Mathematical Language from a Design Perspective

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

January 2018

Formal methods

"Formal methods" refers to the use of logic-based methods in computer science for

- specifying,
- designing, and
- verifying

complex hardware and software systems.

They rely on:

- formal languages
- formal semantics (truth)
- formal inferences (proof)

Formal methods in mathematics

The methods can also be used to specify mathematical objects, and to develop and verify mathematical proofs.

In mathematics, they can be used for:

- verified proof
- verified computation
- automated reasoning and discovery
- infrastructure for communication and search

Formally verified proof

Working with a proof assistant, users can construct formal axiomatic proofs.

Substantial theorems of mathematics have been verified in this way:

- the prime number theorem
- the Jordan curve theorem
- Dirichlet's theorem on primes in an arithmetic progression
- the central limit theorem

Formally verified proof

There are good libraries for

- elementary number theory
- real and complex analysis
- point-set topology
- measure theory and probability
- abstract algebra

In 2012, Georges Gonthier and 13 co-authors announced the verification of the Feit-Thompson Odd Order Theorem.

In 2014, Thomas Hales and 21 co-authors announced the verification of the Kepler conjecture.

Formally verified computation

The verification of the Kepler conjecture involved, in particular, verifying several hundred nonlinear inequalities.

Checking them all requires roughly 5000 processor hours on the Microsoft Azure cloud.

In 2002, Warwick Tucker solved Smale's 14th problem, proving the existence of the Lorenz attractor. The proof relied on very a careful computation of an enclosure of a Poincaré section.

Fabian Immler has recently verified that computation with Isabelle.

Formal methods for discovery

Paul Erdős asked whether it is possible to color the positive integers red and blue such that there is no monochromatic Pythagorean triple ($a^2 + b^2 = c^2$).

In May, 2016, Heule, Kullman, and Marek showed that it is possible to color the integers from 1 to 7,824 in this way, but not 1 to 7,825.

They used sophisticated SAT solver technology and customized heuristics.

They encoded the negative result in an independently-checkable proof format. The original proof was 200 terabytes long, but they managed to produce a 68 gigabyte *certificate*.

Formal infrastructure for communication and search

Formal methods open up new possibilities for storing, communicating, and indexing mathematical knowledge in very precise ways.

The *Formal Abstracts* project is a step in that direction.

Formal languages for mathematics

Formal methods can be used to support:

- verified proof
- verified computation
- automated reasoning and discovery
- infrastructure for communication and search

One thing these technologies all have in common is that they rely on formal modeling of mathematical language.

That is the topic of this talk.

Formal languages for mathematics

Formal languages can be used to specify:

- mathematical data types:

  $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{R} \to \mathbb{R}$, groups, rings, topological spaces, ...
- mathematical objects:

  1, π, $2 + 5i$, $f : x \mapsto x^2$, $\mathbb{Z}/7\mathbb{Z}$, ...
- mathematical assertions:

  "7 is an odd prime," Fermat's last theorem
- mathematical proofs

as well as algorithms, procedures, heuristics, notation, conventions, and so on.

I will focus on the first three.

Assertion languages

```
theorem PrimeNumberTheorem:
  "(%n. pi n * ln (real n) / (real n)) ----> 1"

!C. simple_closed_curve top2 C ==>
  (?A B. top2 A /\ top2 B /\
    connected top2 A /\ connected top2 B /\
  ~(A = EMPTY) /\ ~(B = EMPTY) /\
   (A INTER B = EMPTY) /\ (A INTER C = EMPTY) /\
       (B INTER C = EMPTY) /\
      (A UNION B UNION C = euclid 2)

!d k. 1 <= d /\ coprime(k,d)
  ==> INFINITE { p | prime p /\ (p == k) (mod d) }
```

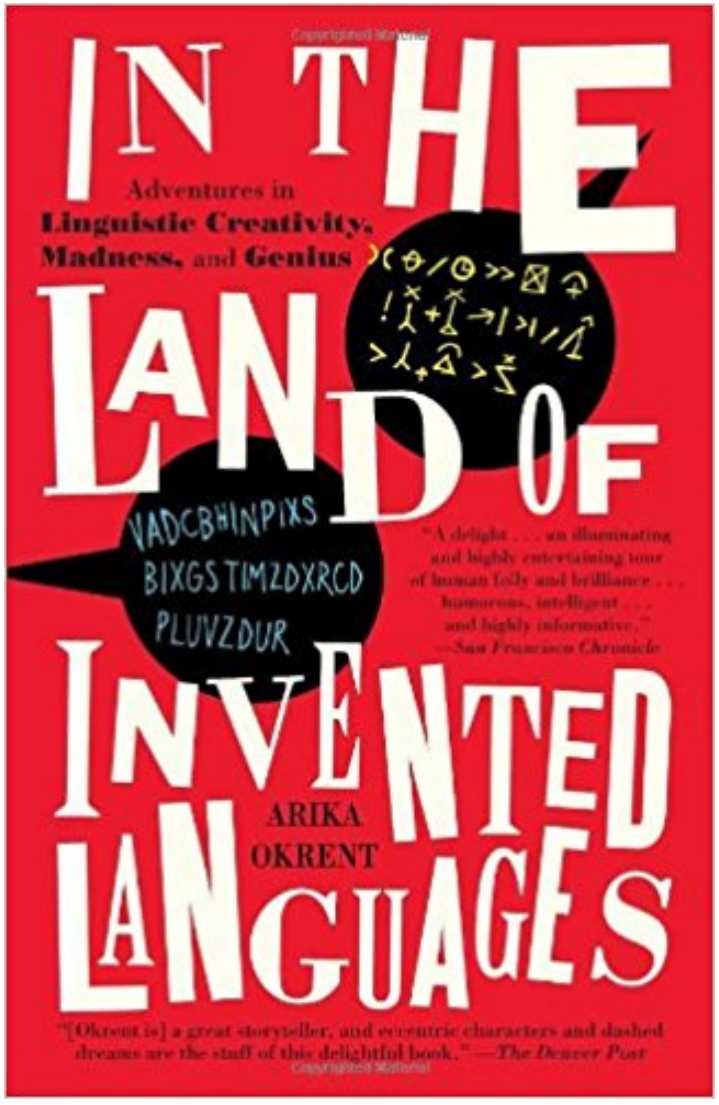
Assertion languages

```
Theorem Sylow's_theorem :
  [/\ forall P,
      [max P | p.-subgroup(G) P] = p.-Sylow(G) P,
      [transitive G, on 'Syl_p(G) | 'JG],
      forall P, p.-Sylow(G) P ->
        #|'Syl_p(G)| = #|G : 'N_G(P)|
   &  prime p -> #|'Syl_p(G)| %% p = 1%N].

Theorem Feit_Thompson (gT : finGroupType)
   (G : {group gT}) :
  odd #|G| → solvable G.

Theorem simple_odd_group_prime (gT : finGroupType)
    (G : {group gT}) :
  odd #|G| → simple G → prime #|G|.
```

Assertion languages

theorem (**in** *prob_space*) *central_limit_theorem*:
  fixes
    X :: "*nat* $\Rightarrow$ 'a $\Rightarrow$ *real*" **and**
    μ :: "*real measure*" **and**
    σ :: *real* **and**
    S :: "*nat* $\Rightarrow$ 'a $\Rightarrow$ *real*"
  assumes
    X_indep: "*indep_vars* (λ*i*. *borel*) *X UNIV*" **and**
    X_integrable: "$\bigwedge$*n*. *integrable M* (*X n*)" **and**
    X_mean_0: "$\bigwedge$*n*. *expectation* (*X n*) = *0*" **and**
    σ*_pos*: "σ > *0*" **and**
    X_square_integrable: "$\bigwedge$*n*. *integrable M* (λ*x*. (*X n x*)^2)" **and**
    X_variance: "$\bigwedge$*n*. *variance* (*X n*) = σ^2" **and**
    X_distrib: "$\bigwedge$*n*. *distr M borel* (*X n*) = μ"
  defines
    "*S n* $\equiv$ λ*x*. $\sum$*i*<*n*. *X i x*"
  shows
    "*weak_conv_m* (λ*n*. *distr M borel* (λ*x*. *S n x* / *sqrt* (*n* * σ^2)))
      (*density lborel std_normal_density*)"

Inventing a language

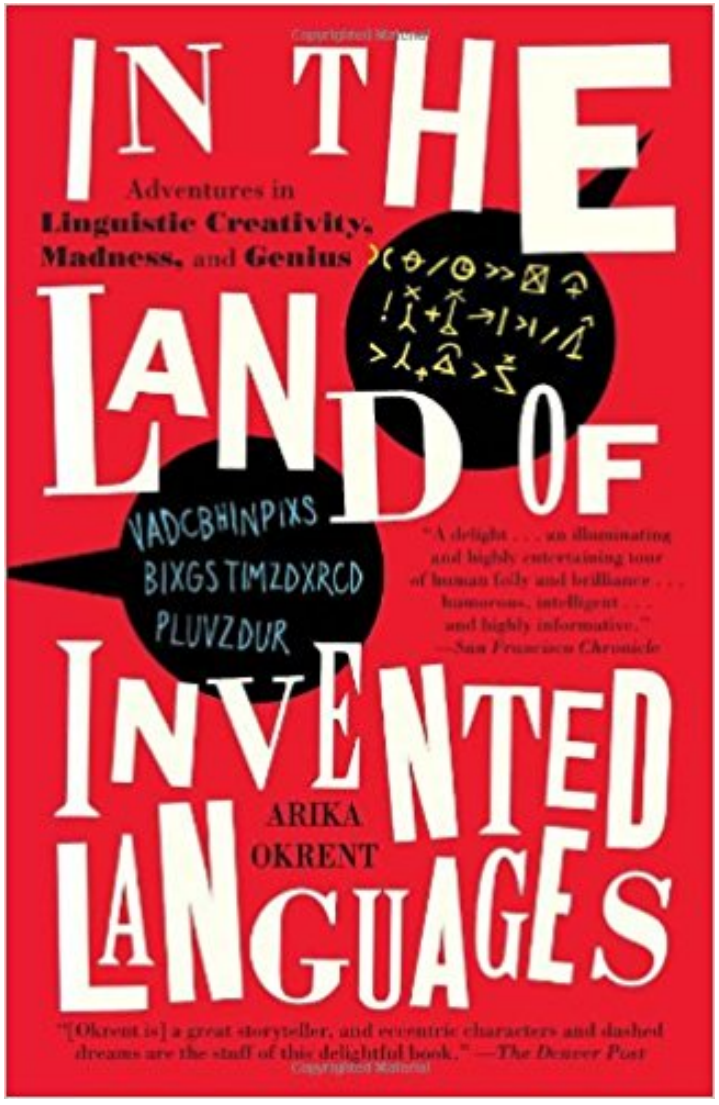


Arika Okrent has written a book about invented languages.

Morals:

- Inventing a language is exhilerating.
- Inventing a good one is harder than you think.

Inventing a language



Arika Okrent has written a book about invented languages.

Morals:

- Inventing a language is exhilerating.
- Inventing a good one is harder than you think.
- People who try tend to be crazy.

Some challenges

Consider the following mathematical statements:

> *For every* $x \in \mathbb{R}$, $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.
>
> *If* G *and* H *are groups and* f *is a homomorphism from* G *to* H, *then for every* $a, b \in G$, $f(a \cdot b) = f(a) \cdot f(b)$.
>
> *If* F *is a field of characteristic* p *and* $a, b \in F$, *then* $(a + b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p$.

How do we parse these?

Some challenges

Observations:

1. The index of the summation is over the natural numbers.
2. $\mathbb{N}$ is embedded in $\mathbb{R}$.
3. In "$a \in G$," G really means the underlying set.
4. ab means multiplication in the relevant group.
5. p is a natural number (in fact, a prime).
6. The summation operator make sense for any monoid (written additively).
7. The summation enjoys extra properties if the monoid is commutative.
8. The additive part of any field is so.
9. $\mathbb{N}$ is also embedded in any field.
10. Alternatively, any abelian group is a $\mathbb{Z}$-module.

Challange #1: Overloading and ambiguity of notation

Notation can be ambiguous:

- The meaning of $\cdot$ in $x \cdot y$ depends on x, y, and context.
- The number 5 can be an integer, a complex number, an element of $\mathbb{Z}/7\mathbb{Z}$, and element of an arbitrary ring.
- If A is a finite set, the cardinality of A, $|A|$, is a natural number. In general, it a a set-theoretic object, a cardinal number.
- $\int f$ may denote the Riemann integral, the Lebesgue integral, integral with respect to some Haar measure, etc.
- $gcd(x, y)$ might be the greatest common divisor on $\mathbb{N}$ or $\mathbb{Z}$, an element up to unit in a UFD, or an ideal in a ring of algebraic integers.

Challange #2: Ambiguity of domain

Intended domains are sometimes left implicit:

- For any n, if $3 < n < 5$, then n is even.
- $3 < \pi < 5$.

We sometimes think of integers as reals, sometimes as embedded in the reals.

Sometimes we identify $\mathbb{S}$ with $[0, 1)$ and $\mathbb{R}/\mathbb{Z}$, and at other times with $[0, 2\pi)$ and $\{z \in \mathbb{C} \mid |z| = 1\}$.

We also identify $\mathbb{R}^1$ with $\mathbb{R}$ and 1×1 matrices with scalars.

Challange #3: Mild abuses and conventions

These are slight abuses:

- If G is a group, we write $x \in G$ instead of $x \in \mathit{carrier}(G)$.
- If F is a functor between categories $\mathcal{C}$ and $\mathcal{D}$, we write both $F(C)$ and $F(f)$.
- We think of elements $f \in L^1$ as functions, but also equivalence classes up to a.e. equivalence.

Challenge #4: Partial functions

We are fast and loose about partially defined functions:

$$\lim_{n\to\infty} \frac{\pi(n)\log n}{n} = 1$$

Suppose

$$f = \begin{cases} 0 & \text{if } 0 \le x \le 1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Is $\lim_{x\to 0} f(x) = 0$?

Challenge #5: Algebraic relationships

Relationships between structures:

- subclasses: every abelian group is a group
- reducts: the additive part of a ring is an abelian group
- instances: the integers are an ordered ring
- embedding: the integers are embedded in the reals
- uniform constructions: the automorphisms of a field form a group

A language for mathematics has to recognize these relationships:

- reuse notation: 0, $a + b$, $a \cdot b$
- reuse definitions: $\sum_{i \in I} a_i$
- reuse facts: e.g. $\sum_{i \in I} c \cdot a_i = c \cdot \sum_{i \in I} a_i$

Challenge #5: Algebraic relationships

Of all the challenges, this is probably the most important.

Mathematics requires generality and abstraction, which, in turn, requires modular reasoning.

This means screening off irrelevant details, assumptions, and information.

When mathematics is working well, the language makes the things that matter explicit, and hides the things that don't.

Dependencies are carefully managed.

Challenge #6: High standards

A formal language for mathematics has to be:

- *precise*:
  - Every expression has to have a clear meaning.
  - The semantic interpretation (rules of inference, set-theoretic interpretation) has to be explicit.
- *expressive*:
  - We want to express everything.
  - It has to be convenient.
- *robust*:
  - It has to cover unexpected use cases.
  - It has to scale to complex definitions and use cases.
  - It has to work virtually all the time.
  - Error messages need to be informative.
  - Recovery has to be straightforward.

Mathematicians have strong opinions.

Summary

Challenges:

- overloading and ambiguity of notation
- ambiguity of domain
- mild abuses and conventions
- partial functions
- algebraic relationships
  - relationships between classes of structures
  - instances of structures
  - constructions of structures
  - embeddings and identifications
- high standards
  - precision
  - expressiveness
  - robustness

These pose design constraints.

Solutions

There is a substantial literature on how to address these issues (building on the literature on programming languages).

We have made a lot of progress, but we are still experimenting with different approaches.

Some ideas:

1. Use types.
2. Infer implicit information.
3. Use coercions.
4. Cope with partiality.
5. Infer algebraic relationships.
6. Drink beer.

Idea #1: Use types

In set theory, everything in a set: numbers, functions, tuples, triangles, groups, measures, ...

But it helps to keep track of what *types* of objects we are dealing with:

- The meaning of $x \cdot y$ can be inferred from the types of x and y.
- The meaning of $\sum_{x \in A} f(x)$ can be inferred from the types of A and f.
- The expression $gcd(5, f)$ is probably a typographical error.

Idea #1: Use types

In a typed framework, every expression has a *type*.

```
variables m n : ℤ
variables x y : ℝ
variables w z : ℂ
variable  R   : Ring
variables a b : R.carrier

#check m * 7 + n
#check x * 7 + y
#check w * 7 + z
#check a * 7 + b
```

Idea #1: Use types

In simple type theory, *type constructors* build compound types:

```
variables m n : ℕ
variable  b   : bool
variable  f   : ℕ → ℕ
variable  g   : ℕ × ℕ → bool
variable  F   : (ℕ → ℕ) → ℕ

#check f m
#check g (m, n)
#check F f
```

Idea #1: Use types

Systems generally allow *polymorphic* constructions over types:

```
variable  α      : Type
variables a b    : α
variables s t    : list α
variables m n    : ℕ
variables l₁ l₂ : list ℕ

#check a :: s ++ t
#check m :: l₁ ++ l₂
```

Idea #1: Use types

In dependent type theory, types can depend on parameters:

```
variables m n : ℕ
variables v   : vector ℝ 2
variable  w   : vector ℝ 3
variable  u   : vector ℝ (m^2 + 1)
variable  M   : matrix ℝ 3 2
variable  K   : matrix ℝ (m + 1) (n + 2)
```

This is useful with structures:

```
variable    R : Ring
variables a b : Ring.carrier R
```

Idea #1: Use types

Dependent types complicate things:

- A type checker needs to determine whether an expression has a given type.
- A type can depend on *any* expression.

```
def m := if goldbach_conjecture then 3 else 4

variable s : vector ℝ 3
variable t : vector ℝ m

#check s + t
```

Moral: dependent types can be useful, but they should be used wisely and sparingly.

Idea #2: Infer implicit information

Some information about an expression can be inferred from context.

For example, we have

```
cons : Π α : Type, α → list α → list α
```

Given `a : α` and `l : list α`, we can write `cons α a l`.

We can make the first argument implicit,

```
cons : Π {α : Type}, α → list α → list α
```

and write `cons a l` or `a :: l`.

Idea #2: Infer implicit information

Ordinarly, *lots* of information is left implicit.

In Lean, `2 + 2 = 4` is interpreted as

```
@eq.{1} nat
(@has_add.add.{0} nat nat.has_add (@bit0.{0} nat nat.has_add
  (@has_one.one.{0} nat nat.has_one))
   (@bit0.{0} nat nat.has_add (@has_one.one.{0} nat
  nat.has_one)))
(@bit0.{0} nat nat.has_add (@bit0.{0} nat nat.has_add
  (@has_one.one.{0} nat nat.has_one)))
```

The process of going from user input to a fully explicit term is known as *elaboration*.

In the other direction, the *pretty-printer* ordinarily prints `2 + 2 = 4`.

Idea #3: Use coercions

When the inferred type of an argument does not match its expected type, the system can insert a *coercion*.

For exmample, given `x : real` and `i : int`, the system can interpret

```
x + i
```

as

```
x + real.of_int i.
```

The pretty-printer can omit the coercion.

Similarly, we can write

- `g : G` instead of `g : Group.carrier G` with `G : Group`
- `F C` instead of `Functor.obj F C` with `F : Functor CC DD`

Idea #4: Cope with partiality

There are various ways to model partial functions and undefined values.

- Use option types:

```
def f (x : ℝ) : option ℝ :=
if 0 ≤ x then some x else none
```

- Use subtypes:

```
def f (x : {y : ℝ // y ≥ 0}) : ℝ := subtype.val x
```

- Use preconditions:

```
def f (x : ℝ) (h : x ≥ 0) : ℝ := x
```

Idea #4: Cope with partiality

- Use relations:

```
converges_to s l
has_deriv f x y
has_integral f S r
```

- Totalize.

  For example, it is convenient to "define" $x/0$ to be 0.

Idea #5: Infer algebraic relationships

Users can instruct the system to associate suitable structures to types.

One common method is known as *type class inference*.

Take an expression `x * y` to be notation for

```
mul _ _ x y
```

where `mul` has type

```
Π (α : Type) [m : has_mul α], α → α → α
```

This is a very general and flexible mechanism.

Idea #5: Infer algebraic relationships

```
class semigroup (α : Type u) extends has_mul α :=
(mul_assoc : ∀ a b c, a * b * c = a * (b * c))

class monoid (α : Type u) extends semigroup α, has_one α :=
(one_mul : ∀ a, 1 * a = a) (mul_one : ∀ a, a * 1 = a)

def pow {α : Type u} [monoid α] (a : α) : ℕ → α
| 0     := 1
| (n+1) := a * pow n

infix `^`:= pow
```

Idea #5: Infer algebraic relationships

```
@[simp] theorem pow_zero (a : α) : a^0 = 1 := by unfold pow

theorem pow_succ (a : α) (n : ℕ)  : a^(n+1) = a * a^n :=
by unfold pow

theorem pow_mul_comm' (a : α) (n : ℕ) : a^n * a = a * a^n :=
by induction n with n ih; simp [*, pow_succ]

theorem pow_succ' (a : α) (n : ℕ) : a^(n+1) = a^n * a :=
by simp [pow_succ, pow_mul_comm']

theorem pow_add (a : α) (m n : ℕ) : a^(m + n) = a^m * a^n :=
by induction n; simp [*, pow_succ', nat.add_succ]

theorem pow_mul_comm (a : α) (m n : ℕ) :
  a^m * a^n = a^n * a^m :=
by simp [(pow_add a m n).symm, (pow_add a n m).symm]

instance : linear_ordered_comm_ring int := ...
```

Idea #6: Drink beer

Engineering problems often have no perfect solution.

Design decisions have advantages and disadvantages.

Try to

- find clever solutions,
- maximize usability,
- minimize the drawbacks,
- revisit decisions in light of experience,
- and, sometimes, live with tradeoffs.

Summary

Challenges:

- overloading and ambiguity of notation
- ambiguity of domain
- mild abuses and conventions
- partial functions
- algebraic relationships
  - relationships between classes of structures
  - instances of structures
  - constructions of structures
  - embeddings and identifications
- high standards
  - precision
  - expressiveness
  - robustness

Summary

Solutions:

- Use types.
- Infer implicit information.
- Use coercions.
- Cope with partiality.
- Infer algebraic relationships.
- Drink beer.

Conclusions

Formal methods can be used for:

- verified proof
- verified computation
- automated reasoning and discovery
- infrastructure for communication and search

These are powerful tools for mathematics.

They extend the boundaries of what we can come to know.

They depend on having good formal languages.

Conclusions

With effort, we can faithfully model important aspects of mathematical language.

This is fundamentally an engineering problem.

But it requires careful thought, experimentation, and a thorough understanding of how mathematical language works.

A public service announcement

We should not leave the task to computer scientists alone.

It is not their job to make mathematicians happy.

We don't want our mathematics to look like hardware and software verification.

The mathematical community needs to take ownership.

A public service announcement

In mathematics, senior faculty generally do not have time to invest in developing a new technology.

Students and junior faculty would be foolish to do so if it will harm their careers.

Suggestions:

- Give junior faculty credit for experimenting with formal methods.
- Accept incremental progress.
- Give credit for publications in *Interactive Theorem Proving* and the *Journal of Automated Reasoning*.
- Hire young people to work in formal methods.

Let's make sure it is done right.