user doesn't notice it.

Example: Unix

- in 1995: the user had to be a Unix hacker
- in 2011: Unix is hidden under MacOS or Ubuntu or Android

When will formal proof systems mature?

Many systems are sophisticated, efficient, and robust.

But they require an expert to use, with special training.

One reason: formalized proof languages

- close to the machine language of the proof engine
- constantly reminds the user of the existence of the engine.

Some use cases

- **Students** wanting help to construct and verify proofs.

- **Mathematicians** wanting to prove new theorems.

- **Engineers** wanting to verify software or hardware.

In all cases: constant **manual conversions** between informal language and the formalism.

Cf. computer algebra

E.g. Mathematica: normal mathematical notations,

$$\sqrt{x}$$

instead of

```
Sqrt[x]
```

This is *one* reason why computer algebras are main-stream tools in mathematics education, unlike proof systems.

NB proof systems like Matita can now deal with this part.

The language of proof systems

What is the counterpart of algebraic formulas in computer algebra?

Unfortunately, it is not a formal notation.

It is **mathematical text** - a mixture of natural language and formulas.

The natural language part cannot be fully replaced by formulas.

Proof systems must support this

- to hide their internal technology
- to reach the maturity as computer algebras

This has been a popular idea

STUDENT **Bobrow 1964**
Evidence Algorithm **Glushkov 1970**
Mathematical Vernacular **de Bruijn 1994**
Mizar **Trybulec 2006**
OMEGA **Benzmüller & al. 1997**
Isar **Wenzel 1999**
MOWGLI **Asperti & al. 2003**
Vip **Zinn 2004**
SAD **Paskevich & al. 2004**
Theorema **Buchberger & al. 2006**
MathLang **Kamareddine & Wells 2008**
Naproche **Cramer & al. 2009**
FMathL **Neumaier 2009**

Not our goal

We are *not* proposing yet another proof language.

Instead:

- an analysis of mathematical language
- a method for implementing parts of it

The method can be applied to existing systems.

We show **software** and a **library** that can be used.

Controlled Natural Language

A common feature of all systems mentioned: they use English-like notations permitting

- user input via **parser** (not always)
- system output via **printer** (not always)

These are always different fragments

- built from scratch over and over again
- requires learning by the user
- considerable effort of development and maintenance

Our goal today

A method to implement natural-like mathematical languages *easily*,

- in a few days or hours
- with a couple of pages of code

Bonus: **multilinguality**

- you can port the interface from one language to another
- you can translate mathematics

Bonus: **incremental parsing**

- the user is guided to stay within the language

Base-line logical language

Formula:

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

Translations to English, German, French, and Finnish:

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

Easy to translate in both directions.

More sophisticated language

The same formula:

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

Translations to English, German, French, and Finnish:

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

The translation is more tricky but not *difficult*.

Easy vs. difficult

Definition. A problem is **easy** if it can be solved by well-known techniques. (*This doesn't mean that it was easy to develop these techniques in the first place*.)

Definition: A problem is **difficult** if it is not easy.

Example: *easy* problems in natural language interfaces don't require training in linguistics but can use existing tools.

Compare to automated theorem proving

Some classes of formulas are *easy*

Some classes are easy for humans but still impossible for computers

Some classes will remain difficult forever

To make progress (in both natural language processing and automated reasoning):

- identify and extend the classes of *easy* problems
- don't get paralyzed by the impossibility of the full problem.

The Language of Mathematics

The structural levels

Text: chapters, sections

Definitions, theorems, lemmas, examples

diagrams*

Sentences

Words: nouns, adjectives, verbs,...

Symbols: formulas, variables, numbers,...

*a diagram showing a triangle may bind variables used for its sides and angles in the text

The sentence level: two kinds of elements

Verbal: natural language words

Symbolic: mathematical formulas

Some concepts are only verbal:

x is even

Some have both verbal and symbolic expressions:

$x > y$

x is greater than y

Legal and illegal mixtures

A verbal part may contain symbolic parts:

- as **noun phrases**,
  x^2 *is divisible by* $\sqrt{x}$
- as subsentences,
  we conclude that $x^2 > \sqrt{x}$.

A symbolic part may not contain verbal parts:

$$* \sqrt{\text{the sum of all numbers from 1 to 100}}$$

with set comprehensions as a possible exception: $\{x \mid x \text{ is divisible by } 7\}$

Logical constants

Connectives and quantifiers are *never* symbolic (if we want to capture the traditional style).

They are expressed by

- conjunctions: *and*, *or*, *if*
- variables: *for all x*,...
- **in situ quantifiers**: *the square of* **every odd number** *is odd*

Expressing logic in language

Rule 1. Eliminate logical constants by using conjunctions and variables.

Rule 2. Eliminate variables by using in situ quantifiers.

Rule 3. Use symbolic expressions whenever possible.

We can thus improve to a certain limit:

for every odd number x, the square of x is odd $\Longrightarrow$

for every odd number x, x^2 *is odd* $\Longrightarrow$

the square of every odd number is odd $\Longrightarrow$

* (*every odd number*)^2 *is odd*

Living with ambiguity

Ganesalingam 2010: mathematical text is *highly ambiguous*.

The controlled language approach: ban ambiguity by design.

But:

- the semantics needs to be learnt and remembered
- hence it may be misunderstood or forgotten

Better: *detect* ambiguity and eliminate it by **paraphrase** or by **semantic considerations**.

Example: operator scope

The sentence

for all x, x is even or x is odd

has two context-free parses,

for all x, (x is even or x is odd)

(for all x, x is even) or x is odd

The latter is rejected in binding analysis.

Example: PP attachment

Chomsky's example (PP = Prepositional Phrase):

John saw a man with a telescope.

(John saw a man) with a telescope

John saw (a man with a telescope)

Both make sense.

PP attachment in mathematics

(Ganesalingam, from Grigoriev \& al. 1995):

> ρ is normal if ρ generates the splitting field of some polynomial over F_0
>
> (ρ generates the splitting field of some polynomial) over F_0
>
> ρ generates ((the splitting field of some polynomial) over F_0)
>
> ρ generates (the splitting field of (some polynomial over F_0))

Only one of these makes sense...

Example: quantifier scope

The linguists' standard example

every man loves a woman

is interpreted as either $\forall\exists$ or $\exists\forall$.

Many mathematicians would follow the rule that scopes go from left to right.

Counter-examples to left-to-right scope

Not always the preferred interpretation:

In New York City, a pedestrian is hit by a car every five minutes.

A solution exists for every equation of the form $x + p = q$.

Every element of some set of natural numers is prime. (Ganesalingam)

Not invariant under translation:

English: *Every man likes a woman*.

Italian: *Una donna piace a ogni uomo*.

The Method: GF in a Nutshell

What is GF

GF = Grammatical Framework = Logical Framework + concrete syntax

GF was born in 1998 at Xerox Research in Grenoble, as

- an extension of ALF into a grammar formalism
- an extension of type theory from mathematics to all kinds of language
- a declarative approach to multilingual translation systems

Abstract and Concrete Syntax

grammar = abstract syntax + concrete syntax

A BNF grammar rule fuses them

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

In GF it is split into two rules

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

`fun`: function for building **trees** (abstract syntax)

`lin`: **linearization** of trees to **strings** (concrete syntax)

Reversibility

A GF grammar is a declarative program for

- **linearizing** trees to strings
- **parsing** strings to trees

```
             --- linearization --->
(EMul x y)                              x * y
             <---- parsing --------
```

Ambiguity

GF grammars can be **ambiguous**.

Then parsing returns many trees.

```
  x * y * z  ------> (EMul (EMul x y) z)
                     (EMul x (EMul y z))
```

Ambiguity can here be avoided by design (by using precedences).

It may also be unavoidable.

Multilinguality

multilingual grammar = one abstract syntax + many concrete syntaxes

```
fun EMul : Exp -> Exp -> Exp      -- abstract
lin EMul x y = x ++ "*" ++ y      -- Java
lin EMul x y = x ++ y ++ "imul"  -- JVM
```

Compilation:

```
2 * x   ----> (EMul x y) ---->  iconst_2
                                iload_0
                                imul
```

By reversibility, we also have decompilation!

Compiling natural language

As a first approximation (to be corrected),

```
lin EMul x y = "the product of" ++ x ++ "and" ++ y      -- English
lin EMul x y = "le produit de" ++ x ++ "et de" ++ y     -- French
lin EMul x y = "das Produkt von" ++ x ++ "und" ++ y     -- German
lin EMul x y = x ++ "ja" ++ y ++ "tulo"                 -- Finnish
```

A multi-source multi-target compiler-decompiler

```
 the product of x and y                  x:n ja y:n tulo
                       \               /
                         (EMul x y)
                       /               \
le produit de x et de y                  x * y
```

Incremental parsing

The user is guided by the grammatically correct next words.

We show the demo in

http://www.grammaticalframework.org/demos/minibar/mathbar.html

Language-specific features

Languages have *parameters* like gender, case, number.

Example: German requires the dative case (*dem*) for the arguments:

> *das Produkt von dem Produkt von x und y und z*

GF can handle this *without changing the abstract syntax*.

The case parameter for German

Expressions are linearized to **inflection tables**, which have values for each case.

```
param Case = Nom | Dat

lin EMul x y = table {
  Nom => "das Produkt von" ++ x ! Dat ++ "und" ++ y ! Dat ;
  Dat => "dem Produkt von" ++ x ! Dat ++ "und" ++ y ! Dat
  }
```

(This is still simplified, since German has four cases.)

A more involved use of parameters

A predication in German, *x is prime*

```
fun Prime : Exp -> Proposition

lin Prime x = \\ord,mod =>
  let
    ist = case <mod,x.n> of {
      <Ind, Sg> => "ist" ;
      <Ind, Pl> => "sind" ;
      <Conj,Sg> => "sei" ;
      <Conj,Pl> => "seien"
      }
  in case ord of {
       Main => x.s ! Nom ++ ist ++ "unteilbar" ;
       Sub  => x.s ! Nom ++ "unteilbar" ++ ist ;
       Inv => ist ++ x.s ! Nom ++ "unteilbar"
       }
```

Grammar Engineering

Getting all linguistic details right is *difficult* - or at least laborious.

GF makes this *easy* by the **GF Resource Grammar Library**:

- low-level details of morphology and syntax
- 20 languages: Afrikaans, Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Italian, Nepali, Norwegian, Persian, Polish, Punjabi, Romanian, Russian, Spanish, Swedish, Urdu
- effort: 3-5 kLOC of GF code, 3-9 person months per language

mkCl	NP -> V -> Cl	*she sleeps*
mkCl	NP -> V2 -> NP -> Cl	*she loves him*
mkCl	NP -> V3 -> NP -> NP -> Cl	*she sends it to him*
mkCl	NP -> VV -> VP -> Cl	*she wants to sleep*
mkCl	NP -> VS -> S -> Cl	*she say*
mkCl	NP -> VQ -> QS -> Cl	*she wor*
mkCl	NP -> VA -> A -> Cl	*she bec*
mkCl	NP -> VA -> AP -> Cl	*she bec*
mkCl	NP -> V2A -> NP -> A -> Cl	*she pai*
mkCl	NP -> V2A -> NP -> AP -> Cl	*she pai*
mkCl	NP -> V2S -> NP -> S -> Cl	*she ans*
mkCl	NP -> V2Q -> NP -> QS -> Cl	*she ask*
mkCl	NP -> V2V -> NP -> VP -> Cl	*she beg*
mkCl	NP -> A -> Cl	*she is o*
mkCl	NP -> A -> NP -> Cl	*she is o*
mkCl	NP -> A2 -> NP -> Cl	*she is m*
mkCl	NP -> AP -> Cl	*she is v*
mkCl	NP -> NP -> Cl	*she is th*
mkCl	NP -> N -> Cl	*she is a*
mkCl	NP -> CN -> Cl	*she is an old woman*
mkCl	NP -> Adv -> Cl	*she is here*
mkCl	NP -> VP -> Cl	*she always sleeps*
mkCl	N -> Cl	*there is a house*
mkCl	CN -> Cl	*there is an old house*

- API: `mkCl she_NP want_VV (mkVP sleep_V)`
- Afr: *sy wil slaap*
- Bul: *тя иска да спи*
- Cat: *ella vol dormir*
- Dan: *hun vil sove*
- Dut: *ze wil slapen*
- Eng: *she wants to sleep*
- Fin: *hän tahtoo nukkumaan*
- Fre: *elle veut dormir*
- Ger: *sie will schlafen*
- Ita: *lei vuole dormire*
- Nep: उनी सुत्न चाहछिन्
- Nor: *hun vil sove*
- Pes: او می خواهد بخوابد
- Pnb: او سونا چاندی اے
- Pol: *ona chce spać*
- Ron: *ea vrea sā doarmă*
- Rus: *она хочет спать*
- Spa: *ella quiere dormir*
- Swe: *hon vill sova*
- Urd: وہ سونا چاھتی ھے

The GF Resource Grammar API

The product function with the library

API for building noun phrases (`NP`) with relational nouns (`N2`):

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

Usage for English, German, French, and Finnish:

```
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"))
```

The morphology function `mkN` infers the noun inflection from the dictionary form (except in German).

Division of labour

Linguist

- writes the resource grammars
- knows the linguistic details

Application programmer

- writes the application grammar
- knows the domain semantics and idiom

Example: *product* in Finnish

- domain knowledge: pick *tulo* in mathematics, not *tuote*
- linguist knowledge: inflects *tulo*, *tulon*, *tuloa*, ..., *tuloin*

More on GF

http://www.grammaticalframework.org

A. Ranta, *Grammatical Framework: Programming with Multilingual Grammars*, CSLI, Stanford, 2011.

CSLI Studies in Computational Linguistics

Grammatical Framework

Programming with Multilingual Grammars

Aarne Ranta

Baseline Translation for the Core Syntax of Logic

The core formalism

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*

Abstract Syntax in GF

```
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
```

Instantiation to a lexicon

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

An example

English sentence

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

Abstract syntax

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

Now we need to write the concrete syntax.

Concrete syntax of the core calculus

This code is common for all languages in the Resource Grammar Library:

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

Concrete syntax of the lexicon, English

```
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")
```

Concrete syntax of the lexicon, German

```
lin
  IInt  = symb
  Add   = app  (mkN2 (mkN "Summe"))
  Mul   = app  (mkN2 (mkN "Product" "Produkte" neuter))
  Nat   = pred (mkCN (mkA "natürlich") (mkN "Zahl" "Zahlen" feminine))
  Even  = pred (mkA "gerade")
  Odd   = pred (mkA "ungerade")
  Equal = pred (mkA "equal")
```

The predication API

```
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
```

The MOLTO lexicon

200 mathematical concepts from OpenMATH domain lexica

Morphology and combinatorics for 12 languages

Built in GF and reusable as library for new applications

Problems with the baseline grammar

Narrow coverage

Clumsy language

Ambiguity

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.	$(\forall x)(P\&Q)$

Example of connective precedence

Restaurant lunch menu:

Bread and salad or soup, 10 zł

Can you get both bread and soup?

Solution: bullet list

$(P\&Q) \vee R$	$P\&(Q \vee R)$
either of the following:	*both of the following:*
• *bread and sallad*	• *bread*
• *soup*	• *sallad* or *soup*

Precedence order by stipulation would not solve the problem

- users would need to know this
- if *and* binds stronger than *or*, $P\&(Q \vee R)$ is inexpressible!

Bullets by parametrization

(*Easy*.) Use a Boolean parameter that indicates whether a proposition is complex (i.e. formed by a connective). Thus propositions are linearized to **records**

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

The rule: if *none* of the operands is complex, use sentence conjunction; otherwise, use bullets:

```
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}
  }
```

Managing ambiguity, in general

Whether a GF grammar is ambiguous is undecidable.

But it is decidable for any give sentence: test with the parser.

Method:

1. Generate a set of **paraphrases**
2. Order them by e.g. shortness
3. Take the shortest unambigous one

Even better: use *semantics* to disambiguate (but this is in general *difficult*).

Semantic disambiguation methods

1. **Overload resolution** can in GF by **dependent types**:

```
EMul : (t : NumType) -> Exp t -> Exp t -> Exp t
```

to select `dmul` rather than `imul` for `3.14 * x`.

2- **Binding analysis** can in GF be expressed with **higher-order abstract syntax**:

```
Forall : (Var -> Prop) -> Prop
```

to disambiguate *for all x, x is even or x is odd*.

In general *difficult* but needed in full math text.

Beyond the Baseline Translation: Easy Improvements

Compositionality

Linearization in GF is **compositional**:

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

Thus: the subtree structure no longer available.

We have been translating logic to language in this way.

Now:

1. Extend the abstract syntax beyond logic
2. Find best expressions by a non-compositional procedure

Extended Abstract Syntax

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 points 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	*Refl*$(F^2)(x)$	*x is equal to itself*
modified predicate	*Mod*$(K, F)(x)$	*x is an even number*

($\vee$ similar to & and $\exists$ similar to $\forall$.)

New categories

Lists of propositions, predicates, variables, and individual terms.

Predicates with one or two places ("adjectives").

Kind predicates ("nouns").

Atomic propositions.

What we get

The sentence

every natural number is even or odd

as a *compositional* translation of the formula

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

which is a *paraphrase* of the formula

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

whose *compositional* translation is

for all natural numbers x, x is even or x is odd

Defining the paraphrase

Extended to core: easy ("denotational semantics")

Core to extended: more tricky (an optimization problem)

From Extended Syntax to Core Syntax

Denotational semantics, with the core syntax as a *model* of the extended syntax.

The semantics follows the ideas of Montague (1974)

Key question: in-situ quantification.

Key idea: `Ind` is interpreted as `(Ind -> Prop) -> Prop`.

Interpreting quantification

Expected type:

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

Definition:

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

(The bound variable x must be fresh in the context of application.)

Moving out the domain

In the usual way:

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

But the intermediate stage is retained if the target logic formalism supports domain-restricted quantifiers, as e.g. TFF and THF (Sutcliffe & Benzmüller 2010).

Conjunctions of terms

Like quantifiers: functions on propositional functions,

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

Simple predication

Must be "reversed": the argument is applied to the predicate.

$$(F(a))^* = a^* F^*$$

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

Cf. Frege, *Begriffsschrift* (1879), §10: "one can conceive $\Phi(A)$ as a function of the argument Φ"!

Complex predication

Conjunction of predicates:

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

Reflexive predicates:

$$(\mathit{Refl}(F))^* x \ = \ F^* x \, x$$

Modified kind predicates:

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

One direction complete

The sentence

every natural number is even or odd

is parsed to a tree for the formula

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

whose interpretation is the tree for the formula

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

which linearizes to

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

From Core Syntax to Extended Syntax

Problem: given a proposition P, find the **best** possible tree (or trees) that express the **same** proposition as P.

Same is defined by the interpretation above.

Best is defined by the criteria

- minimal use of variables
- maximal use of symbolism
- shortness
- no ambiguity

NLG techniques

Not invented by us - but in GF we can make them language-independent.

NLG = Natural Language Generation (**Reiter & Dale 2000**)

Extracting Text from Proofs (**Coscoy, Kahn & Théry 1995**)

Expressing logical formulas in natural language (**Friedman 1981**)

Implementation in Haskell with

- **embedded grammars**
- **almost compositional operations** (**Bringert and Ranta 2008**)

In-situ quantification

Replace a bound variable with a quantifier phrase.

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

Conditions (dictated by the semantics):

- P is atomic
- P has exactly one occurrence of the variable.

Examples:

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

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

Aggregation

Share common parts - subjects or predicates:

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

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

Examples:

x is even or x is odd $\implies$ *x is even or odd.*

x is even or y is even $\implies$ *x or y is even*

Further optimization: sorting the conjuncts to group maximally long segments.

Effects of aggregation

Shortens the expression

Reduces ambiguity

x is even or x is odd and y is odd $\Longrightarrow$

x is even or odd and y is odd | *x is even or x and y are odd*.

Reduces the number of occurrences of a variable, thus helps in-situ quantification.

A cumulative effect

Thus aggregation can create an opportunity for in-situ quantification:

for all numbers x, x is even or x is odd

$\Longrightarrow$ *for all numbers x, x is even or odd*

$\Longrightarrow$ *every number is even or odd*

Recall: in-situ *before* aggregation would create

every number is even or every number is odd

Extracting kind predicates

To create opportunity for in-situ quantification:

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

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

Verb negation

For atomic propositions,

$$\sim A \Longrightarrow \overline{A}$$

Example:

it is not the case that x is even $\Longrightarrow$ *x is not even*

Condition: no in-situ quantifiers in A, since *every natural number is not even* is ambiguous.

Reflexivization

Equal first and second arguments:

$$F(x,x) \Longrightarrow \textit{Refl}(F)(x)$$

Example:

x is equal to x $\Longrightarrow$ *x is equal to itself*

Opportunity for in-situ quantification: *every natural number is equal to itself*.

Modification

Combine a kind and a modifying predicate into a complex kind predicate

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

Example:

x is a number and x is even $\implies$ *x is an even number*

Opportunity for in-situ quantification: *some even number is prime*.

Flattening

Binary conjunctions to list conjunctions:

$$P \textit{ and } Q \textit{ and } R \Longrightarrow P\text{, } Q \textit{ and } R$$

Effects:

- syntactic ambiguity is eliminated
- bullet lists can have arbitrary length
- opportunities are created for aggregation.

Verbal vs. Symbolic

Principles:

- symbolic expressions are unwanted for logical structure
- otherwise, symbolic expressions are preferred to verbal
- but, symbolic expressions may not have verbal parts

Strategy:

1. Perform the above optimizations.
2. Express symbolically whatever is possible.

Successful symbolic expression

for all x, if x is a number and x is odd, then the sum of x and the square of x is even

$\Longrightarrow$ *for all odd numbers x, the sum of x and the square of x is even*

$\Longrightarrow$ *for all odd numbers x,* $x + x^2$ *is even*

Unsuccessful symbolic expression

for all x, if x is a number and x is odd, then x^2 *is odd*

$\Longrightarrow$ * (*every odd number*)^2 *is odd*

$\Longrightarrow$ *the square of every odd number is odd*

Implementation

Easy: possible in linearization!

Just use a Boolean parameter to say whether an expression is symbolic.

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

A demo system

Input: sentence in Eng, Fin, Fre, Ger, Swe, Latex

Outputs:

- abstract syntax tree and its translations
- **normalized** tree and its translations
- **optimized** tree and its translations

Example 1, parsed

```
echo "for all x , if x is a number , then x is even or x is odd" | ./Trans
```

```
PUniv (VString "x") (PImpl (PAtom (AKind Nat (IVar (VString "x")))) (PConj C
(PAtom (APred1 Even (IVar (VString "x")))) (PAtom (APred1 Odd (IVar (VStrin
```

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

jokaiselle x , jos x on luku , niin x on parillinen tai x on pariton

pour tout x , si x est un entier , alors x est pair ou x est impair

für alle x , wenn x eine Zahl ist , dann ist x gerade oder x ist ungerade

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

för alla x , om x är ett tal , så är x jämnt eller x är udda

Example 1, normalized

```
echo "for all x , if x is a number , then x is even or x is odd" | ./Trans
```

```
PUniv (VString "x") (PImpl (PAtom (AKind Nat (IVar (VString "x")))) (PConj (
(PAtom (APred1 Even (IVar (VString "x")))) (PAtom (APred1 Odd (IVar (VStrin
```

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

jokaiselle x , jos x on luku , niin x on parillinen tai x on pariton

pour tout x , si x est un entier , alors x est pair ou x est impair

für alle x , wenn x eine Zahl ist , dann ist x gerade oder x ist ungerade

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

för alla x , om x är ett tal , så är x jämnt eller x är udda

Example 1, optimized

```
echo "for all x , if x is a number , then x is even or x is odd" | ./Trans
```

```
PAtom (APred1 (ConjPred1 COr (BasePred1 Even Odd)) (IUniv Nat))
```

every number is even or odd

jokainen luku on parillinen tai pariton

tout entier est pair ou impair

jede Zahl ist gerade oder ungerade

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

varje tal är jämnt eller udda

Example 2, parsed

```
echo "for all even numbers x , the square of x is even" | ./Trans
```

```
PUnivs (BaseVar (VString "x")) (ModKind Nat Even) (PAtom (APred1 Even
(IFun1 Square (IVar (VString "x")))))
```

for all even numbers x , x^2 is even

kaikelle parillisille luvuille x , x^2 on parillinen

pour tous les entiers pairs x , x^2 est pair

für alle gerade Zahlen x , ist x^2 gerade

$(\forall x \in \mathit{Mod}(N, \mathit{Even}))(\mathit{Even}(x^2))$

för alla jämna tal x , är kvadraten av x jämn

Example 2, normalized

```
echo "for all even numbers x , the square of x is even" | ./Trans

PUniv (VString "x") (PImpl (PConj CAnd (PAtom (AKind Nat (IVar (VString "x"
(PAtom (APred1 Even (IVar (VString "x"))))) (PAtom (APred1 Even (IFun1 Squa
(IVar (VString "x"))))))
```

for all x , if x is a number and x is even , then x^2 is even

jokaiselle x , jos x on luku ja x on parillinen , niin x^2 on parillinen

pour tout x , si x est un entier et x est pair , alors x^2 est pair

für alle x , wenn x eine Zahl ist und x gerade ist , dann ist x^2 gerade

$(\forall x)(((x \in N)\&(\mathit{Even}(x))) \supset \mathit{Even}(x^2))$

för alla x , om x är ett tal och x är jämnt , så är kvadraten av x jämn

Example 2, optimized

```
echo "for all even numbers x , the square of x is even" | ./Trans
```

```
PAtom (APred1 Even (IFun1 Square (IUniv (ModKind Nat Even))))
```

the square of every even number is even

jokaisen parillisen luvun neliö on parillinen

le carré de tout entier pair est pair

das Quadrat von jeder geraden Zahl ist gerade

$\mathit{Even}(\mathit{square}(\forall \mathit{Mod}(N, \mathit{Even})))$

kvadraten av varje jämnt tal är jämn

The Limits of Easy Techniques

Translating arbitrary text to logic

Can be done: Boxer (**Bos & al. 2004**).

Used for **textual entailment tasks**.

But not precise enough for e.g. proof checking.

In linguistics, there is always a trade-off between **coverage** and **precision**.

The dynamicity of language

(**Ganesalingam 2010**)

The interpretation depends on context.

The context may even extend the *syntax*, when new concepts are defined.

$$\lambda + K = S$$

in **Barendregt 1981** stands for the theory λ enriched with the axiom $K = S$, and should hence be parsed $\lambda + (K = S)$.

(NB programming languages like Haskell permit something similar.)

Generating text from logic

Easy: definitions and theorems, and their sequences.

Difficult: proofs, which are trees.

Generating text from proofs

Main problems:

- restructuring the prood
- ignoring details

To be solved first: make *formal* proofs readable!

Some projects using GF for logic and mathematics

Alfa

Type-theoretical proof editor Alfa + GF grammar + lexical annotations.

Hallgren & Ranta 2000

No dependent types in GF; type checking in Alfa.

Definition. A natural number is defined by the following constructors:
- zero
- the successor of n where n is a natural number.

Definition. Let a and b be natural numbers. Then the sum of a and b is a natural number, defined depending on a as follows:
- for zero, choose b
- for the successor of n, choose the successor of the sum of n and b.

Définition. Les entiers naturels sont définis par les constructeurs suivants :
- zéro
- le successeur de n où n est un entier naturel.

Définition. Soient a et b des entiers naturels. Alors la somme de a et de b est un entier naturel, qu'on définit dépendant de a de la manière suivante :
- pour zéro, choisissons b
- pour le successeur de n, choisissons le successeur de la somme de n et de b.

Definition. Ett naturligt tal definieras av följande konstruerare:
- noll
- efterföljaren till n där n är ett naturligt tal.

Definition. Låt a och b vara naturliga tal. Summan av a och b är ett naturligt tal, som definieras beroende på a enligt följande:
- för noll, välj b
- för efterföljaren till n, välj efterföljaren till summan av n och b.

KeY

Software verification system + GF grammar + lexical annotations + NLG techniques.

Johannisson 2005, **Beckert & al. 2006**

Dependent types giving a type system for OCL and guiding authoring.

- if the try counter is equal to 0 then this implies that the result is equal to false
- if the following conditions are true
  - the try counter is greater than 0
  - *pin* is not equal to null
  - *offset* is at least 0
  - *length* is at least 0
  - *offset* plus *length* is at most the size of *pin*
  - the query `arrayCompare ( the pin , 0 ,` *pin* `,` *offset* `,` *length* `)`[1] on `Util` is equal to 0

  then this implies that the following conditions are true
  - the result is equal to true
  - this owner PIN is validated
  - the try counter is equal to the maximum number of tries
- if the try counter is greater than 0 and at least one of the following conditions is not true
  - *pin* is not equal to null
  - *offset* is at least 0
  - *length* is at least 0
  - *offset* plus *length* is at most the size of *pin*
  - the query `arrayCompare ( the pin , 0 ,` *pin* `,` *offset* `,` *length* `)`[2] on `Util` is equal to 0

  then this implies that the following conditions are true
  - this owner PIN is not validated
  - the try counter is equal to the previous value of the try counter minus 1
  - at least one of the following conditions is true
    * an exception is not thrown and the result is equal to false
    * a null pointer exception is thrown
    * an array index out of bounds exception is thrown

WebALT

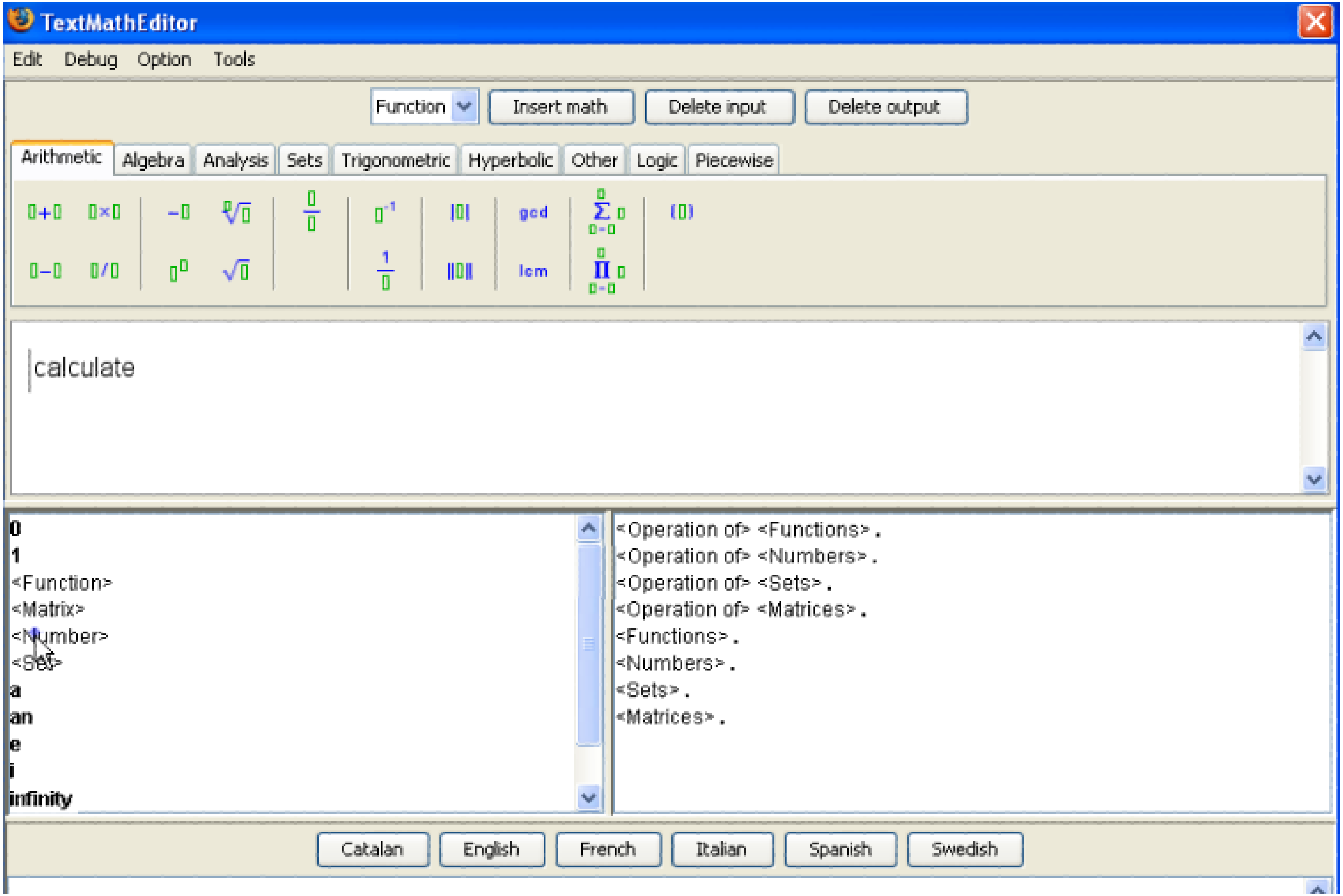
Web Advanced Learning Technology, a European project aiming to build a repository of multilingual math exercises.

Caprotti 2006

Used formalizations from the OpenMath project **Abbott & 1996**

Continued in the MOLTO project

Saludes & Xambó 2011 (THedu at CADE).


TextMathEditor
Edit
Debug
Option
Tools
Function
Insert math
Delete input
Delete output
Arithmetic
Algebra
Analysis
Sets
Trigonometric
Hyperbolic
Other
Logic
Piecewise
gcd
lcm
calculate
0
1
<Function>
<Matrix>
<Number>
<Set>
a
an
e
i
infinity
<Operation of> <Functions> .
<Operation of> <Numbers> .
<Operation of> <Sets> .
<Operation of> <Matrices> .
<Functions> .
<Numbers> .
<Sets> .
<Matrices> .
Catalan
English
French
Italian
Spanish
Swedish

Attempto

Attempto Controlled English, a natural language fragment used for knowledge representation and reasoning (**Fuchs & al 2008**.

Reimplemented in GF and ported to five other languages.

Angelov & Ranta 2010.

On top of this, a natural language interface to OWL in English and Latvian **Gruzitis & Barzdins 2011**.

1	Everything that *eats* something is **an** *animal*.	Tas, kas kaut ko *ed*, ir *dzīvnieks*.
2	Every *carnivore* is **an** *animal* that *eats* **an** *animal*. Every *animal* that *eats* **an** *animal* is **a** *carnivore*.	Ikviens *plesejs* ir *dzīvnieks*, kas *ed* **kadu** *dzīvnieku*. Ikviens *dzīvnieks*, kas *ed* **kadu** *dzīvnieku* ir *plesejs*.
3	Every *herbivore* is **an** *animal* that *eats* nothing but things that are **a** *plant* or that are **a** *part* of nothing but *plants*.	Ikviens *zāledājs* ir *dzīvnieks*, kas *ed* tikai kaut ko, kas ir *augs* vai kas ir tikai *auga daļa*.
4	Every *giraffe* is **a** *herbivore*.	Ikviena *žirafe* ir *zāledājs*.
5	Everything that is *eaten* by **a** *giraffe* is **a** *leaf*.	Tas, ko *ed* **kada** *žirafe*, ir *lapa*.
6	Everything that has **a** *leaf* as **a** *part* is **a** *branch*.	Tas, kura *daļa* ir **kada** *lapa*, ir *zars*.
7	Every *tasty plant* is **a** *nourishment* of **a** *carnivore*.	Ikviens *garšīgs augs* ir **kada** *pleseja barība*.
8	No *animal* is **a** *plant*.	Neviens *dzīvnieks* nav *augs*.
9	If X *eats* Y then Y is **a** *nourishment* of X.	Ja X-s *ed* Y-u, tad Y-s ir X-a *barība*.

SUMO

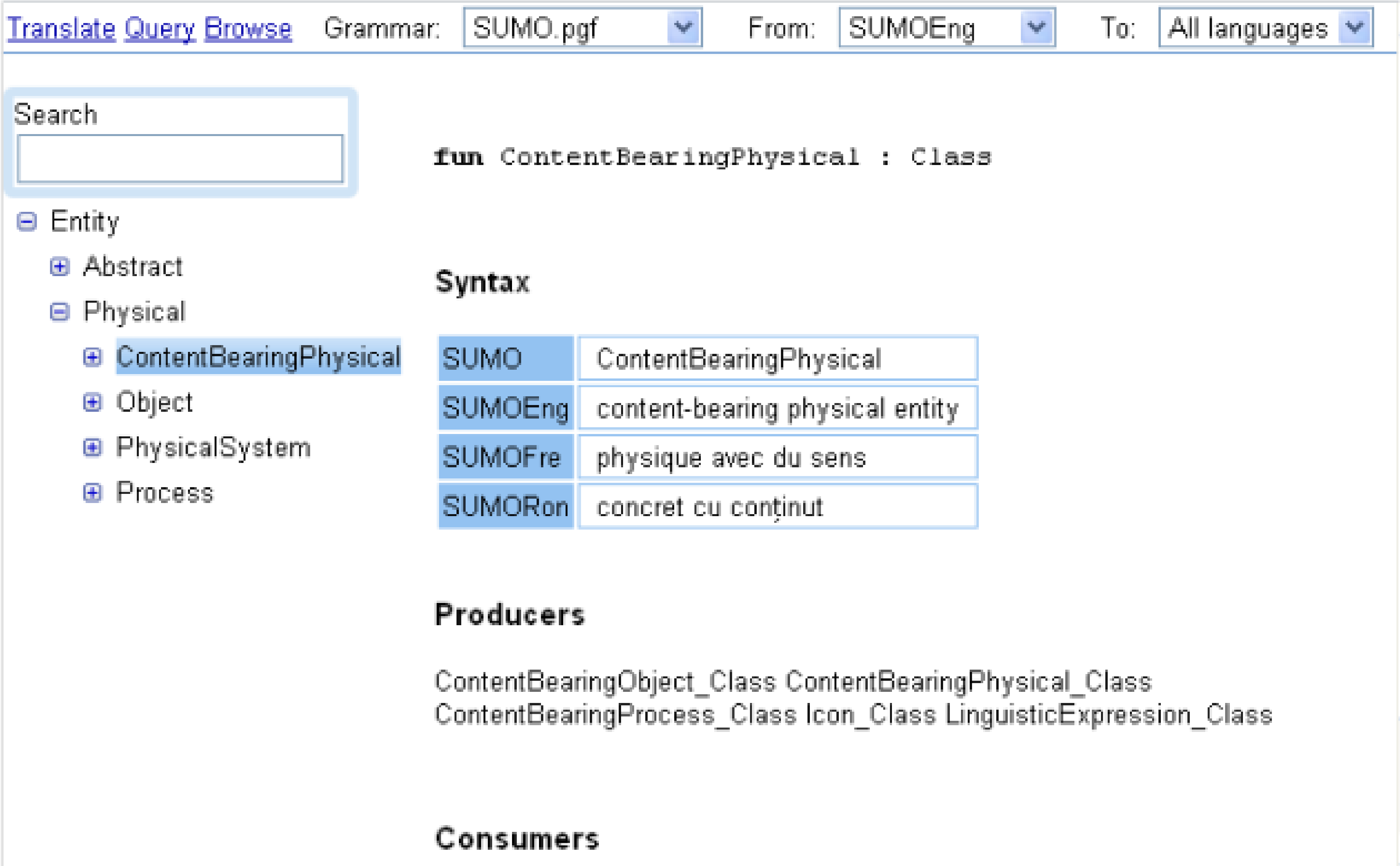
Suggested Upper Merged Ontology (**Pease 2011**)

Converted to GF, with improved natural language generation for three languages using RGL.

Angelov & Enache 2010.

Uses dependent types to express the semantics of SUMO.

Fig. 2. Browser for combined ontology and syntax exploration



MathNat

An educational proof system linked to theorem proving in the TPTP format

Humayoun & Raffalli 2010

http://www.lama.univ-savoie.fr/~humayoun/phd/mathnat.html

Ambitious features, e.g. pronoun resolution.

Theorem

Prove that $\sqrt{2}$ is an irrational number.

Proof: suppose that $\sqrt{2}$ is a rational number. We can assume that $\sqrt{2} = a/b$ by the definition of rational number, where a and b are non zero integers with no common factor. Thus, $\sqrt{2} * b = a$. We get $2 * b^2 = a^2$ - (i) by squaring both sides. Since b^2 and a^2 are non zero integers, we conclude that a^2 is even. By the last deduction, a is even. We can write $a = 2 * c$ by the definition of even numbers, where c is an integer. We get $2 * b^2 = (2 * c)^2 = 4 * c^2$ by substituting the value of a into equation (i). Dividing both sides by 2, yields $b^2 = 2 * c^2$. Because b is a multiple of 2, we conclude that b^2 is even. If a and b are even, then they have a common factor. It is a contradiction.

MOLTO KRI (Knowledge Representation Infrastructure

A query language with a back end in SPARQL for ontology-based reasoning.

Mitankin & al-2010

http://molto.ontotext.com/

Some conclusions

What is easy

To build translators between formal and informal languages in GF:

- reasonably nice language
- portable to 20 languages in the library
- effort: a few days' engineering, or an undergraduate project

What is difficult

To translate arbitrary mathematical text to logic.

To generate good text from complex proofs.

Available code

http://www.grammaticalframework.org/gf-tutorial-cade-2011/code/

```
Trans.hs        -- top loop
TransProp.hs    -- conversions
Makefile
Prop.gf         -- abstract syntax
PropI.gf        -- concrete syntax, functor
PropEng.gf      -- concrete syntax, English
PropFin.gf      -- concrete syntax, Finnish
PropFre.gf      -- concrete syntax, French
PropGer.gf      -- concrete syntax, German
PropSwe.gf      -- concrete syntax, Swedish
PropLatex.gf    -- concrete syntax, symbolic logic in LaTeX
```

(PGreeting GThankYou)

благодаря ти !

gràcies !

tak !

dank je wel !

thank you !

kiitos !

merci !

Danke !

grazie !

takk !

dziękuję !

mulțumesc !

¡ gracias !

tack !

شکریه