Levels of Abstraction in Language and Logic

Aarne Ranta
aarne@chalmers.se

Philosophy and Foundations of Mathematics:
Epistemological and Ontological Aspects

Uppsala, May 5-8, 2009

To Per Martin-Löf

Background

Type theory and linguistics

Sundholm 1986; AR 1990, 1994,...

Type theory as model for language

- formal and informal language
- conceptual, mathematical, and computational model

How has this improved our understanding? our processing capability?

Abstraction

In psychology: formation of general concepts, suppression of details

In mathematics: more and more general definitions and theorems (calculus to topology to category theory)

In linguistics: levels of phonetics, phonology, morphology, syntax, semantics, pragmatics

Prototypical case: phonetics vs. phonology

Invented by Jan Baudouin de Courtenay (1879)

Cornerstone of structuralistic linguistics of the Prague school (Jakobson, Trubetzkoy, 1930's)

Concrete elements: **phones** (sounds); [p], [p'], [b]

Abstract elements: **phonemes** (elements of distinctive sound system of a language): /p/, /b/

Opposition, complementary distribution, free variation

Opposition: [p], [b] represent different phonemes /p/, /b/

- /pin/ [pin] vs. /bin/ [bin]

Complementary distribution: [p], [p'] represent the same phoneme /p/ in in the beginning of a syllable

- /pin/ [p'in] vs. /spin/ [spin]

Free variation: [p], [p'] represent the same phoneme /p/ in the end of a syllable

- *sip* can be pronounced [sip] or [sip']

Morphs and morphemes

Morphs (words, suffixes,...) represent **morphemes**.

The plural *s* and *es* in English are in complementary distribution:

- *tree-trees* vs. *bush-bushes*

The passive *s* and *es* in Swedish are in free variation:

- *skriva - skrivs*|*skrives*

Note: ”free variation” abstracts away from style, dialects, etc

Mathematical models of abstraction

Lambda abstraction: constant to variable, expression to function

Equivalence relations: identify objects that are equal w.r.t. a relevant property

The latter can model **levels of abstraction**

Levels of abstraction in type theory

Syntactic equality

```
4*x + 5  =  4*x + 5
```

Definitional equality

```
4*x + 5  =  x + x + x + x + 5
```

Propositional equality

```
I(N, x + y, y + x)
```

Equivalence relation

```
I(A, p(x), p(y))
```

Questions of syntactic equality

Do the following syntactic equalities hold?

```
          A & B  =  A /\ B

        4*x + 5  =  (4*x) + 5

      plus(x,y)  =  x + y

              N
             2   =  N -> 2

(All x : N)P(x)  =  (All y : N)P(y)
```

Levels of abstraction in syntax

Concrete syntax: string equality

Abstract syntax: tree equality, structural equality (Carnap, Curry, McCarty, Landin)

In abstact syntax, all that matters is

- what parts does the expression have?
- in which way are the parts put together?

Irrelevant in abstract syntax

- How do the parts look like?
- In what order do the parts appear?

Let's try to make this precise!

From logical to grammatical framework

LF, Logical Lramework (Martin-Löf; Harper, Honsell, and Plotkin): abstract syntax rules as **function declarations**,

```
fun plus : N -> N -> N
```

GF, Grammatical Framework (AR): concrete syntax as **linearization rules**,

```
lin plus x y = x "+" y
```

GF = LF + concrete syntax

Context-free grammar

Fuses together abstract and concrete syntax

```
plus. N ::= N "+" N
```

Every rule can be translated to a GF rule pair, abstract + concrete

But GF is more expressive:

- permutation: `F x y = y x`
- suppression: `F x y = y`
- reduplication: `F x = x x`

Parsing and linearization

Linearization: from **abstract syntax tree** to string:

```
plus (plus x y) z  ===> "x + y + z"
```

Parsing: from string to abstract syntax trees:

```
"x + y + z"         ===> plus (plus x y) z ; plus x (plus y z)
```

Thus a string can be **ambiguous** between many trees.

Avoiding ambiguity

We could change the linearization rule to

```
lin plus x y = "(" x "+" y ")"
```

But we don't want this: we use parentheses only when necessary:

- left associativity: `(x + y) + z` can be written `x + y + z`

Precedence as parameter

Linearization of a number expression: a **record** with a string and a precedence number

```
lin plus = infixl 2 "+"
 where
  infixl i f x y = {
    s = parenth i x ++ f ++ parenth (i+1) y ;
    p = i
    } ;
  parenth i x = if (x.p < i) then "(" ++ x.s ++ ")" else x.s
```

We start to use explicit **concatenation** operator `++`

Syntactic equality

Does the following syntactic equality hold?

```
(x + y)  =  x + y
```

Answer:

- in concrete syntax (strings): no
- in abstract syntax (trees): yes, in the sense "strings resulting via linearization from the same tree"

Linearization types

Logical Framework declares new types (**categories**)

```
cat N
```

Grammatical Framework defines their **linearization types**

```
lincat N = {s : Str ; p : Ints 6}
```

Thus: not only linearizations, but also their types can be varied.

Multilingual grammar

One abstract syntax

```
cat N ;
fun plus : N -> N -> N ;
```

Many concrete syntaxes

```
lincat N = {s : Str ; p : Ints 6} ;
lin plus = infixl 2 "+" ;

lin N = Str ;
lin plus x y = x ++ y ++ "iadd" ;
```

Translation in multilingual grammar

Parse in one concrete syntax, linearize in another:

```
1 + 2 + 3   ===>  plus (plus 1 2) 3  ===>  iconst_1
                                           iconst_2
                                           iadd
                                           iconst_3
                                           iadd
```

Thus a multilingual grammar can also define a **compiler**.

Multilingual syntactic equality

Does the following syntactic equality hold?

```
1 + 2  =  iconst_1 iconst_2 iadd
```

Answer:

- in concrete syntax (strings): no
- in abstract syntax (trees)
  - no, not in any single language
  - yes, in a multilingual grammar where they are *strings resulting via linearization from the same tree*

Extending to natural language

English

```
lincat N = Str ;
lin plus x y = "the sum of" ++ x ++ "and"" ++ y
```

Finnish: **inflection table** depending on **case**

```
lincat N = Case => Str ;
lin plus x y = table {
  c => x ! Gen ++ "ja" ++ y ! Gen ++ summa ! c
  }
 where
   summa = table {Nom => "summa" ; Gen => "summan"}
```

Examples

2

two

kaksi

1 + 2

the sum of one and two

yhden ja kahden summa

1 + 2 + 3

the sum of the sum of one and two and three

yhden ja kahden summan ja kolmen summa

Abstract syntax in natural language

What is the abstract syntax of *the sum of one and two* ?

In type theory:

```
plus 1 2
```

In ”standard linguistic syntax“:

```
DetCN the_Det (PrepCN sum_CN of_Prep (ConjNP and_Conj one_NP two_NP))
```

Proper question: what is the abstract syntax of string s in grammar G?

Linguistic syntax

Categories: noun phrase, common noun, determiner, preposition, conjunction

```
cat NP ; CN ; Det ; Prep ; Conj
```

Rules: determination, prepositional modification, coordination

```
fun
  DetCN   : Det -> CN -> NP ;
  PrepCN  : CN -> Prep -> NP -> CN ;
  ConjNP  : Conj -> NP -> NP -> NP ;
```

the sum of one and two

```
DetCN the_Det (PrepCN sum_CN of_Prep (ConjNP and_Conj one_NP two_NP))
```

Grammar composition

Abstract syntax: "semantic structures"

Concrete syntax: use the "syntactic structures" of linguistic grammar to define linearization

```
lincat N = NP ;
lin plus x y =
  DetCN the_Det (PrepCN sum_CN of_Prep (ConjNP and_Conj x y) ;
```

The chain of `lin` rules can be composed at compile time, to produce strings directly from semantic structures.

The GF resource grammar library

Linguistic grammar of 12 languages (20 more forthcoming)

Common syntax: 50 categories, 200 functions

Language-specific morphologies and lexica

Hypothesis: the same abstract structure can be found in different languages. Confirmed for the 12 languages, refuted for none so far.

A substantial task for the 6,000 languages of the world...

Compositionality and reversibility

The hypothesis would be empty, if `lin` could be arbitrary functions.

However, they are constrained by two principles:

1. **Compositionality**: linearization is a homomorphism.

```
(F a1 ... an)* = F* a1* ... an*
```

2. **Reversibility**: linearization rules are usable for parsing (in practice: they are finite datastructures - nested tuples like in PMCFG, cf. Seki 1990, Ljunglöf 2004).

Utility of common syntax

Linearization can be made to a **functor**, a **parametrized module**

```
lincat N = NP ;
lin plus x y =
  DetCN the_Det (PrepCN sum_CN of_Prep (ConjNP and_Conj x y) ;
```

with the **interface** declaring the syntactic structures and the constant

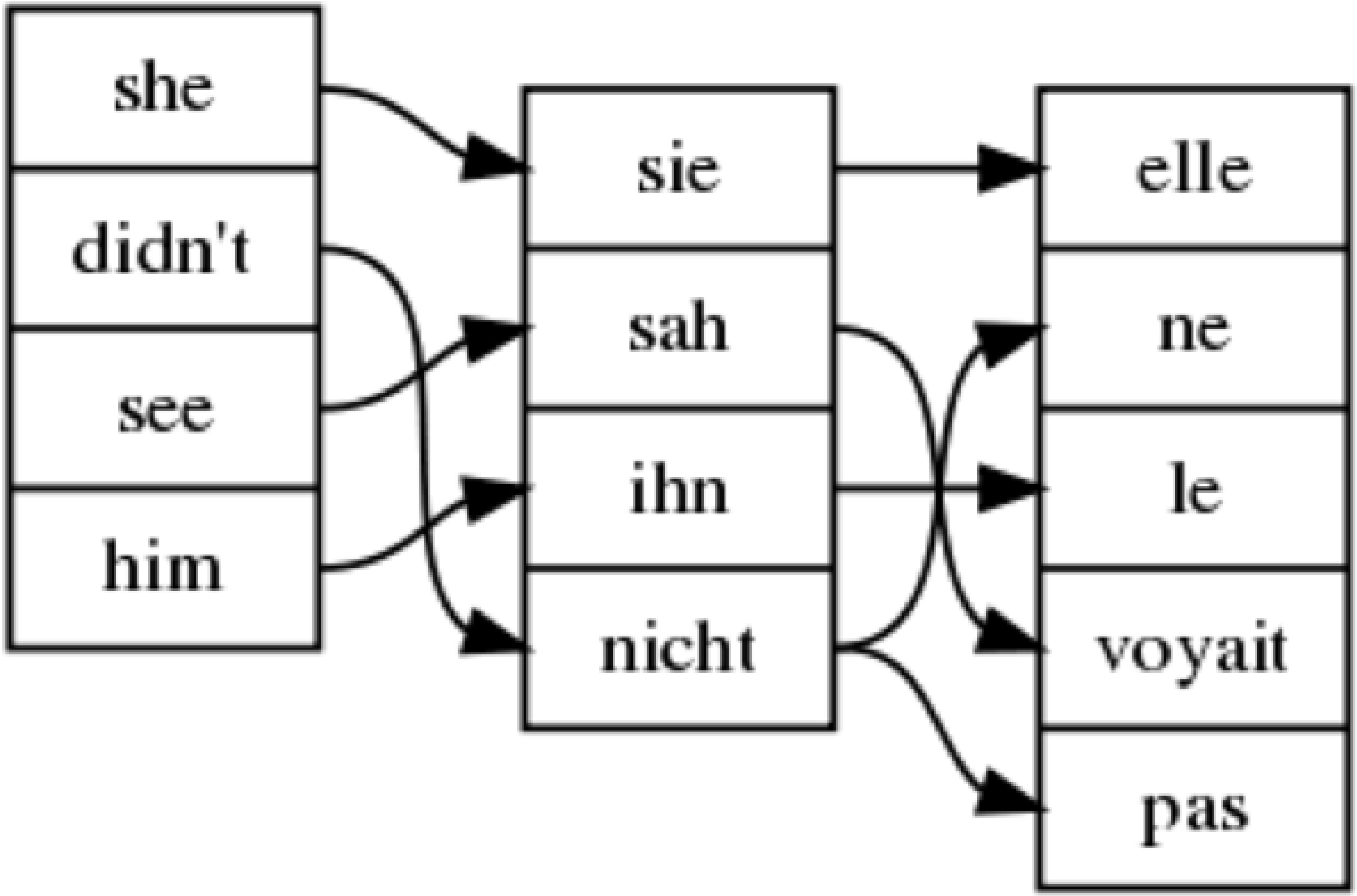
```
sum_CN : CN
```

instantiated in different ways in different languages

```
sum_CN = regCN "sum"     -- English
sum_CN = regCN "summa"   -- Finnish
sum_CN = regCN "somme"   -- French
```

Word alignment

Link words with common smallest spanning abstract syntax subtree.


she
didn't
see
him
sie
sah
ihn
nicht
elle
ne
le
voyait
pas

Literal translation

Literal translation: use the same syntactic structures in source and target.

Functors are the closest we can get to literate translation:

- words are of course different
- also inflection, agreement, and word order differ
- but the parts and their combinations are the same

Literal = syntactic-structure-preserving

Non-literal translation

Use the linguistic resource in different ways in different languages

```
fun Like : Person -> Person -> Prop

lin Like x y = PredVP x (ComplV2 like_V2 y)      -- English
lin Like x y = PredVP y (ComplV2 piacere_V2 x)  -- Italian
```

John likes Mary ===> `Like John Mary` ===> *Maria piace a Giovanni*

This translation is still *compositional*.

Compositional = semantic-structure-preserving

Non-compositional translation

Easy to construct from non-literal translation via subsentential coordination:

John likes and admires Mary ===>

`ConjRel Like Admire John Mary` ===>

`ConjProp (Like John Mary) (Admire John Mary)` ===>

Maria piace a Giovanni e Giovanni ammira Maria

We need to **paraphrase** the sentence.

What is presented is a **denotation** (e.g. truth value).

Levels of translation

translation	**preserves sameness of**	
identical	string	
literal	linguistic structure	
compositional	semantic structure	
paraphrasing	denotation	

Free variation

In what sense is

```
A & B  =  A /\ B
```

Obviously: linearization to different notations:

```
lin Conj A B  =  A ++ "&"  ++ B
lin Conj A B  =  A ++ "/\" ++ B
```

But GF also has operator | for **free variation**

```
lin Conj A B  =  A ++ ("/\" | "&") ++ B
```

Does this example show a sensible language?

Input vs. output grammars

Free variation is up to a *level of abstraction*.

For instance: abstraction from style

- one author wouldn't use `&` and `/\` in free variation
- it is rarely adequate for **generating output**

But free variation can be useful for **recognizing input**, for instance, in information retrieval.

Input grammars for dialogue systems

Abstract syntax: **requests** such as

```
fun BuyTicket : City -> City -> Request

lin BuyTicket x y =
  (("I want" ++ ((("to buy" | []) ++ ("a ticket")) | "to go"))
   |
  (("can you" | [] ) ++ "give me" ++ "a ticket")
   |
  []) ++
  "from" ++ x ++ "to" ++ y ++
  ("please" | [])
```

In free variation as requests

I want to buy a ticket from Gothenburg to Uppsala

I want to go from Gothenburg to Uppsala

can you give me a ticket from Gothenburg to Uppsala

a ticket from Gothenburg to Uppsala

from Gothenburg to Uppsala

Complementary distribution

Typical case: inflection tables

```
lin Even = table {
 AF Masc Sg => "pair" ;
 AF Masc Pl => "pairs" ;
 AF Fem  Sg => "paire" ;
 AF Fem  Pl => "paires"
 }
```

The French words *pair*, *pairs*, *paire*, *paires* are in complementary distribution.

They express the same abstract syntax in the same concrete syntax, but in different contexts.

Questions of syntactic equality revisited

Do the following syntactic equalities hold?

```
           A & B  =  A /\ B

         4*x + 5  =  (4*x) + 5

       plus(x,y)  =  x + y

               N
              2   =  N -> 2

 (All x : N)P(x)  =  (All y : N)P(y)
```

Questions of syntactic equality revisited

Do the following syntactic equalities hold?

```
          A & B  =  A /\ B            YES

        4*x + 5  =  (4*x) + 5         NO

      plus(x,y)  =  x + y             YES

              N
             2   =  N -> 2            YES

(All x : N)P(x)  =  (All y : N)P(y)   NO
```

Free parentheses?

Both `(x + y) + z` and `x + y + z` should be OK, and can be obtained,

```
parenth i x = if (x.p < i)
  then "(" ++ x.s ++ ")"
  else x.s | "(" ++ x.s ++ ")"
```

But current GF can't permit *arbitrarily many* parentheses.

Neither can we permit alpha conversion as syntactic equality.

Finite automata

To express the variations in the abstract syntax

```
MkNP : Prep -> Num -> CN -> NP
for, with, Nom : Prep
one, five : Num
man, woman : CN
```

picture

Two dimensions of syntactic equality

Equality in given abstract syntax
- as strings
- in complementary distribution
- in free variation

Level of granularity in abstract syntax
- syntactic
- propositional-semantic
- pragmatic