Type Theory and Practical Foundations

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

February 2016

Digitized mathematics

Mathematical artifacts in digital form:

- definitions
- assertions
- proofs
- algorithms

Maybe also heuristics, intuitions, conjectures, open problems, questions, . . .

Digitized mathematics

Goals: computational support for

- exploration and discovery
- reasoning
- verification
- calculation
- search
- education

Where we are

Arrived:

- numerical computation
- symbolic computation
- web search
- mathematical typesetting (TEX, LATEX)
- specialized mathematical software

Arriving:

- formal verification in computer science
- formally verified mathematical proofs
- formally verified mathematical programs and results

Where we are

Formal methods are gaining traction in industry:

- Model checking is widely used.
- Intel and AMD use formal methods to verify processors.
- The CompCert project verified a C compiler.
- The seL4 microkernel has been verified in Isabelle.
- The NSF has just funded a multi-year, multi-institution *DeepSpec* initiative.

Interactive theorem proving is central to the last four.

Where we are

There are a number of theorem provers in active use: Mizar, ACL2, HOL4, HOL Light, Isabelle, Coq, Agda, NuPrl, PVS, Lean, . . .

Substantial theorems have been verified:

- the prime number theorem
- the four-color theorem
- the Jordan curve theorem
- Gödel's first and second incompleteness theorems
- Dirichlet's theorem on primes in an arithmetic progression
- the Cartan fixed-point theorems
- the central limit theorem

Where we are

There are good libraries for

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

High-profile projects:

- the Feit-Thompson theorem (Gonthier et al.)
- The Kepler conjecture (Hales et al.)
- Homotopy type theory (Voevodsky, Awodey, Shulman, Coquand, et al.)

Where we are

Theorem provers, decision procedures, and search procedures are used in computer science:

- fast satisfiability solvers
- first-order theorem provers (resolution, tableaux)
- equational reasoning systems
- constraint solvers
- SMT solvers
- ...

These have had notably little impact on mathematics.

We need to learn how to take advantage of these.

The importance of foundations

We can make mistakes:

- in our assertions
- in our proofs
- in our algorithms and software
- in our automated reasoning systems

But without a semantics, we are “not even wrong.”

The importance of foundations

Model-theoretic semantics:

- $\mathbb{N}$ denotes the natural numbers
- "$2 + 2 = 4$" means that $2 + 2 = 4$

Leads to regress.

In ordinary mathematics, we share and implicit understanding of what the correct rules of reasoning are.

Foundational "semantics":

- Give an explicit grammar for making assertions.
- Give explicit rules for proving them.

The importance of foundations

Many take set theory to be the official foundation of mathematics.

There is also a venerable history of "typed" foundations:

- Frege, *Die Grundlagen der Arithmetik*, 1884
- Russell and Whitehead, *Principia Mathematica*, 1910-1913
- Church, "A formulation of a simple theory of types," 1940
- Martin-Löf, "A theory of types," 1971
- Coquand and Huet, "The calculus of constructions," 1988

These are straightforwardly interpretable in set theory.

The importance of foundations

My goals here:

- to tell you about dependent type theory
- to argue that it provides a *practical* foundation for mathematics

The key issue

In ordinary mathematics, an expression may denote:

- a natural number: 3, $n^2 + 1$
- an integer: -5, $2j$
- an ordered triple of natural numbers: $(1, 2, 3)$
- a function from natural numbers to reals: $(s_n)_{n \in \mathbb{N}}$
- a set of reals: $[0, 1]$
- a function which takes a measurable function from the reals to the reals and a set of reals and returns a real: $\int_A f \; d\lambda$
- an additive group: $\mathbb{Z}/m\mathbb{Z}$
- a ring: $\mathbb{Z}/m\mathbb{Z}$
- a module over some ring: $\mathbb{Z}/m\mathbb{Z}$ as a $\mathbb{Z}$-module
- an element of a group: $g \in G$
- a function which takes an element of a group and a natural number and returns another element of the group: g^n
- a homomorphism between groups: $f : G \to G$
- a function which takes a sequence of groups and returns a group: $\prod_i G_i$
- a function which takes a sequence of groups indexed by some diagram and homomorphisms between them and returns a group: $\lim_{i \in D} G_i$

The key issue

In set theory, these are all sets.

We have to rely on hypotheses and theorems to establish that these objects fall into the indicated classes.

For many purposes, it is useful to have a discipline which assigns to each syntactic object a *type*.

This is what type theory is designed to do.

Simple type theory

In simple type theory, we start with some basic types, and build compound types.

```
check ℕ    -- Type₁
check bool
check ℕ → bool
check ℕ × bool
check ℕ → ℕ
check ℕ × ℕ → ℕ
check ℕ → ℕ → ℕ
check ℕ → (ℕ → ℕ)
check ℕ → ℕ → bool
check (ℕ → ℕ) → ℕ
```

Simple type theory

We then have terms of the various types:

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

check f
check f n
check g m n
check g m
check (m, n)
check pr₁ p
check m + n^2 + 7
check F f
check F (g m)
check f (pr₂ (pr₁ (p, (g m, n))))
```

Simple type theory

Modern variants include variables ranging over types and type constructors.

```
variables A B : Type

check list A
check set A
check A × B
check A × ℕ

variables (l : list A) (a b c : A) (s : set A)

check a :: l
check [a, b] ++ c :: l
check length l
check '{a} ∪ s
```

Dependent type theory

In dependent type theory, type constructors can take terms as arguments:

```
variables (A : Type) (m n : ℕ)

check tuple A n
check matrix ℝ m n
check Zmod n

variables (s : tuple A m) (t : tuple A n)

check s ++ t     -- tuple A (m + n)
```

The trick: types themselves are now terms in the language.

Dependent type theory

For example, type constructors are now type-valued functions:

```
variables A B : Type

constant prod : Type → Type → Type
constant list : Type → Type

check prod A B
check list A
check prod A ℕ
check list (prod A ℕ)
```

Note: foundationally, we want to *define* `prod` and `list`, not declare then.

Dependent type theory

Now it is easy to express dependencies:

```
constant list   : Type → Type
constant tuple  : Type → ℕ → Type
constant matrix : Type → ℕ → ℕ → Type
constant Zmod   : ℕ → Type

variables (A : Type) (m n : ℕ)

check tuple A n
check matrix ℝ m n
check Zmod n
```

Interactive theorem provers

Among some of the ITP's with large mathematical libraries:

- ACL2 is close to quantifier-free many sorted logic
- Mizar and Metamath are based on first-order logic / set theory
- HOL4, HOL light, and Isabelle use simple type theory
- Coq, Agda, and Lean use dependent type theory
- NuPrl, PVS use extensional dependent type theories

The Lean theorem prover

The examples in this talk have been checked with the Lean theorem prover:

http://leanprover.github.io/

There is an online interactive tutorial.

Lean's principal developer is Leonardo de Moura, Microsoft Research.

It is open source, released under the Apache 2.0 license.

Lean's standard mode is based on the *Calculus of Inductive Constructions*.

I will focus on this version of dependent type theory.

One language fits all

In simple type theory, we distinguish between

- types
- terms
- propositions
- proofs

Dependent type theory is flexible enough to encode them all in the same language.

Encoding propositions

```
variables p q r : Prop

check p ∧ q
check p ∧ (p → q ∨ ¬ r)

variable  A : Type
variable  S : A → Prop
variable  R : A → A → Prop

local infix ‘ ≺ ‘:50 := R

check ∀ x, S x
check ∀ f : ℕ → A, ∃ n : ℕ, ¬ f (n + 1) ≺ f n
check ∀ f, ∃ n : ℕ, ¬ f (n + 1) ≺ f n
```

Encoding proofs

Given

```
    P : Prop
```

view

```
    t : P
```

as saying "*t* is a proof of *P*."

```
theorem and_swap : p ∧ q → q ∧ p :=
assume H  : p ∧ q,
have H1 : p, from and.left H,
have H2 : q, from and.right H,
show q ∧ p, from and.intro H2 H1

theorem and_swap' : p ∧ q → q ∧ p :=
λ H, and.intro (and.right H) (and.left H)

check and_swap -- ∀ (p q : Prop), p ∧ q → q ∧ p
```

Encoding proofs

```
theorem sqrt_two_irrational {a b : ℕ} (co : coprime a b) :
  a^2 ≠ 2 * b^2 :=
assume H : a^2 = 2 * b^2,
have even (a^2),
  from even_of_exists (exists.intro _ H),
have even a,
  from even_of_even_pow this,
obtain (c : ℕ) (aeq : a = 2 * c),
  from exists_of_even this,
have 2 * (2 * c^2) = 2 * b^2,
  by rewrite [-H, aeq, *pow_two, mul.assoc, mul.left_comm c],
have 2 * c^2 = b^2,
  from eq_of_mul_eq_mul_left dec_trivial this,
have even (b^2),
  from even_of_exists (exists.intro _ (eq.symm this)),
have even b,
  from even_of_even_pow this,
assert 2 | gcd a b,
  from dvd_gcd (dvd_of_even `even a`) (dvd_of_even `even b`),
have 2 | 1,
  by rewrite [gcd_eq_one_of_coprime co at this]; exact this,
show false,
  from absurd `2 | 1` dec_trivial
```

One language fits all

We have:

- types (`T : Type`)
- terms (`t : T`)
- propositions (`P : Prop`)
- proofs (`p : P`)

Everything is a term, and every term has a type, which is another term.

(Under the hood, type universes: `Type.{i} : Type.{i+1}`.)

Implicit arguments

Type theory is verbose. The expression:

```
[a, b] ++ c :: l
```

is really:

```
@list.append.{l_1} A
  (@list.cons.{l_1} A a
    (@list.cons.{l_1} A b (@list.nil.{l_1} A)))
  (@list.cons.{l_1} A c l)
```

Most of the information can be left implicit.

Systems for dependent type theory infer the full representation from the semi-formal expression, and print the more compact version.

Type-theoretic foundations

In the Calculus of Inductive Constructions, we have:

- universes: `Type.{1} : Type.{2} : Type.{3} : ...`
- the type of propositions: `Prop`
- dependent products: `Π x : A, B`
- inductive types

It is striking that so much mathematics can be reduced to these.

The dependent product

The dependent type `Π x : A, B` denotes the type of functions `f` that take an element `x` in `A`, and returns an element `f x` in `B`.

In general, `B` may depend on `x`.

When it doesn't, we write `A → B`.

When `B : Prop`, we can write `∀ x : A, B` instead of `Π x : A, B`.

Inductive types

```
inductive weekday : Type :=
| sunday : weekday
| monday : weekday
| tuesday : weekday
...
| saturday : weekday

inductive empty : Type

inductive unit : Type :=
star : unit

inductive bool : Type :=
| tt : bool
| ff : bool
```

Inductive types

```
inductive prod (A B : Type) :=
mk : A → B → prod A B

inductive sum (A B : Type) : Type :=
| inl {} : A → sum A B
| inr {} : B → sum A B

inductive sigma {A : Type} (B : A → Type) :=
dpair : Π a : A, B a → sigma B
```

Inductive types

The more interesting ones are recursive:

```
inductive nat : Type :=
| zero : nat
| succ : nat → nat

inductive list (A : Type) : Type :=
| nil {} : list A
| cons : A → list A → list A
```

Inductive propositions

```
inductive false : Prop

inductive true : Prop :=
intro : true

inductive and (a b : Prop) : Prop :=
intro : a → b → and a b

inductive or (a b : Prop) : Prop :=
| intro_left  : a → or a b
| intro_right : b → or a b

inductive Exists {A : Type} (P : A → Prop) : Prop :=
intro : ∀ (a : A), P a → Exists P
```

Inductive types

We can also define records inductively:

```
record color := (red : ℕ) (green : ℕ) (blue : ℕ)
```

And structures:

```
structure Semigroup : Type :=
(carrier : Type)
(mul : carrier → carrier → carrier)
(mul_assoc :
    ∀ a b c, mul (mul a b) c = mul a (mul b c))
```

This is just syntactic sugar for inductive definitions.

Additional axioms and constructions

In Lean's standard library we assume "proof irrelevance" for `Prop`.

We can add the following:

- propositional extensionality
- quotients (and hence function extensionality)
- Hilbert choice (which implies excluded middle)

This gives us ordinary classical reasoning.

The purer fragments have better computational behavior.

The number systems

```
inductive nat :=
| zero : nat
| succ : nat → nat

definition add : nat → nat → nat
| add m zero     := m
| add m (succ n) := succ (add m n)

inductive int : Type :=
| of_nat          : nat → int
| neg_succ_of_nat : nat → int

definition rat := quot prerat.setoid

definition real := quot reg_seq.setoid

record complex : Type := (re : ℝ) (im : ℝ)
```

Algebraic structures

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

Goals:

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

Algebraic structures

```
structure semigroup [class] (A : Type) extends has_mul A :=
(mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))

structure monoid [class] (A : Type)
  extends semigroup A, has_one A :=
(one_mul : ∀ a, mul one a = a) (mul_one : ∀ a, mul a one = a)

definition pow {A : Type} [s : monoid A] (a : A) : ℕ → A
| 0     := 1
| (n+1) := pow n * a

theorem pow_add (a : A) (m : ℕ) : ∀ n, a^(m + n) = a^m * a^n
| 0        := by rewrite [nat.add_zero, pow_zero, mul_one]
| (succ n) := by rewrite [add_succ, *pow_succ, pow_add,
                         mul.assoc]

definition int.linear_ordered_comm_ring [instance] :
  linear_ordered_comm_ring int := ...
```

Levels of formality

Informal mathematics:

> **Theorem.** *Let G be any group, $g_1, g_2 \in G$. Then $(g_1 g_2)^{-1} = g_2^{-1} g_1^{-1}$.*

Lean semiformal input:

```
theorem my_theorem (G : Type) [group G] :
    ∀ g₁ g₂ : G, (g₁ * g₂)⁻¹ = g₂⁻¹ * g₁⁻¹
```

Levels of formality

The internal representation:

```
my_theorem.{l_1} :
  ∀ (G : Type.{l_1}) [_inst_1 : group.{l_1} G] (g₁ g₂ : G),
    @eq.{l_1} G
      (@inv.{l_1} G (@group.to_has_inv.{l_1} G _inst_1)
         (@mul.{l_1} G (@group.to.has_mul.{l_1} G _inst_1) g₁ g₂))
      (@mul.{l_1} G (@group.to.has_mul.{l_1} G _inst_1)
         (@inv.{l_1} G (@group.to_has_inv.{l_1} G _inst_1) g₂)
         (@inv.{l_1} G (@group.to_has_inv.{l_1} G _inst_1) g₁))
```

The pretty-printer ordinarily displays this as:

```
my_theorem :
  ∀ (G : Type) [_inst_1 : group G] (g₁ g₂ : G),
    (g₁ * g₂)⁻¹ = g₂⁻¹ * g₁⁻¹
```

Levels of formality

The semi-formal input and output are not terrible, but we certainly want friendlier user interfaces and interactions.

I am advocating for the internal representation:

- It encodes the *precise* meaning of the expression.
- It encodes the user's *intentions*.

The type discipline is important:

- It allows more natural input (the system can infer a lot from type information).
- It allows for more informative error messages.
- It allows for more robust indexing, matching, unification, and search (and hence, hopefully, automation).
- We can translate "down" to first-order encodings when needed.

Computational properties

Terms in dependent type theory come with a computational interpretation:

- `(λ x, t) s` reduces to `t [s / x]`.
- $\texttt{pr}_1$ `(a, b)` reduces to `a`.
- `nat.rec a f 0` reduces to `a`.

Computational behavior:

- In the pure system, closed terms of type $\mathbb{N}$ reduce to numerals.
- Extensionality and excluded middle are compatible with code extraction.
  - `t : T : Type` are data
  - `p : P : Prop` are assertions
- Hilbert choice allows nonconstructive definitions.

Computational properties

Sometimes a type checker has to perform computational reductions to make sense of terms.

For example, suppose `a : A` and `b : B a`. Then $\langle$`a, b`$\rangle$ `: Σ x : A, B x`.

We expect $\texttt{dpr}_2$ $\langle$`a, b`$\rangle$ `= b`, but

- the left-hand side has type `B` ($\texttt{dpr}_1$ $\langle$`a, b`$\rangle$), while
- the right-hand side has type `B a`.

The type checker has to reduce to recognize these as the same.

These issues come up when dealing with records and structures and projections.

Coercions and casts

Two formal tasks:

- Type inference: what type of object is t?
- Type checking: does t have type T?

The dark side of type theory:

- Every term has to infer a type.
- Type checking should be decidable, both in theory and practice.

This makes type checking rigid.

Coercions and casts

Given

- `s : tuple A n`
- `t : tuple A m`
- `u : tuple A k`

we have

- `(s ++ t) ++ u  : tuple A ((n + m) + k)`
- `s ++ (t ++ u) : tuple A (n + (m + k))`

We can prove

- `p : n + (m + k) = (n + m) + k`, and then
- `(s ++ t) ++ u = subst p (s ++ (t ++ u))`.

In other words, we have to "cast" along `p`. Similar headaches arise with `matrix A m n`.

Coercions and casts

Notes:

- We can always revert to a first-order encoding:
  - `t : tuple A,   n = length t`
  - `M : matrix A,   r = rows M,   c = cols M`

  So we are no worse off than in set theory.
- We can use heterogeneous equality , `a == b`.
- But we still want to instantiate `matrix R n n` as a ring, infer parameters, catch user errors, and so on.
- With automation, which should be able to
  - insert casts automatically,
  - manage them, and
  - hide them from users.
- Then we can store with each term the reason it makes sense.

The ideal

Perhaps one day:

- Every mathematical assertion will be expressed formally.
- Every mathematical proof will be expressed formally (and checked).
- Every piece of mathematical software will be checked against a formal specification.
- The results of numeric and symbolic calculation will have formal proofs.
- Automation will produce formal proofs.

The ideal

Don’t hold your breath.

- The technology is “not ready for prime time.”
- Being completely formal places a high burden on users.
- We don’t want to hamper discovery, exploration.
- In some domains (like predicting the weather?) formality doesn’t add much.

Realistic approximations

In the meanwhile, we can:

- verify particularly difficult and important mathematical proofs
- verify some mathematical code
- develop good libraries of formalized mathematics
- develop proof producing automation
- verify safety critical systems in engineering
- verify hardware, software, network protocols
- verify financial systems
- reference formal specifications in computer algebra systems
- make formal methods available to mathematicians, for exploration
- develop general infrastructure

The ideal

Even when full formalization is unattainable, it is an important ideal.

With a formal semantics, our claims, goals, and accomplishments are clear.

Without it, things are always fuzzy.

At times, we are “not even wrong.”

So let’s aim for the ideal.