Informath:
Interlingual Autoformalization and Informalization with Dedukti and GF

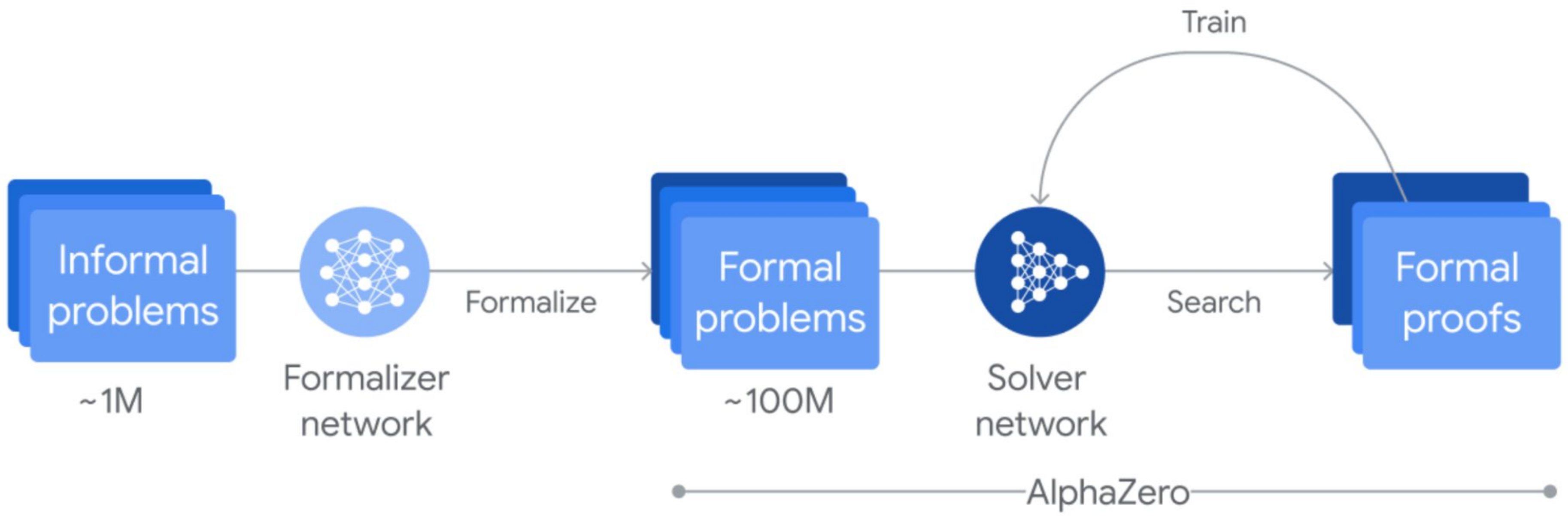
INRIA/ENS Saclay 10 April 2025

Aarne Ranta
aarne.ranta@cse.gu.se

Prologue:

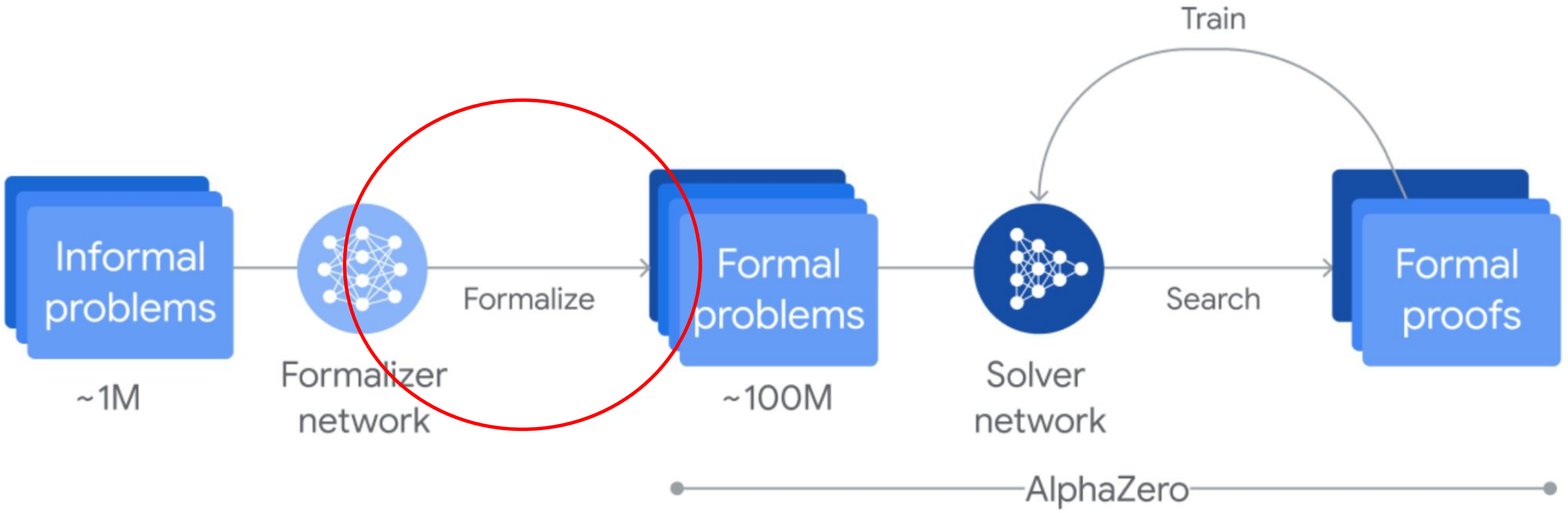
AlphaProof and Autoformalization

AI achieves silver-medal standard solving International Mathematical Olympiad problems



https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

AI achieves silver-medal standard solving International Mathematical Olympiad problems



"First, the problems were manually translated into formal mathematical language for our systems to understand."

https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

(In)formalization, symbolic CNL

Coscoy, Kahn & Théry 1994: Coq proofs to text

Wenzel 1999: Isabelle-Izar

Hallgren & Ranta 2000: GF-Alfa (Agda)

Paskevich 2007: ForTheL

Cramer, Koepke & al 2009: Naproche

Humayoun & Raffalli 2011: MathNat

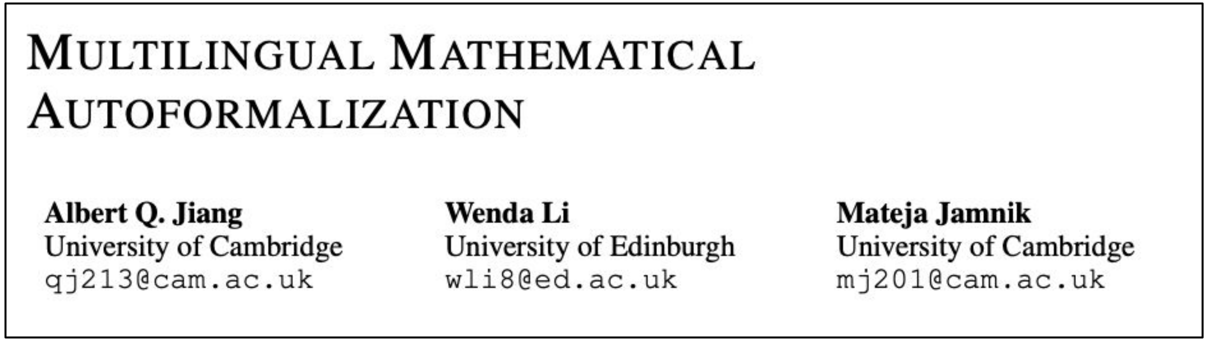
Pathak 2023: GF-Lean

Massot 2024: Verbose-Lean4

Autoformalization, neural

Wang, Brown, Kaliczyk & Urban 2020: NMT and Mizar

Wu, Jiang, Li, Rabe, Staats, Jamnik & Szegedy 2022 : autoformalization with LLM


Multilingual Mathematical Autoformalization

Albert Q. Jiang
University of Cambridge
qj213@cam.ac.uk

Wenda Li
University of Edinburgh
wli8@ed.ac.uk

Mateja Jamnik
University of Cambridge
mj201@cam.ac.uk

- "informalisation is much easier than formalisation"

- uses an GPT-4 to produce the dataset MMA to fine-tune LLaMA
  - ~70% more or less acctable

- resulting autoformalization:
  - 16-18% "acceptable with minimal corrections"

- ~~symbolic informalization~~

https://arxiv.org/abs/2311.03755

"symbolic informalisation tools

- result in natural language content that lacks the inherent diversity and flexibility in expression: they are rigid and not natural-language-like.

- symbolic informalisation tools are hard to design and implement

- They also differ a lot for different formal languages, hence the approach is not scalable for multiple formal languages. "

https://arxiv.org/abs/2311.03755

Informath

The goal of Informath

Symbolic informalization that

has

- results in natural language content that ~~lacks~~ the inherent diversity and flexibility in expression: they are ~~rigid and not~~ natural-language-like.

feasible

- symbolic informalisation tools are ~~hard~~ to design and implement *with proper methods*

can be shared

- They ~~also differ a lo~~t for different formal languages, hence the approach is ~~not~~ scalable for multiple formal languages. *And even for multiple natural languages.*

Agda:

```
postulate prop110 :
  (a : Int) -> (c : Int) ->
  and (odd a) (odd c) -> all Int (\ b ->
    even (plus (times a b) (times b c)))
```

Prop110. Let $a, c \in Z$. Assume that both a and c are odd. Then $ab + bc$ is even for all integers b.

Coq:

```
prop110 : forall a : Int, forall c : Int,
  (odd a /\ odd c -> forall b : Int,
   even (a * b + b * c)) .
```

Prop110. Soient $a, c \in Z$. Supposons que a et c sont impairs. Alors $ab + bc$ est pair pour tous les entiers b.

Lean:

```
prop110 (a c : Int) (x : odd a ∧ odd c)
:
  ∀ b : Int, even (a * b + b * c)
```

Prop110. Låt $a, c \in Z$. Anta att både a och c är udda. Då är $ab + bc$ jämnt för alla heltal b.

Agda:

```
postulate prop110 :
  (a : Int) -> (c : Int) ->
  and (odd a) (odd c) -> all Int (\ b ->
    even (plus (times a b) (times b c)))
```

Coq:

```
prop110 : forall a : Int, for
  (odd a /\ odd c -> forall b
   even (a * b + b * c)) .
```

Dedukti:

```
prop110 : (a : Elem Int) ->
  (c : Elem Int) ->
    Proof (and (odd a)
   (odd c)) ->
    Proof (forall Int
   (b => even (plus
   (times a b) (times b c)))).
```

Lean:

```
prop110 (a c : Int) (x : odd a ∧ odd c)
:
  ∀ b : Int, even (a * b + b * c)
```

Prop110. Let $a, c \in Z$. Assume that both a and c are odd. Then $ab + bc$ is even for all integers b.

Prop110. Soient $a, c \in Z$. Supposons que a et c sont impairs. Alors $ab + bc$ est pair pour tous les entiers b.

Prop110. Låt $a, c \in Z$. Anta att både a och c är udda. Då är $ab + bc$ jämnt för alla heltal b.

```
Agda:

postulate prop110 :
  (a : Int) -> (c : Int) ->
  and (odd a) (odd c) -> all Int (\ b ->
    even (plus (times a b) (times b c)))
```

```
Coq:

prop110 : forall a : Int, for
  (odd a /\ odd c -> forall b
   even (a * b + b * c)) .
```

```
Lean:
prop110 (a c : Int) (x : odd a ∧ odd c)
:
  ∀ b : Int, even (a * b + b * c)
```

```
Dedukti:

prop110 : (a : Elem Int) ->
  (c : Elem Int) ->
    Proof (and (odd a)
   (odd c)) ->
    Proof (forall Int
   (b => even (plus
   (times a b) (times b c)))).
```

```
GF:

AxiomJmt (StrLabel "prop110")
(ConsHypo (LetFormulaHypo (FElem
(ConsTerm (TIdent (StrIdent "a"))
(BaseTerm (TIdent (StrIdent "c"))))
(SetTerm integer_Set))) (ConsHypo
(PropHypo (AdjProp odd_Adj (AndExp
(BaseExp (TermExp (TIdent (StrIdent
"a"))) (TermExp (TIdent (StrIdent
"c"))))))) BaseHypo)) (PostQuantProp
(AdjProp even_Adj (TermExp
(AppOperTerm plus_Oper (TTimes (TIdent
(StrIdent "a")) (TIdent (StrIdent
"b"))) (TTimes (TIdent (StrIdent "b"))
(TIdent (StrIdent "c")))))))
(AllIdentsKindExp (BaseIdent (StrIdent
"b")) (SetKind integer_Set)))
```

Prop110. Let $a, c \in Z$. Assume that both a and c are odd. Then $ab + bc$ is even for all integers b.

t $a, c \in Z$. Supposons que airs. Alors $ab + bc$ est pair ntiers b.

Prop110. Låt $a, c \in Z$. Anta att både a och c är udda. Då är $ab + bc$ jämnt för alla heltal b.

Interlude: GF

GF = Grammatical Framework

GF = Logical Framework + Grammar

First release 1998 at Xerox Research Centre Europe, Grenoble

Based on earlier work with ALF (Another LF, predecessor of Agda) 1992

https://www.grammaticalframework.org/

Abstract and concrete syntax: judgements

```
-- abstract syntax = LF

cat C  Γ

fun f : T

def t = u
```

```
-- concrete syntax

lincat C = L

lin f = t

param P = C | … | C

oper h : T = t
```

Abstract and concrete syntax: examples

```
-- abstract syntax = LF

cat Prop ; Term

fun commutative : Term -> Prop

def commutative f =
  forall Obj (\x, y ->
     Id Obj (f x y) (f y x)
```

```
-- concrete syntax

lincat Prop, Term = Str

lin commutative x =
     x ++ "is commutative"
```

Concrete syntax: parameters and operations

```
-- abstract syntax = LF

cat Prop ; Term

fun commutative : Term -> Prop
```

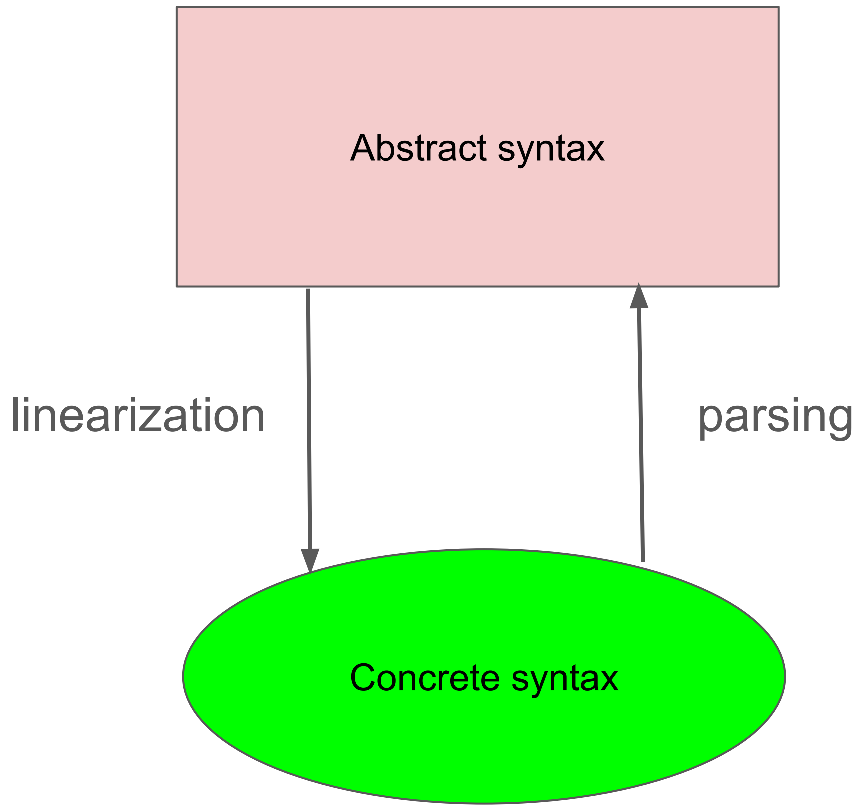
```
-- concrete syntax for English

lincat
  Prop = Str
  Term = {s : Str ; n : Number}

lin commutative x = x.s ++
   copula ! x.n ++ "commutative"

param
  Number = Sg | Pl

oper
  copula : Number => Str =
    table {Sg => "is" ; Pl => "are"}
```

Concrete syntax: parameters and operations

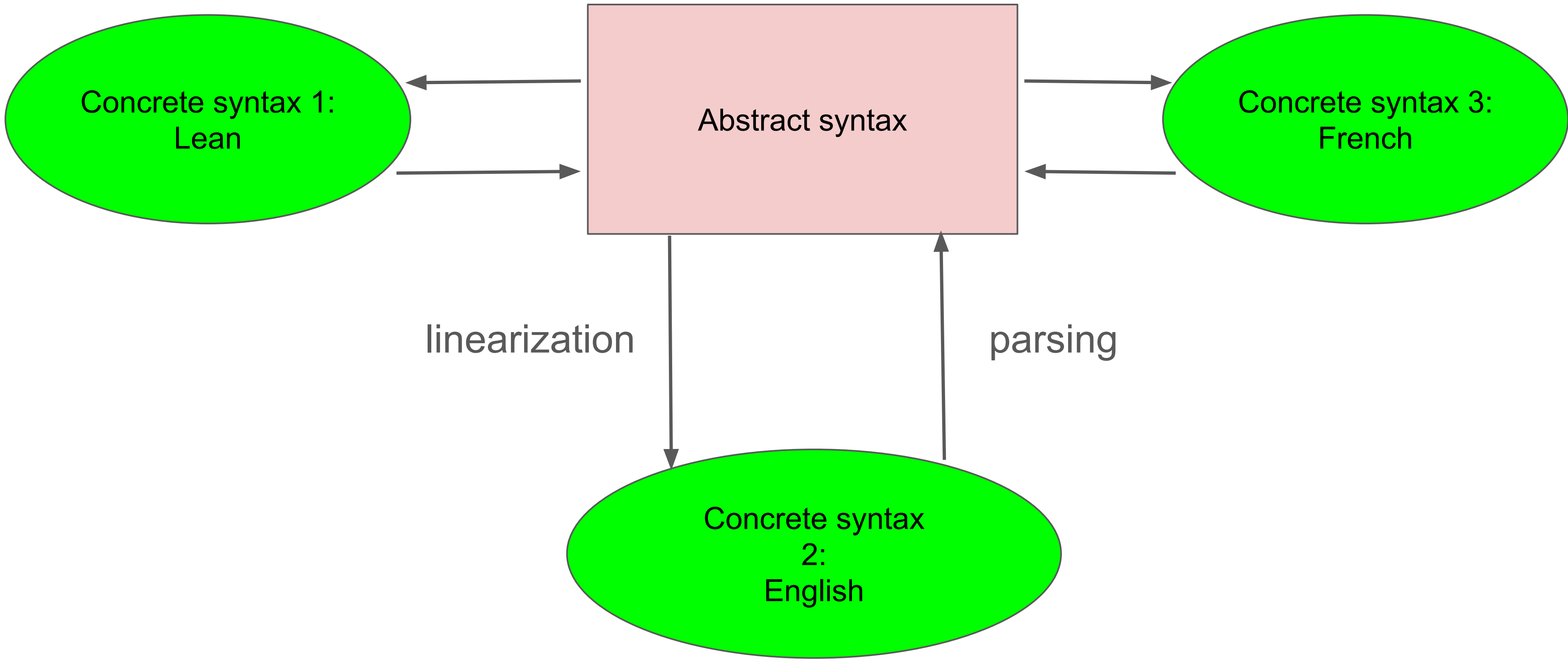
```
-- abstract syntax = LF

cat Prop ; Term

fun commutative : Term -> Prop
```

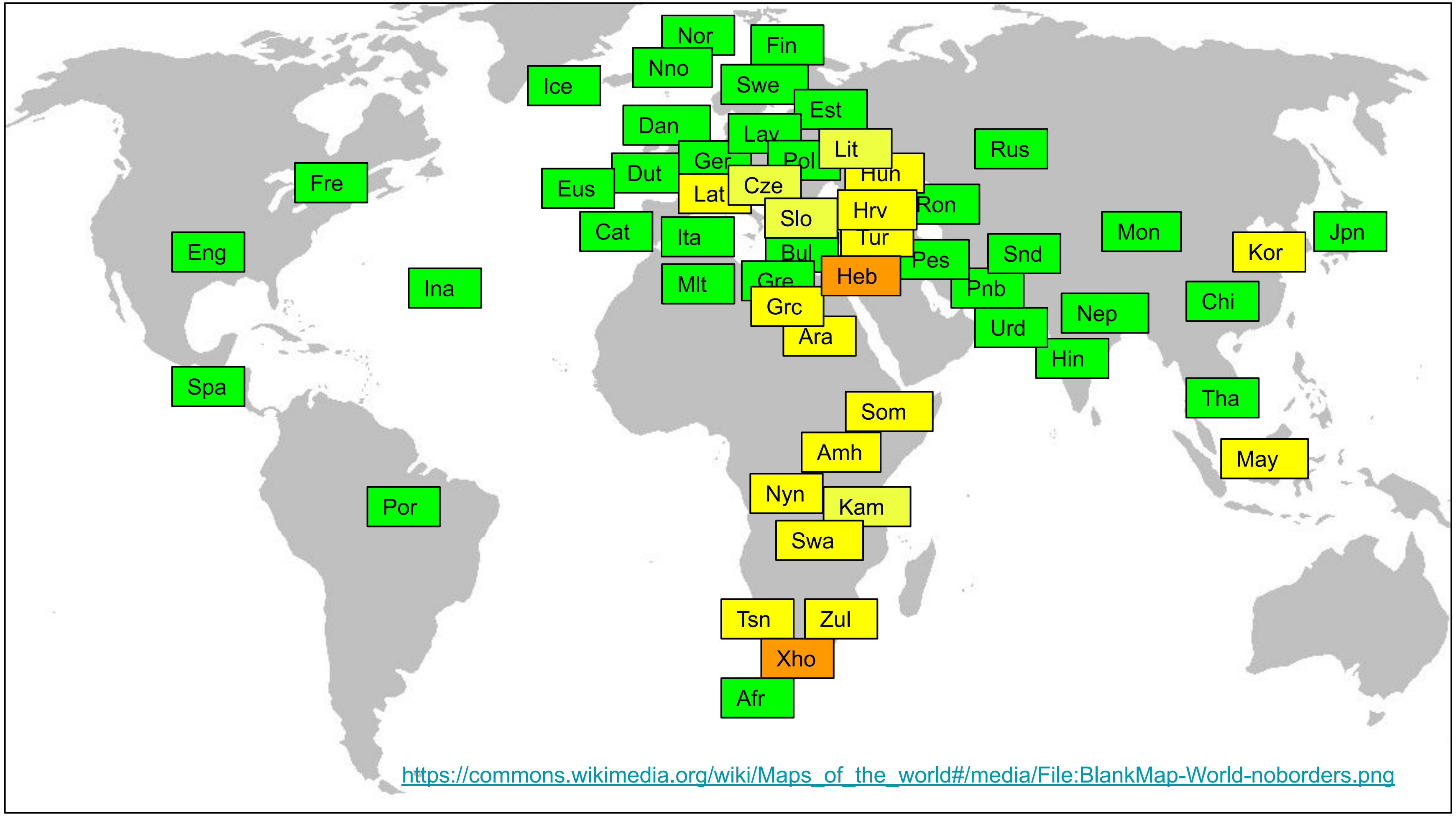
```
-- concrete syntax for French

lincat
  Prop = Mood => Str
  Term = {s : Str ; g : Gender ; n : Number}

lin commutative x = \\m => x.s ++
   copula ! m ! n ++
   mkA "commutatif" ! x.g ! x.n


param
   Number = Sg | Pl
   Gender = Masc | Fem
   Mood = Ind | Subj
oper
  mkA : Str -> Gender => Number = Str = ...
  copula : Mood => Number => Str = ...
```

Reversible mappings



Multilingual grammars





RGL = Resource Grammars Library, created by the GF community 2001-2024

RGL = Resource Grammar Library

morphology and syntax for ~50 languages

```
-- inflection of French adjectives, slightly simplified

mkA : Str -> A = \adj ->
    case adj of {
      _ + "eux"=> <adj, init adj + "se", adj, init adj + "ses"> ;
      _ + "al" => <adj, adj + "e", init adj + "ux", adj + "es"> ;
      _ + "en" => <adj, adj + "ne", adj + "s", adj + "nes"> ;
      _ + "el" => <adj, adj + "le", adj + "s", adj + "les"> ;
      x + "er" => <adj, x + "ère", adj + "s", x + "ères"> ;
      _ + "if" => <adj, init adj + "ve", adj + "s", init adj + "ves"> ;
      _ + "s"  => <adj, adj + "e", adj, adj + "es"> ;
      _ + "e"  => <adj, adj, adj + "s", adj + "s"> ;
      _        => <adj, adj + "e", adj + "s", adj + "es">
      } ;
```

http://www.grammaticalframework.org/lib/doc/synopsis/

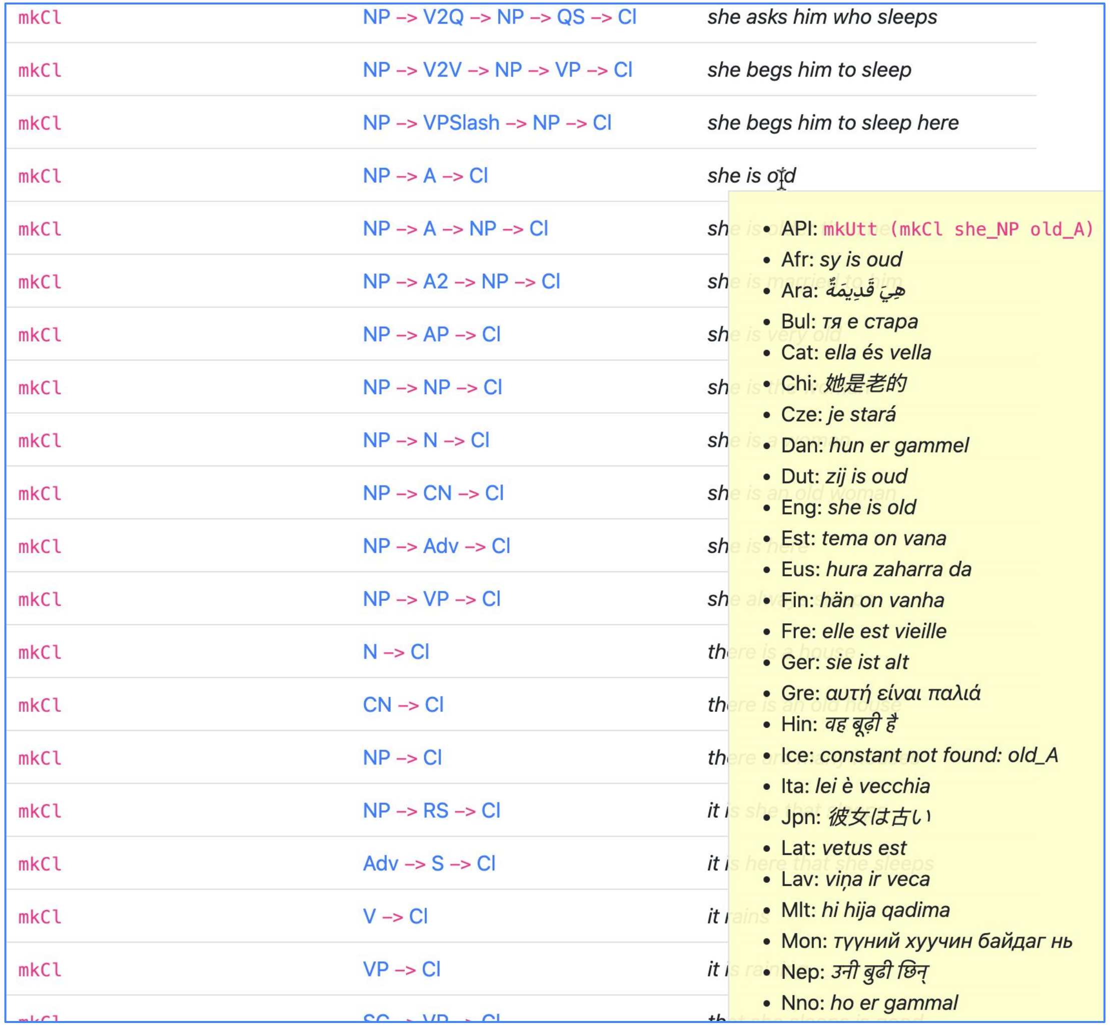
RGL

syntactic combination API

shared by all languages in the library

usable as functor interface + instances

http://www.grammaticalframework.org/lib/doc/synopsis/



`mkCl`	NP -> V2Q -> NP -> QS -> Cl	*she asks him who sleeps*
`mkCl`	NP -> V2V -> NP -> VP -> Cl	*she begs him to sleep*
`mkCl`	NP -> VPSlash -> NP -> Cl	*she begs him to sleep here*
`mkCl`	NP -> A -> Cl	*she is old*
`mkCl`	NP -> A -> NP -> Cl	*sh*
`mkCl`	NP -> A2 -> NP -> Cl	*sh*
`mkCl`	NP -> AP -> Cl	*sh*
`mkCl`	NP -> NP -> Cl	*sh*
`mkCl`	NP -> N -> Cl	*sh*
`mkCl`	NP -> CN -> Cl	*sh*
`mkCl`	NP -> Adv -> Cl	*sh*
`mkCl`	NP -> VP -> Cl	*sh*
`mkCl`	N -> Cl	*th*
`mkCl`	CN -> Cl	*th*
`mkCl`	NP -> Cl	*th*
`mkCl`	NP -> RS -> Cl	*it i*
`mkCl`	Adv -> S -> Cl	*it i*
`mkCl`	V -> Cl	*it i*
`mkCl`	VP -> Cl	*it i*
`mkCl`	SC -> VP -> Cl	*th*

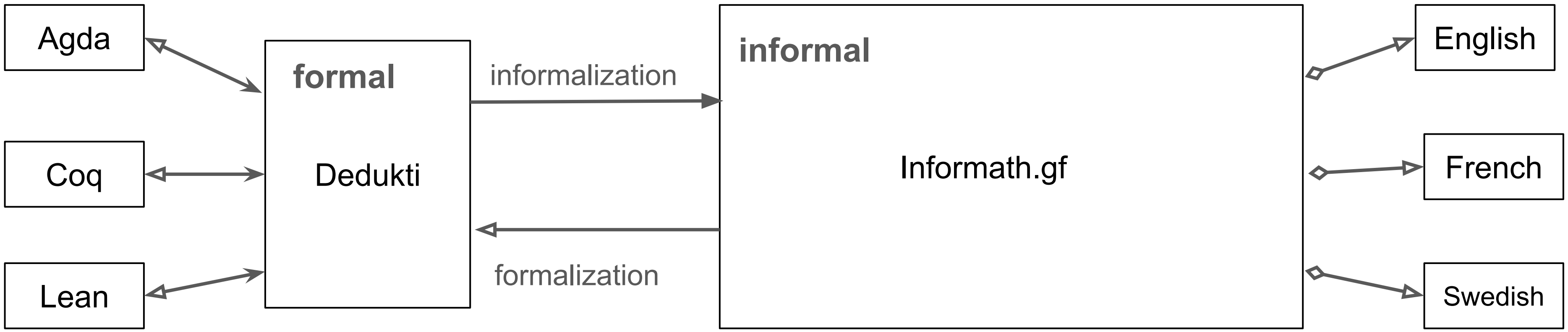
- API: `mkUtt (mkCl she_NP old_A)`
- Afr: *sy is oud*
- Ara: *هِيَ قَدِيمَةٌ*
- Bul: *тя е стара*
- Cat: *ella és vella*
- Chi: *她是老的*
- Cze: *je stará*
- Dan: *hun er gammel*
- Dut: *zij is oud*
- Eng: *she is old*
- Est: *tema on vana*
- Eus: *hura zaharra da*
- Fin: *hän on vanha*
- Fre: *elle est vieille*
- Ger: *sie ist alt*
- Gre: *αυτή είναι παλιά*
- Hin: *वह बूढ़ी है*
- Ice: *constant not found: old_A*
- Ita: *lei è vecchia*
- Jpn: *彼女は古い*
- Lat: *vetus est*
- Lav: *viņa ir veca*
- Mlt: *hi hija qadima*
- Mon: *түүний хуучин байдаг нь*
- Nep: *उनी बुढी छिन्*
- Nno: *ho er gammal*

Concrete syntax: functor over the RGL

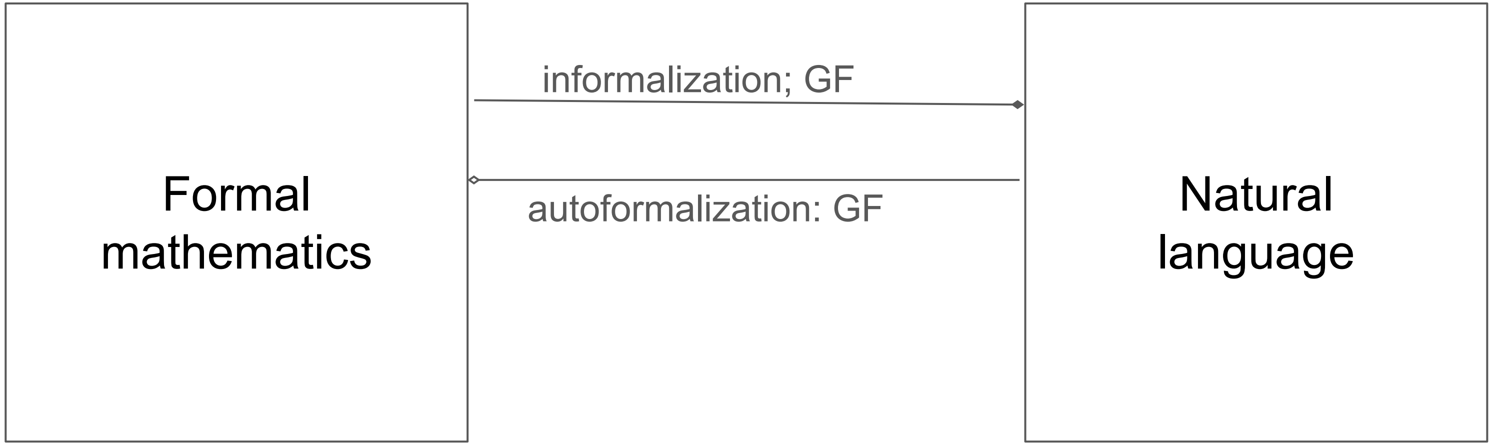
```
-- abstract syntax code

cat Prop ; Term
fun commutative : Term -> Prop


-- shared functor code

lincat
  Prop = Cl
  Term = NP

lin
  commutative x =
     mkCl x commutative_A
```

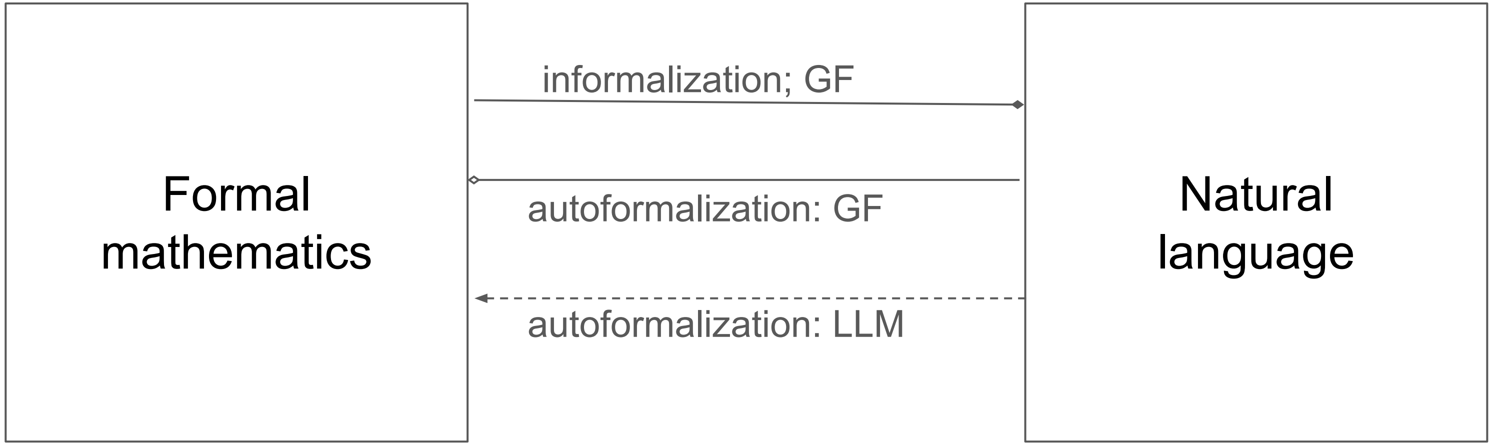
```
-- added code for each language

-- Eng
  commutative_A =
    mkA "commutative"

-- Fre
  commutative_A =
    mkA "commutatif"

-- Fin
  commutative_A =
    mkA "kommutatiivinen"
```

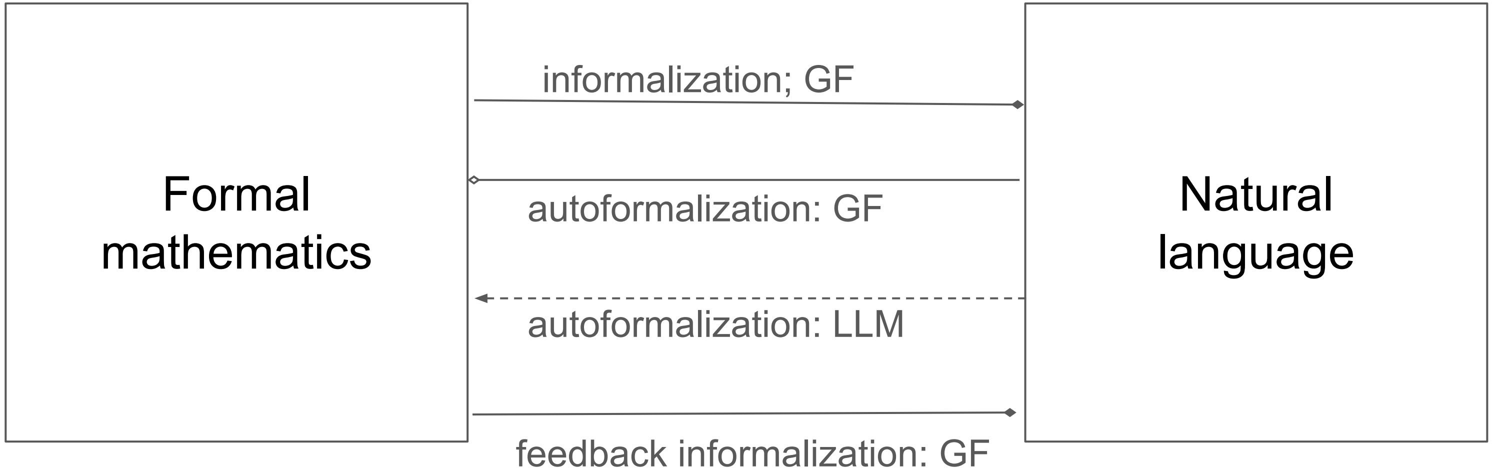
Back to Informath

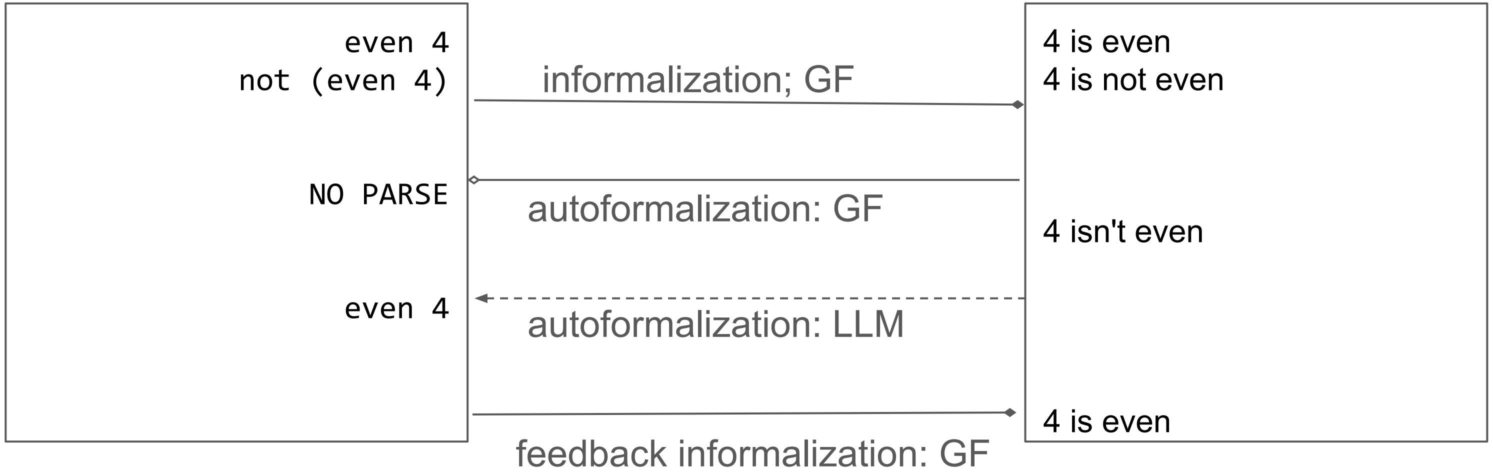


	to one	to many
total	—▶	—◆
partial	—▷	—◇


Formal mathematics
informalization; GF
autoformalization: GF
Natural language


Formal mathematics
Natural language
informalization; GF
autoformalization: GF
autoformalization: LLM

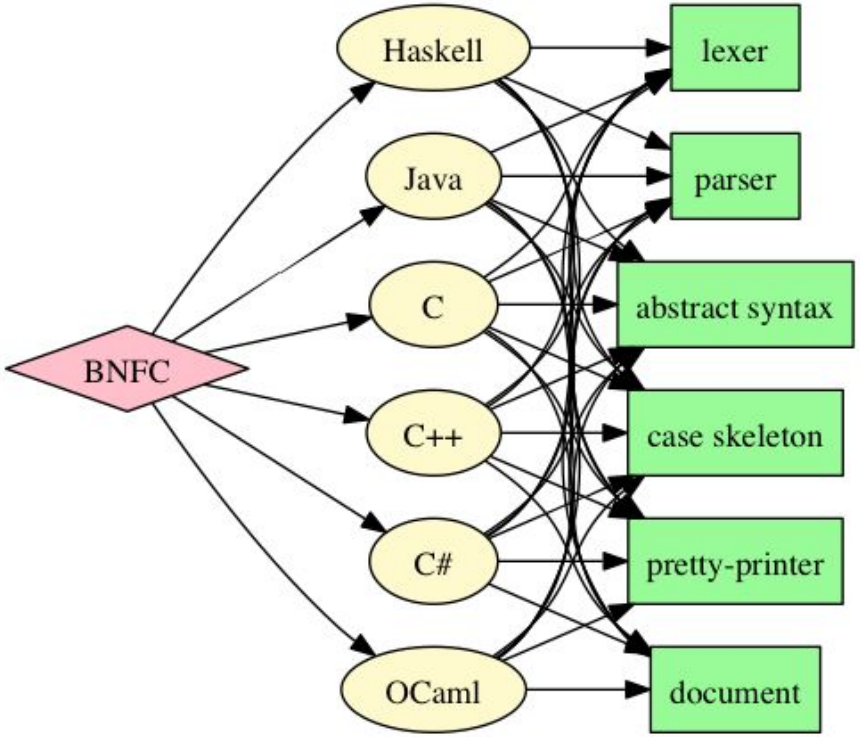

Formal mathematics
Natural language
informalization; GF
autoformalization: GF
autoformalization: LLM
feedback informalization: GF


even 4
not (even 4)
informalization; GF
4 is even
4 is not even
NO PARSE
autoformalization: GF
4 isn't even
even 4
autoformalization: LLM
4 is even
feedback informalization: GF

Mapping between Dedukti and GF

```
-- Dedukti.bnf

MJmts. Module ::= [Jmt] ;

terminator Jmt "" ;

comment "(;" ";)" ;
comment "#" ; ----

JStatic.  Jmt ::= QIdent ":" Exp "." ;
JDef.     Jmt ::= "def" QIdent MTyp MExp "." ;
JInj.     Jmt ::= "inj" QIdent MTyp MExp "." ;
JThm.     Jmt ::= "thm" QIdent MTyp MExp "." ;
JRules.   Jmt ::= [Rule] "." ;

RRule.  Rule ::= "[" [Pattbind] "]" Patt "-->" Exp ;
separator nonempty Rule "" ;

separator Pattbind "," ;

MTNone. MTyp ::= ;
MTExp.  MTyp ::= ":" Exp ;

MENone. MExp ::= ;
MEExp.  MExp ::= ":=" Exp ;

EIdent.  Exp9 ::= QIdent ;
EApp.    Exp5 ::= Exp5 Exp6 ;
EAbs.    Exp2 ::= Bind "=>" Exp2 ;
EFun.    Exp1 ::= Hypo "->" Exp1 ;

coercions Exp 9 ;

-- plus some rules for Hypo and Bind

token QIdent (letter | digit | '_' | '!' | '?' | '\'')+
('.' (letter | digit | '_' | '!' | '?' | '\'')+)? ;
```

```
-- Dedukti.bnf

MJmts. Module ::= [Jmt] ;

terminator Jmt "" ;

comment "(;" ";)" ;
comment "#" ; ----

JStatic.  Jmt ::= QIdent ":" Exp "." ;
JDef.     Jmt ::= "def" QIdent MTyp MExp "." ;
JInj.     Jmt ::= "inj" QIdent MTyp MExp "." ;
JThm.     Jmt ::= "thm" QIdent MTyp MExp "." ;
JRules.   Jmt ::= [Rule] "." ;

RRule.  Rule ::= "[" [Pattbind] "]" Patt "-->" Exp ;
separator nonempty Rule "" ;

separator Pattbind "," ;

MTNone. MTyp ::= ;
MTExp.  MTyp ::= ":" Exp ;

MENone. MExp ::= ;
MEExp.  MExp ::= ":=" Exp ;

EIdent.  Exp9 ::= QIdent ;
EApp.    Exp5 ::= Exp5 Exp6 ;
EAbs.    Exp2 ::= Bind "=>" Exp2 ;
EFun.    Exp1 ::= Hypo "->" Exp1 ;

coercions Exp 9 ;

-- plus some rules for Hypo and Bind

token QIdent (letter | digit | '_' | '!' | '?' | '\'')+
('.' (letter | digit | '_' | '!' | '?' | '\'')+)? ;
```



https://bnfc.digitalgrammars.com/

```
-- Dedukti.bnf

MJmts. Module ::= [Jmt] ;

terminator Jmt "" ;

comment "(;" ";)" ;
comment "#" ; ----

JStatic.  Jmt ::= QIdent ":" Exp "." ;
JDef.     Jmt ::= "def" QIdent MTyp MExp "." ;
JInj.     Jmt ::= "inj" QIdent MTyp MExp "." ;
JThm.     Jmt ::= "thm" QIdent MTyp MExp "." ;
JRules.   Jmt ::= [Rule] "." ;

RRule.  Rule ::= "[" [Pattbind] "]" Patt "-->" Exp ;
separator nonempty Rule "" ;

separator Pattbind "," ;

MTNone. MTyp ::= ;
MTExp.  MTyp ::= ":" Exp ;

MENone. MExp ::= ;
MEExp.  MExp ::= ":=" Exp ;

EIdent.  Exp9 ::= QIdent ;
EApp.    Exp5 ::= Exp5 Exp6 ;
EAbs.    Exp2 ::= Bind "=>" Exp2 ;
EFun.    Exp1 ::= Hypo "->" Exp1 ;

coercions Exp 9 ;

-- plus some rules for Hypo and Bind

token QIdent (letter | digit | '_' | '!' | '?' | '\'')+
('.' (letter | digit | '_' | '!' | '?' | '\'')+)? ;
```

```
-- MathCore.gf

abstract MathCore =
  Terms, UserConstants
  ** {
cat
  Jmt ;
  Exp ;
  Exps ;
  Prop ;
  Kind ;
  Hypo ;
  [Hypo] ;
  Proof ;
  Label ;
  -- plus more categories
fun
  ThmJmt : Label -> [Hypo] -> Prop -> Proof -> Jmt ;
  AxiomJmt : Label -> [Hypo] -> Prop -> Jmt ;
  DefPropJmt : Label -> [Hypo] -> Prop -> Prop -> Jmt ;
  DefKindJmt : Label -> [Hypo] -> Kind -> Kind -> Jmt ;
  DefExpJmt  : Label -> [Hypo] -> Exp -> Kind -> Exp -> Jmt ;
  AxiomPropJmt : Label -> [Hypo] -> Prop -> Jmt ;
  AxiomKindJmt : Label -> [Hypo] -> Kind -> Jmt ;
  AxiomExpJmt  : Label -> [Hypo] -> Exp -> Kind -> Jmt ;

  AppExp : Exp -> Exps -> Exp ;
  AbsExp : [Ident] -> Exp -> Exp ;
  TermExp : Term -> Exp ;
  KindExp : Kind -> Exp ;
  TypedExp : Exp -> Kind -> Exp ;

  AndProp : [Prop] -> Prop ;
  OrProp : [Prop] -> Prop ;
  IfProp : Prop -> Prop -> Prop ;
  IffProp : Prop -> Prop -> Prop ;
  NotProp : Prop -> Prop ;
  -- plus many more functions
-- Dedukti.bnf

MJmts. Module ::= [Jmt] ;

terminator

comment "(
comment "#

JStatic.
JDef.
JInj.
JThm.
JRules.

RRule.   Ru
separator

separator

MTNone.  MT
MTExp.   MT

MENone.  ME
MEExp.   ME

EIdent.  E
EApp.    E
EAbs.    E
EFun.    E

coercions

-- plus so

token QIde
('.' (letter | digit | '_' | '!' | '?' | '\'')+)? ;
```

```
module AbsDedukti where

data Tree (a :: Tag) where
    MJmts :: [Jmt] -> Tree 'Module_
    JStatic :: QIdent -> Exp -> Tree 'Jmt_
    JDef :: QIdent -> MTyp -> MExp -> Tree 'Jmt_
    JInj :: QIdent -> MTyp -> MExp -> Tree 'Jmt_
    JThm :: QIdent -> MTyp -> MExp -> Tree 'Jmt_
    JRules :: [Rule] -> Tree 'Jmt_
    RRule :: [Pattbind] -> Patt -> Exp -> Tree 'Rule_
    MTNone :: Tree 'MTyp_
    MTExp :: Exp -> Tree 'MTyp_
    MENone :: Tree 'MExp_
    MEExp :: Exp -> Tree 'MExp_
    EIdent :: QIdent -> Tree 'Exp_
    EApp :: Exp -> Exp -> Tree 'Exp_
    EAbs :: Bind -> Exp -> Tree 'Exp_
    EFun :: Hypo -> Exp -> Tree 'Exp_
    BVar :: QIdent -> Tree 'Bind_
    BTyped :: QIdent -> Exp -> Tree 'Bind_
    PBVar :: QIdent -> Tree 'Pattbind_
    PBTyped :: QIdent -> Exp -> Tree 'Pattbind_
    HExp :: Exp -> Tree 'Hypo_
    HVarExp :: QIdent -> Exp -> Tree 'Hypo_
    HParVarExp :: QIdent -> Exp -> Tree 'Hypo_
    PVar :: QIdent -> Tree 'Patt_
    PBracket :: Patt -> Tree 'Patt_
    PApp :: Patt -> Patt -> Tree 'Patt_
    PBind :: Bind -> Patt -> Tree 'Patt_
    QIdent :: String -> Tree 'QIdent_
```

```
-- MathCore.gf

abstract MathCore =
  Terms, UserConstants
  ** {
cat
  Jmt ;
  Exp ;
  Exps ;
  Prop ;
  Kind ;
  Hypo ;
  [Hypo] ;
  Proof ;
  Label ;
  -- plus more categories
fun
  ThmJmt : Label -> [Hypo] -> Prop -> Proof -> Jmt ;
  AxiomJmt : Label -> [Hypo] -> Prop -> Jmt ;
  DefPropJmt : Label -> [Hypo] -> Prop -> Prop -> Jmt ;
  DefKindJmt : Label -> [Hypo] -> Kind -> Kind -> Jmt ;
  DefExpJmt  : Label -> [Hypo] -> Exp -> Kind -> Exp -> Jmt ;
  AxiomPropJmt : Label -> [Hypo] -> Prop -> Jmt ;
  AxiomKindJmt : Label -> [Hypo] -> Kind -> Jmt ;
  AxiomExpJmt  : Label -> [Hypo] -> Exp -> Kind -> Jmt ;

  AppExp : Exp -> Exps -> Exp ;
  AbsExp : [Ident] -> Exp -> Exp ;
  TermExp : Term -> Exp ;
  KindExp : Kind -> Exp ;
  TypedExp : Exp -> Kind -> Exp ;

  AndProp : [Prop] -> Prop ;
  OrProp : [Prop] -> Prop ;
  IfProp : Prop -> Prop -> Prop ;
  IffProp : Prop -> Prop -> Prop ;
  NotProp : Prop -> Prop ;
  -- plus quite a few more functions
-- Dedukti.bnf

MJmts. Module ::= [Jmt] ;

terminator
comment "(
comment "#

JStatic.
JDef.
JInj.
JThm.
JRules.

RRule.  Ru
separator

separator

MTNone. MT
MTExp.  MT

MENone. ME
MEExp.  ME

EIdent.  E
EApp.    E
EAbs.    E
EFun.    E

coercions

-- plus so

token QIde
('.' (letter | digit | '_' | '!' | '?' | '\'')+)? ;
```

```
module AbsDedukti where

data Tree (a :: Tag) where
    MJmts :: [Jmt] -> Tree 'Module_
    JStatic :: QIdent -> Exp -> Tree 'Jmt_
    JDef :: QIdent -> MTyp -> MExp -> Tree 'Jmt_
    JInj :: QIdent -> MTyp -> MExp -> Tree 'Jmt_
    JThm :: QIdent -> MTyp -> MExp -> Tree 'Jmt_
    JRules :: [Rule] -> Tree 'Jmt_
    RRule :: [Pattbind] -> Patt -> Exp -> Tree 'Rule_
    MTNone :: Tree 'MTyp_
    MTExp :: Exp -> Tree 'MTyp_
    MENone :: Tree 'MExp_
    MEExp :: Exp -> Tree 'MExp_
    EIdent :: QIdent -> Tree 'Exp_
    EApp :: Exp -> Exp -> Tree 'Exp_
    EAbs :: Bind -> Exp -> Tree 'Exp_
    EFun :: Hypo -> Exp -> Tree 'Exp_
    BVar :: QIdent -> Tree 'Bind_
    BTyped :: QIdent -> Exp -> Tree 'Bind_
    PBVar :: QIdent -> Tree 'Pattbind_
    PBTyped :: QIdent -> Exp -> Tree 'Pattbind_
    HExp :: Exp -> Tree 'Hypo_
    HVarExp :: QIdent -> Exp -> Tree 'Hypo_
    HParVarExp :: QIdent -> Exp -> Tree 'Hypo_
    PVar :: QIdent -> Tree 'Patt_
    PBracket :: Patt -> Tree 'Patt_
    PApp :: Patt -> Patt -> Tree 'Patt_
    PBind :: Bind -> Patt -> Tree 'Patt_
    QIdent :: String -> Tree 'QIdent_
-- this is all
```

```
-- MathCore.gf

abstract MathCore =
  Terms, UserConstants

t ;
t ;
-> Jmt ;

mt ;

  NotProp : Prop -> Prop ;
  -- plus more functions
```

```
module Informath where

data Tree :: * -> * where
  GAndAdj :: GListAdj -> Tree GAdj_
  GComparAdj :: GCompar -> GExp -> Tree GAdj_
  GOrAdj :: GListAdj -> Tree GAdj_
  GReladjAdj :: GReladj -> GExp -> Tree GAdj_
  LexAdj :: String -> Tree GAdj_
  GIdentsArgKind :: GKind -> GListIdent -> Tree GArgKind_
  GKindArgKind :: GKind -> Tree GArgKind_
  LexCompar :: String -> Tree GCompar_
  LexComparnoun :: String -> Tree GComparnoun_
  LexConst :: String -> Tree GConst_
  GComparEqsign :: GCompar -> Tree GEqsign_
  GComparnounEqsign :: GComparnoun -> Tree GEqsign_
  GEBinary :: GEqsign -> GTerm -> GTerm -> Tree GEquation_
  GAbsExp :: GListIdent -> GExp -> Tree GExp_
  GAllIdentsKindExp :: GListIdent -> GKind -> Tree GExp_
  GAllKindExp :: GKind -> Tree GExp_
  GAndExp :: GListExp -> Tree GExp_
  GAppExp :: GExp -> GExps -> Tree GExp_
  GCoercionExp :: GCoercion -> GExp -> Tree GExp_
  GConstExp :: GConst -> Tree GExp_
  GEveryIdentKindExp :: GIdent -> GKind -> Tree GExp_
  GEveryKindExp :: GKind -> Tree GExp_
  GFunListExp :: GFun -> GExps -> Tree GExp_
  GIndefIdentKindExp :: GIdent -> GKind -> Tree GExp_
  GIndefKindExp :: GKind -> Tree GExp_
  GIndexedTermExp :: GInt -> Tree GExp_

-- plus quite a few more
```

```
module Dedukti2Core where

import Dedukti.AbsDedukti
import Informath
import DeduktiOperations

jmt2jmt :: Jmt -> GJmt
jmt2jmt jmt = case jmt of
  JDef ident (MTExp typ) meexp ->
    let mexp = case meexp of
          MEExp exp -> Just exp
          _ -> Nothing
    in case (splitType typ, guessCat ident typ) of
      ((hypos, kind), c) | elem c ["Noun", "Set"] ->
          (maybe (GAxiomKindJmt axiomLabel)
               (\exp x y -> GDefKindJmt definitionLabel x y (exp2kind exp)) mexp)
            (GListHypo (hypos2hypos hypos))
            (ident2kind ident)
      ((hypos, kind), c) | elem c ["Name", "Const", "Unknown"] ->
          (maybe (GAxiomExpJmt axiomLabel)
                 (\exp x y z -> GDefExpJmt definitionLabel x y z (exp2exp exp)) mexp)
            (GListHypo (hypos2hypos hypos))
            (ident2exp ident)
            (exp2kind kind)

…

exp2kind :: Exp -> GKind

exp2prop :: Exp -> GProp

exp2exp :: Exp -> GExp

exp2proof :: Exp -> GProof
```

```
-- Dedukti.bnf

MJmts. Module ::= [Jmt]

terminator

comment "(
comment "#

JStatic.
JDef.
JInj.
JThm.
JRules.

RRule.  Ru
separator

separator

MTNone. MT
MTExp.  MT

MENone. ME
MEExp.  ME

EIdent.  E
EApp.    E
EAbs.    E
EFun.    E

coercions

-- plus so

token QIde
('.' (letter | digit |
```

```
module AbsDe

data Tree (a
    MJmts ::
    JStatic
    JDef ::
    JInj ::
    JThm ::
    JRules :
    RRule ::
    MTNone :
    MTExp ::
    MENone :
    MEExp ::
    EIdent :
    EApp ::
    EAbs ::
    EFun ::
    BVar ::
    BTyped :
    PBVar ::
    PBTyped
    HExp ::
    HVarExp
    HParVarE
    PVar ::
    PBracket
    PApp ::
    PBind ::
    QIdent :
```

```
GAdj_

GAdj_

-> Tree GArgKind_
_

rnoun_

gn_
ee GEqsign_
-> Tree GEquation_
GExp_
nd -> Tree GExp_


ree GExp_

-> Tree GExp_

Exp_
-> Tree GExp_
```

```
t ;
t ;
 -> Jmt ;


mt ;
```

Dedukti Exp	GF category	linearization	linguistic category
`union A B`	`Exp`	*the union of A and B*	noun phrase
`Nat`	`Kind`	*natural number*	common noun
`divisible 9 3`	`Prop`	*9 is divisible by 3*	sentence
`oddS 0 evenZ`	`Proof`	*0 is even. Therefore 1 is odd.*	text

```
-- Dedukti.bnf

MJmts. Module ::= [Jmt]

terminator

comment "(
comment "#

JStatic.
JDef.
JInj.
JThm.
JRules.

RRule.  Ru
separator

separator

MTNone. MT
MTExp.  MT

MENone. ME
MEExp.  ME

EIdent.  E
EApp.    E
EAbs.    E
EFun.    E

coercions

-- plus so

token QIde
('.' (letter | digit |
```

```
module AbsDe

data Tree (a
    MJmts ::
    JStatic
    JDef ::
    JInj ::
    JThm ::
    JRules :
    RRule ::
    MTNone :
    MTExp ::
    MENone :
    MEExp ::
    EIdent :
    EApp ::
    EAbs ::
    EFun ::
    BVar ::
    BTyped :
    PBVar ::
    PBTyped
    HExp ::
    HVarExp
    HParVarE
    PVar ::
    PBracket
    PApp ::
    PBind ::
    QIdent :
```

```
module Dedukti2Core where

impo
impo
impo

jmt2
jmt2
  JD
    (
  JD

...

exp2
exp2
exp2
exp2

iden
iden
  QI
```

```
e GArgKind_

sign_                 t ;
e GEquation_          t ;
                       -> Jmt ;
Tree GExp_

                      mt ;

xp_

e GExp_

e GExp_
```

```
module Core2Dedukti where

import Dedukti.AbsDedukti
import Informath
import DeduktiOperations

prop2dedukti :: GProp -> Exp
prop2dedukti prop = case prop of
  GProofProp p -> EApp (EIdent (QIdent "Proof")) (prop2dedukti p)
  GFalseProp -> propFalse
  GIdentProp ident -> EIdent (ident2ident ident)
  GAndProp (GListProp props) -> foldl1 propAnd (map prop2dedukti props)

kind2dedukti :: GKind -> Exp
kind2dedukti kind = case kind of
  GElemKind k -> EApp (EIdent (QIdent "Elem")) (kind2dedukti k)
  GTermKind (GTIdent ident) -> EIdent (ident2ident ident)
  GSetKind (LexSet s) -> EIdent (QIdent (s))

exp2dedukti :: GExp -> Exp
exp2dedukti exp = case exp of
  GTermExp (GTNumber (GInt n)) -> int2exp n
  GTermExp (GTIdent ident) -> EIdent (ident2ident ident)
  GAppExp exp exps ->
    foldl1 EApp (map exp2dedukti (exp : (exps2list exps)))
  GAbsExp (GListIdent idents) exp ->
    foldr
      (\x y -> EAbs (BVar (ident2ident x)) y)
      (exp2dedukti exp)
      idents
```

Dealing with identifiers

```
-- BaseConstants.dk

Set : Type.
Prop : Type.

(; ignored in Dedukti2Core ;)
Elem : Set -> Type.
Proof : Prop -> Type.

(; logical operators, hard-coded in MathCore ;)
false : Prop.
and : (A : Prop) -> (B : Prop) -> Prop.
or : (A : Prop) -> (B : Prop) -> Prop.
if : Prop -> Prop -> Prop.
forall : (A : Set) -> (Elem A -> Prop) -> Prop.
exists : (A : Set) -> (Elem A -> Prop) -> Prop.

def not : Prop -> Prop := A => if A false.
def iff : Prop -> Prop -> Prop :=
  A => B => and (if A B) (if B A).

(; constants defined in a lexicon ;)

def Nat : Set := Num.
def Int : Set := Num.
def Rat : Set := Num.
def Real : Set := Num.

Eq : Elem Real -> Elem Real -> Prop.
Lt : Elem Real -> Elem Real -> Prop.
Gt : Elem Real -> Elem Real -> Prop.
Neq : Elem Real -> Elem Real -> Prop.
Leq : Elem Real -> Elem Real -> Prop.
Geq : Elem Real -> Elem Real -> Prop.

plus : (x : Elem Real) -> (y : Elem Real) -> Elem Real.
minus : Elem Real -> Elem Real -> Elem Real.
times : Elem Real -> Elem Real -> Elem Real.
```

```
(; BaseConstants.dk ;)

Set : Type.
Prop : Type.

(; ignored in Dedukti2Core ;)
Elem : Set -> Type.
Proof : Prop -> Type.

(; logical operators, hard-coded in MathCore ;)
false : Prop.
and : (A : Prop) -> (B : Prop) -> Prop.
or : (A : Prop) -> (B : Prop) -> Prop.
if : Prop -> Prop -> Prop.
forall : (A : Set) -> (Elem A -> Prop) -> Prop.
exists : (A : Set) -> (Elem A -> Prop) -> Prop.

def not : Prop -> Prop := A => if A false.
def iff : Prop -> Prop -> Prop :=
  A => B => and (if A B) (if B A).

(; constants defined in a lexicon ;)

Nat : Set.
Int : Set.
Rat : Set.
Real : Set.

Eq : Elem Real -> Elem Real -> Prop.
Lt : Elem Real -> Elem Real -> Prop.
Gt : Elem Real -> Elem Real -> Prop.

even : Elem Int -> Prop.
def odd : Elem Int -> Prop := n => not (even n).

plus : (x : Elem Real) -> (y : Elem Real) -> Elem Real.
minus : Elem Real -> Elem Real -> Elem Real.
times : Elem Real -> Elem Real -> Elem Real.
```

```
# constant_data.dkgf

# for translating between Dedukti and GF abstract syntax

Nat BASE Set natural_Set
Int BASE Set integer_Set
Rat BASE Set rational_Set
Real BASE Set real_Set

Eq BASE Compar Eq_Compar
Lt BASE Compar Lt_Compar
Gt BASE Compar Gt_Compar

plus BASE Oper plus_Oper
minus BASE Oper minus_Oper
times BASE Oper times_Oper

even BASE Adj even_Adj
odd BASE Adj odd_Adj

# for generating GF linearization rules

#LIN Eng natural_Set = mkSet "N" "natural" number_N
#LIN Fre natural_Set = mkSet L.natural_Set "naturel" nombre_N
#LIN Swe natural_Set = mkSet L.natural_Set "naturlig" tal_N

#LIN Eng Lt_Compar = mkCompar "<" "less" "than"
#LIN Fre Lt_Compar = mkCompar "<" (mkAP (mkA "inférieur")) dative
#LIN Swe Lt_Compar = mkCompar "<" "mindre" "än"

#LIN Eng even_Adj = mkAdj "even"
#LIN Fre even_Adj = mkAdj "pair"
#LIN Swe even_Adj = mkAdj "jämn"

# for converting identifiers from third-party projects

le ALIAS matita Leq
```

```
abstract BaseConstants = {

-- GF cat         usage                            example
-----------------------------------------------------------------------
  Noun ;       -- Kind                          -- set
  Fam ;        -- Kind -> Kind                  -- list of integers
  Adj ;        -- Exp -> Prop                   -- even
  Verb ;       -- Exp -> Exp                    -- converge
  Reladj ;     -- Exp -> Exp -> Prop            -- divisible by
  Relverb ;    -- Exp -> Exp -> Prop            -- divide
  Relnoun ;    -- Exp -> Exp -> Prop            -- root of
  Name ;       -- Exp                           -- contradiction
  Fun ;        -- [Exp] -> Exp                  -- radius of
  Label ;      -- Exp                           -- theorem 1

  Set ;        -- Kind | Term                   -- integer, Z
  Const ;      -- Exp | Term                    -- the empty set, Ø
  Oper ;       -- Exp -> Exp -> Exp   | Term    -- the sum of, +
  Compar ;     -- Exp -> Exp -> Prop | Formula -- greater than, >
  Comparnoun ; -- Exp -> Exp -> Prop | Formula -- a subset of, \sub
```

```
def sphenic : Nat -> Prop
  := ...
(; GF: sphenic number ;)
```

lexical rule extraction

```
# from Wikidata

{"Q638185": {
  "pl": "Liczby sfeniczne",
  "de": "sphenische Zahl",
  "en": "sphenic number",
  "es": "número esfénico",
  "fr": "nombre sphénique",
  "zh": "楔形数",
  "sv": "sfeniskt tal",
  "ta": "ஸ்ஃபீனிக் எண்",
  }
}
```

```
sphenic NEW number_theory Adj spenic_Adj

#LIN Eng sphenic_Adj = mkAdj "sphenic"
#LIN Fre sphenic_Adj = mkAdj "sphénique"
#LIN Swe sphenic_Adj = mkAdj "sfenisk"
```

AR, Building Grammar Libraries for Mathematics and Avoiding Manual Work.. Presentation at Hausdorff Center for Mathematics, 2024, https://www.youtube.com/watch?v=EQ-k_JQ7fDM&t=5s

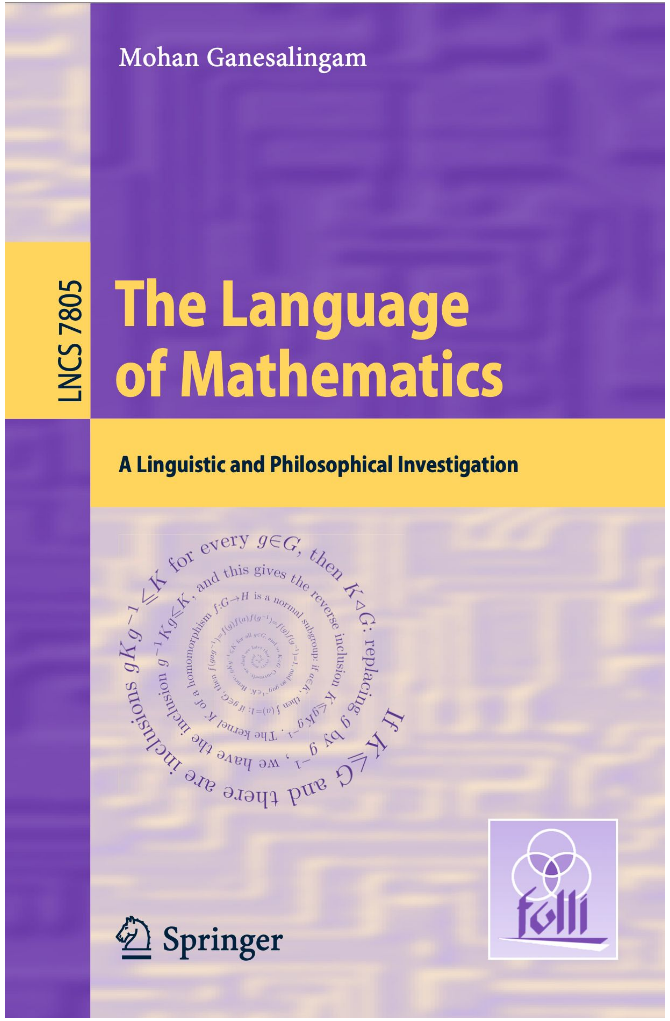
From MathCore to full Informath

has

natural language content that ~~lacks~~ the inherent diversity and flexibility in expression: they are ~~rigid and not~~ natural-language-like.

has

natural language content that ~~lacks~~ the inherent diversity and flexibility in expression: they are ~~rigid and not~~ natural-language-like.

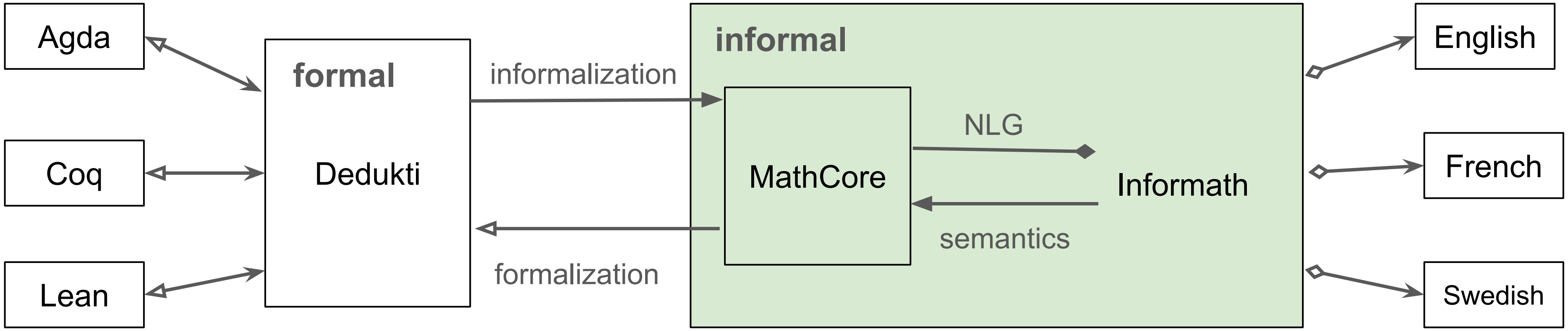

Mohan Ganesalingam

LNCS 7805

The Language of Mathematics

A Linguistic and Philosophical Investigation

Springer



	to one	to many
total	➝	─◆
partial	─▷	─◇

```
abstract Informath = MathCore ** {

cat
  [Adj] {2} ;
  [Exp] {2} ;

fun
-- to use symbolic expressions whenever possible
  FormulaProp : Formula -> Prop ;
  SetTerm : Set -> Term ;
  ConstTerm : Const -> Term ;
  ComparEqsign : Compar -> Eqsign ;

-- to remove parentheses around complex propositions
  SimpleAndProp : [Prop] -> Prop ;

-- to aggregate adjectives and expressions
  AndAdj : [Adj] -> Adj ;
  OrAdj : [Adj] -> Adj ;

  AndExp : [Exp] -> Exp ;
  OrExp : [Exp] -> Exp ;

-- in situ quantifiers

  AllKindExp : Kind -> Exp ;
  AllIdentsKindExp : [Ident] -> Kind -> Exp ;

  SomeKindExp : Kind -> Exp ;
  SomeIdentsKindExp : [Ident] -> Kind -> Exp ;

  NoIdentsKindExp : [Ident] -> Kind -> Exp ;
  NoKindExp : Kind -> Exp ;

-- miscellaneous alternative expressions

  PostQuantProp : Prop -> Exp -> Prop ;
}
```

```
prop110 : (a : Elem Int) -> (c : Elem Int) ->
 Proof (and (odd a) (odd c)) -> Proof (forall
 Int (b => even (plus (times a b) (times b c)))).
```

Prop110. For all instances a and c of integers, if we can prove that a is odd and c is odd, then we can prove that for all integers b, the sum of the product of a and b and the product of b and c is even.

```
abstract Informath = MathCore ** {

cat
  [Adj] {2} ;
  [Exp] {2} ;

fun
-- to use symbolic expressions whenever possible
  FormulaProp : Formula -> Prop ;
  SetTerm : Set -> Term ;
  ConstTerm : Const -> Term ;
  ComparEqsign : Compar -> Eqsign ;

-- to remove parentheses around complex propositions
  SimpleAndProp : [Prop] -> Prop ;

-- to aggregate adjectives and expressions
  AndAdj : [Adj] -> Adj ;
  OrAdj : [Adj] -> Adj ;

  AndExp : [Exp] -> Exp ;
  OrExp : [Exp] -> Exp ;

-- in situ quantifiers

  AllKindExp : Kind -> Exp ;
  AllIdentsKindExp : [Ident] -> Kind -> Exp ;

  SomeKindExp : Kind -> Exp ;
  SomeIdentsKindExp : [Ident] -> Kind -> Exp ;

  NoIdentsKindExp : [Ident] -> Kind -> Exp ;
  NoKindExp : Kind -> Exp ;

-- miscellaneous alternative expressions

  PostQuantProp : Prop -> Exp -> Prop ;
}
```

```
prop110 : (a : Elem Int) -> (c : Elem Int) ->
 Proof (and (odd a) (odd c)) -> Proof (forall
 Int (b => even (plus (times a b) (times b c)))).
```

Prop110. For all instances a and c of integers, if we can prove that a is odd and c is odd, then we can prove that for all integers b, the sum of the product of a and b and the product of b and c is even.

Prop110. Let $a, c \in Z$. Assume that both a and c are odd. Then for all integers b, $ab + bc$ is even.

Prop110. Let $a, c \in Z$. Assume that both a and c are odd. Then $ab + bc$ is even for all integers b.

```
abstract Informath = MathCore ** {

cat
  [Adj] {2} ;
  [Exp] {2} ;

fun
-- to use symbolic expressions whenever possible
  FormulaProp : Formula -> Prop ;
  SetTerm : Set -> Term ;
  ConstTerm : Const -> Term ;
  ComparEqsign : Compar -> Eqsign ;

-- to remove parentheses around complex propositions
  SimpleAndProp : [Prop] -> Prop ;

-- to aggregate adjectives and expressions
  AndAdj : [Adj] -> Adj ;
  OrAdj : [Adj] -> Adj ;

  AndExp : [Exp] -> Exp ;
  OrExp : [Exp] -> Exp ;

-- in situ quantifiers

  AllKindExp : Kind -> Exp ;
  AllIdentsKindExp : [Ident] -> Kind -> Exp ;

  SomeKindExp : Kind -> Exp ;
  SomeIdentsKindExp : [Ident] -> Kind -> Exp ;

  NoIdentsKindExp : [Ident] -> Kind -> Exp ;
  NoKindExp : Kind -> Exp ;

-- miscellaneous alternative expressions

  PostQuantProp : Prop -> Exp -> Prop ;
}
```

```
prop50 : Proof (forall Nat
  (n => not (and (even n) (odd n)))).
```

Prop50. We can prove that for all natural numbers n, it is not the case that n is even and n is odd.

Prop50. For all natural numbers n, n is not both even and odd.

Prop50. No natural number n is both even and odd.

Prop50. No natural number is both even and odd.

```
module Core2Informath where

import Informath

nlg :: Opts -> Tree a -> [Tree a]
nlg opts tree = concatMap variations [t, ut, ft, aft, iaft, viaft]
 where
   t = unparenth tree
   ut = uncoerce t
   ft = formalize ut
   aft = aggregate (flatten ft)
   iaft = insitu aft
   viaft = varless iaft


insitu :: Tree a -> Tree a
insitu t = case t of
  GAllProp (GListArgKind [argkind]) (GAdjProp adj exp) -> case subst argkind exp of
    Just (x, kind) -> GAdjProp adj (GAllIdentsKindExp (GListIdent [x]) kind)
    _ -> t
  GAllProp (GListArgKind [argkind]) (GNotAdjProp adj exp) -> case subst argkind exp of
    Just (x, kind) -> GAdjProp adj (GNoIdentsKindExp (GListIdent [x]) kind)
    _ -> t
  GExistProp (GListArgKind [argkind]) (GAdjProp adj exp) -> case subst argkind exp of
    Just (x, kind) -> GAdjProp adj (GSomeIdentsKindExp (GListIdent [x]) kind)
    _ -> t
  _ -> composOp insitu t


varless :: Tree a -> Tree a
varless t = case t of
  GAllIdentsKindExp (GListIdent [_]) kind -> GAllKindExp kind
  GNoIdentsKindExp (GListIdent [_]) kind -> GNoKindExp kind
  GSomeIdentsKindExp (GListIdent [_]) kind -> GSomeKindExp kind
  _ -> composOp varless t
```

NLG (Natural Language Generation) is a combination of selected **almost compositional operations.**

Example: **in situ quantification** is possible if the variable x occurs exactly once in the body.

The variable can optionally be omitted.

B Bringert and A. Ranta, A pattern for almost compositional functions. Journal of Functional Programming 18 (5-6), 567-598, 2008.

```haskell
module Informath2Core where

import Informath

data SEnv = SEnv {varlist :: [String]}

initSEnv = SEnv {varlist = []}

newVar :: SEnv -> (GIdent, SEnv)

sem :: SEnv -> Tree a -> Tree a
sem env t = case t of

  GAdjProp (GAndAdj (GListAdj adjs)) x ->
    let sx = sem env x
    in GAndProp (GListProp [GAdjProp adj sx | adj <- adjs])

  GAdjProp adj (GEveryKindExp kind) ->
    let (x, env') = newVar env
    in sem env'
      (GAllProp (GListArgKind [GIdentsArgKind kind (GListIdent [x])])
        (GAdjProp adj (GTermExp (GTIdent x))))
```

The opposite direction is, again, simpler:

- deterministic conversion of Informath extensions to MathCore
- like logical semantics (since MathCore is an unambiguous syntax for logic)
- fresh variables must be created for varless in situ quantifiers

Order is important:

every number is even or odd

→ *for all numbers x, x is (even or odd)*
→ *for all numbers x, (x is even or x is odd)*

~~→ *every number is even or every number is odd*~~
~~→ *(for all numbers x, x is even) or (for all numbers x, x is odd)*~~

Demos

make

```
all: Informath.pgf Dedukti Agda Coq Lean RunInformath

Informath.pgf:
      cd grammars ; gf --make -output-format=haskell -haskell=lexical --haskell=gadt
-lexical=Name,Noun,Fam,Adj,Rel,Fun,Label,Const,Oper,Compar,Set,Coercion,Relverb,Relno
un,Reladj,Comparnoun,Verb --probs=Informath.probs InformathEng.gf InformathFre.gf
InformathSwe.gf

Dedukti:
      cd typetheory ; bnfc -m -p Dedukti --haskell-gadt Dedukti.bnf ; make
Agda:
      cd typetheory ; bnfc -m -p Agda --haskell-gadt Agda.bnf ; make
Lean:
      cd typetheory ; bnfc -m -p Lean --haskell-gadt Lean.bnf ; make
Coq:
      cd typetheory ; bnfc -m -p Coq --haskell-gadt Coq.bnf ; make

RunInformath:
      ghc -package gf RunInformath.hs
```

make demo

```
demo:
	./RunInformath -lang=Fre test/exx.dk
	./RunInformath -lang=Swe test/exx.dk
	./RunInformath -lang=Eng test/exx.dk
	./RunInformath -lang=Eng test/exx.dk >exx.txt
	./RunInformath -lang=Eng exx.txt
	./RunInformath -lang=Eng test/gflean-data.txt

	cat BaseConstants.dk test/exx.dk >bexx.dk
	dk check bexx.dk

	./RunInformath -to-agda test/exx.dk >exx.agda
	agda --prop exx.agda

	./RunInformath -to-coq test/exx.dk >exx.v
	cat BaseConstants.v exx.v >bexx.v

	coqc bexx.v
	./RunInformath -to-lean test/exx.dk >exx.lean

	cat BaseConstants.lean exx.lean >bexx.lean
	lean bexx.lean

	./RunInformath -to-latex-file -variations test/top100.dk >out/top100.tex
	echo "consider pdflatex out/top100.tex"

	./RunInformath -to-latex-file -variations test/sets.dk >out/sets.tex
	echo "consider pdflatex out/sets.tex"
```

Generating synthetic data

For those who are interested just in the generation of synthetic data, the following commands (after building Informath with `make`) can do it: assuming that you have a `.dk` file available, build a `.jsonl` file with all conversions of each Dedukti judgement:

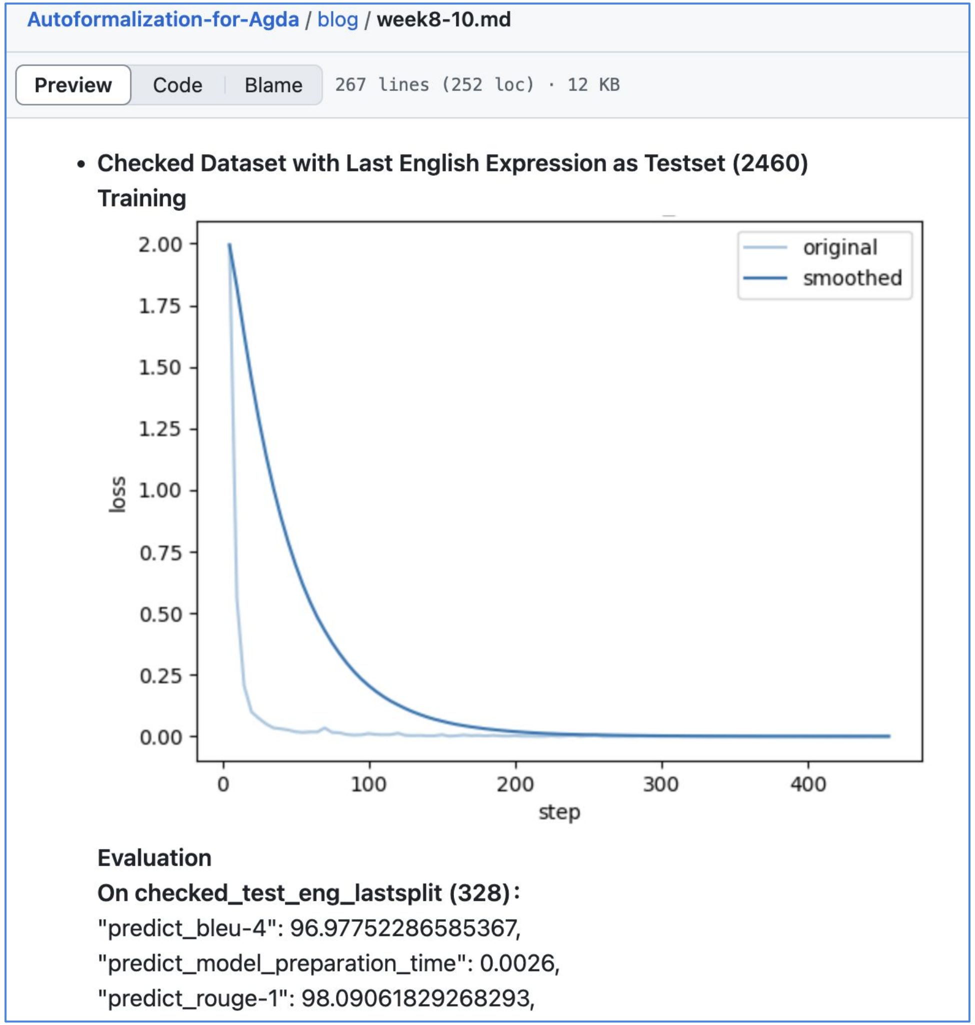
```
 $ ./RunInformath -parallel <file>.dk > <file>.jsonl
```

After that, select the desired formal and informal languages to generate a new `.jsonl` data with just those pairs:

```
 $ python3 test/jsonltest.py <file.jsonl> <formal> <informal>
```

The currently available values of `<formal>` and `<informal>` are the keys in `<file>.jsonl` - for example, `agda` and `InformathEng`, respectively.

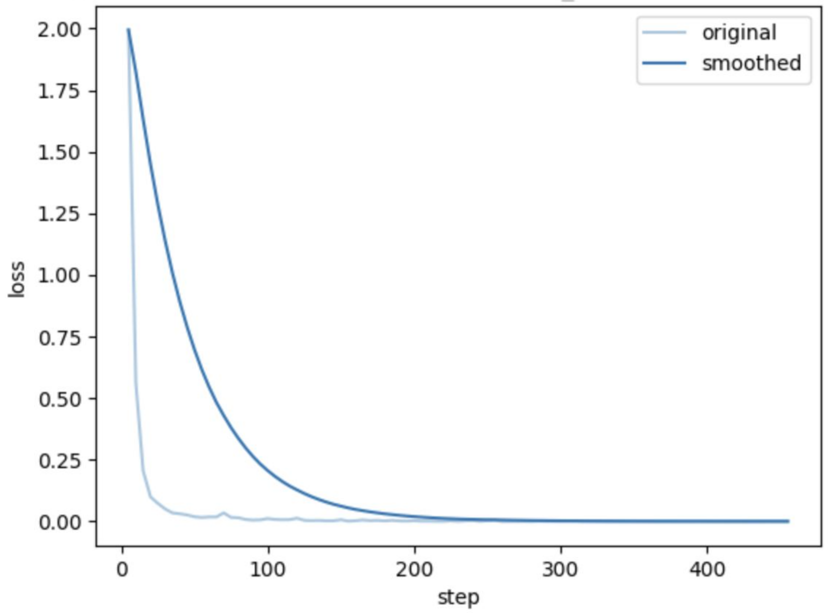
https://github.com/aarneranta/informath/


Autoformalization-for-Agda / blog / week8-10.md

Preview Code Blame 267 lines (252 loc) · 12 KB

- **Checked Dataset with Last English Expression as Testset (2460)**

  Training



Evaluation
On checked_test_eng_lastsplit (328):
"predict_bleu-4": 96.97752286585367,
"predict_model_preparation_time": 0.0026,
"predict_rouge-1": 98.09061829268293,

Pei Huang, MSc project at Chalmers, 2025

Conclusion

Symbolic informalization can be

- natural and fluent
    - by extending CNL towards the full language of mathematics

- feasible to develop
    - by Dedukti, GF, and rule extraction

- shared by different formal and informal languages
    - by Dedukti and GF interlinguas

- inverted to autoformalization
    - by fine-tuned LLM + feedback informalization

Questions to work on here

Collect Dedukti formalizations

- from Agda
- from Lean
- 100 theorems

Communicate with Dedukti in type checking

- choose from ambiguous parse results

Resolve hidden arguments

- by lambda-pi?
- overloading
- coercions, number types

FIN

https://github.com/aarneranta/informath