Let's make set theory great again!

John Harrison
Amazon Web Services

AITP 2018, Aussois

27th March 2018 (10:45–11:30)

Contents

- Why types? Why not?
- Set theory as a foundation
- Formalizing mathematics in set theory
  - Avoiding fake theorems
  - Numeric subtypes
  - Encoding undefinedness
  - Reflection principles
- Relevance to AITP
- Questions / discussions

Type theory and set theory

The divide between type theory and 'untyped' axiomatic set theory goes back to different reactions to the paradoxes of naive set theory:

Type theory and set theory

The divide between type theory and 'untyped' axiomatic set theory goes back to different reactions to the paradoxes of naive set theory:

- Russell — introduced a system of types
- Zermelo — developed axioms for set construction

Type theory and set theory

The divide between type theory and 'untyped' axiomatic set theory goes back to different reactions to the paradoxes of naive set theory:

- Russell — introduced a system of types
- Zermelo — developed axioms for set construction

This divide is still with us today and pretty much all type theories are (distant) descendants of Russell's system.

Foundations in theorem proving

Many of the most popular interactive theorem provers are based on type theory

- Simple type theory (HOL family, Isabelle/HOL)
- Constructive type theory (Agda, Coq, Nuprl)
- Other typed formalisms (IMPS, PVS)

Foundations in theorem proving

Many of the most popular interactive theorem provers are based on type theory

- Simple type theory (HOL family, Isabelle/HOL)
- Constructive type theory (Agda, Coq, Nuprl)
- Other typed formalisms (IMPS, PVS)

Far fewer substantial systems are based on set theory:

- Metamath
- Isabelle/ZF (but much less popular than Isabelle/HOL)
- Mizar (but that layers a type system on top)

Why types?

The dominance of types has come about for a mix of technical and social reasons:

Why types?

The dominance of types has come about for a mix of technical and social reasons:

- Types make logical inference simpler (or even avoid it): $\forall x : \mathbb{R}.\ P(x)$ instead of $\forall x.\ x \in \mathbb{R} \Rightarrow P(x)$

Why types?

The dominance of types has come about for a mix of technical and social reasons:

- Types make logical inference simpler (or even avoid it): $\forall x : \mathbb{R}.\, P(x)$ instead of $\forall x.\, x \in \mathbb{R} \Rightarrow P(x)$
- Types give a systematic way of assigning implicit properties: if $f : G \to H$ is a homomorphism then you know what $+$ means where in $f(x + y) = f(x) + f(y)$

Why types?

The dominance of types has come about for a mix of technical and social reasons:

- Types make logical inference simpler (or even avoid it): $\forall x : \mathbb{R}.\ P(x)$ instead of $\forall x.\ x \in \mathbb{R} \Rightarrow P(x)$
- Types give a systematic way of assigning implicit properties: if $f : G \to H$ is a homomorphism then you know what $+$ means where in $f(x + y) = f(x) + f(y)$
- Types are part of an overall philosophical approach to foundations, e.g. from Martin-Löf

Why types?

The dominance of types has come about for a mix of technical and social reasons:

- Types make logical inference simpler (or even avoid it): $\forall x : \mathbb{R}.\, P(x)$ instead of $\forall x.\, x \in \mathbb{R} \Rightarrow P(x)$
- Types give a systematic way of assigning implicit properties: if $f : G \to H$ is a homomorphism then you know what $+$ means where in $f(x + y) = f(x) + f(y)$
- Types are part of an overall philosophical approach to foundations, e.g. from Martin-Löf
- Types are natural to computer scientists who develop many theorem proving programs.

Why types?

The dominance of types has come about for a mix of technical and social reasons:

- Types make logical inference simpler (or even avoid it): $\forall x : \mathbb{R}.\, P(x)$ instead of $\forall x.\, x \in \mathbb{R} \Rightarrow P(x)$
- Types give a systematic way of assigning implicit properties: if $f : G \to H$ is a homomorphism then you know what $+$ means where in $f(x + y) = f(x) + f(y)$
- Types are part of an overall philosophical approach to foundations, e.g. from Martin-Löf
- Types are natural to computer scientists who develop many theorem proving programs.
- Types are a rich topic of pure research and therefore more 'interesting'

Why types?

The dominance of types has come about for a mix of technical and social reasons:

- Types make logical inference simpler (or even avoid it): $\forall x : \mathbb{R}.\, P(x)$ instead of $\forall x.\, x \in \mathbb{R} \Rightarrow P(x)$
- Types give a systematic way of assigning implicit properties: if $f : G \to H$ is a homomorphism then you know what $+$ means where in $f(x + y) = f(x) + f(y)$
- Types are part of an overall philosophical approach to foundations, e.g. from Martin-Löf
- Types are natural to computer scientists who develop many theorem proving programs.
- Types are a rich topic of pure research and therefore more 'interesting'

But not all these are good reasons, and some are perverse incentives.

Why not types?

My thesis is that types, despite their merits, have significant disadvantages:

Why not types?

My thesis is that types, despite their merits, have significant disadvantages:

- Types can create dilemmas or inflexibility

Why not types?

My thesis is that types, despite their merits, have significant disadvantages:

- Types can create dilemmas or inflexibility
- Types can clutter proofs

Why not types?

My thesis is that types, despite their merits, have significant disadvantages:

- Types can create dilemmas or inflexibility
- Types can clutter proofs
- Subtypes may not work smoothly

Why not types?

My thesis is that types, despite their merits, have significant disadvantages:

- Types can create dilemmas or inflexibility
- Types can clutter proofs
- Subtypes may not work smoothly
- Type systems are complicated

There are simple type theories like HOL but they are the most inflexible.

Types can create dilemmas or inflexibility

When formalizing anything intuivtively corresponding to a predicate/set, say over some domain D

- We can formalize it as a predicate $P : D \to B$ or subset $S \subseteq D$
- We can introduce a new type corresponding to P

Types can create dilemmas or inflexibility

When formalizing anything intuivtively corresponding to a predicate/set, say over some domain D

- We can formalize it as a predicate $P : D \to B$ or subset $S \subseteq D$
- We can introduce a new type corresponding to P

We have to make a choice, and depending on other features of the type system, that can greatly influence how easy or hard it is to prove something.
For example, if you prove something generic about groups over a type, you may not be able to instantiate it later to a group over a subset of a type.

Subtypes may not work smoothly

There are type systems with subtypes, but many type systems do not permit it. One special but annoyingly uniquitous case is that you need to distinguish various different number systems

- $\mathbb{N}$, $\mathbb{N}^+ = \mathbb{N} - \{0\}$
- $\mathbb{Z}$
- $\mathbb{Q}$
- $\mathbb{R}$
- $\mathbb{R}^+ = \{x \mid x \in \mathbb{R} \wedge x \geq 0\}$, $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$
- $\mathbb{C}$

You may need multiple versions of theorems, explicit or implicit type casts, lots of complications even if the system partly hides it from the average user.

Types can clutter proofs

Consider a very elementary construction in algebra where we start from an arbitrary field F and construct an extension F' with a root of the irreducible polynomial p:

- Take the ring of polynomials in one variable $F[x]$ (set of finite partial functions $\mathbb{N} \to F$)
- Take the quotient $F[x]/(p(x))$ by the ideal generated by p (elements are equivalence classes, i.e. *sets* of polynomials)

Types can clutter proofs

Consider a very elementary construction in algebra where we start from an arbitrary field F and construct an extension F' with a root of the irreducible polynomial p:

- Take the ring of polynomials in one variable $F[x]$ (set of finite partial functions $\mathbb{N} \to F$)
- Take the quotient $F[x]/(p(x))$ by the ideal generated by p (elements are equivalence classes, i.e. *sets* of polynomials)

Thinking of F as a base type, we have jumped up a couple of levels in the type hierarcy just to adjoin one root.
If we want to construct the algebraic closure of a field we have to do this transfinitely . . .

Type systems are complicated

This inference rule is from Coq (or more precisely Matita)

$$(\mathcal{K}\text{−match})\quad \frac{\begin{array}{l}(\Sigma',\Phi',\mathcal{I}) \in \mathsf{Env} \quad \Sigma' = \emptyset \quad \Phi' = \emptyset \quad \mathsf{Env},\Sigma,\Phi,\Gamma \vdash t : T \\ \mathsf{Env},\Sigma,\Phi,\Gamma \vdash T \triangleright_{\mathrm{whd}} I_l^p\ \overrightarrow{u_l}\ \overrightarrow{u_r} \\ A_p[\overrightarrow{x_l/u_l}] = \Pi\overrightarrow{y_r : Y_r}.s \quad K_p^j[\overrightarrow{x_l/u_l}] = \Pi\overrightarrow{x_{n_j}^j : Q_{n_j}^j}.I_l^p\ \overrightarrow{x_l}\ \overrightarrow{v_r} \qquad j = 1 \ldots m_p \\ \mathsf{Env},\Sigma,\Phi,\Gamma \vdash U : V \quad \mathsf{Env},\Sigma,\Phi,\Gamma \vdash V \triangleright_{\mathrm{whd}} \Pi\overrightarrow{z_r : Y_r}.\Pi z_{r+1} : I_l^p\ \overrightarrow{u_l}\ \overrightarrow{z_r}.s' \\ (s,s') \in \mathsf{elim}(\mathsf{PTS}) \\ \mathsf{Env},\Sigma,\Phi,\Gamma \vdash \lambda\overrightarrow{x_{n_j}^j : P_{n_j}^j}.t_j : T_j \qquad j = 1,\ldots,m_p \\ \mathsf{Env},\Sigma,\Phi,\Gamma \vdash T_j \downarrow \Pi\overrightarrow{x_{n_j}^j : Q_{n_j}^j}.U\ \overrightarrow{v_r}\ (k_j^p\ \overrightarrow{u_l}\ \overrightarrow{x_{n_j}^j}) \qquad j = 1,\ldots,m_p\end{array}}{\begin{array}{l}\mathsf{Env},\Sigma,\Phi,\Gamma \vdash \quad \mathbf{match}\ t\ \mathbf{in}\ I_l^p\ \mathbf{return}\ U \\ \qquad [k_1^p\ (\overrightarrow{x_{n_1}^1 : P_{n_1}^1}) \Rightarrow t_1 \mid \ldots \mid k_{m_p}^p\ (\overrightarrow{x_{n_{m_p}}^{m_p} : P_{n_{m_p}}^{m_p}}) \Rightarrow t_{m_p}\ ] : U\ \overrightarrow{u_r}\ t\end{array}}$$

Set theory as a foundation

We propose in some sense the ‘obvious’ foundation in set theory, and the only innovations are a few conventions we think make thing smoother or more natural.

Set theory as a foundation

We propose in some sense the 'obvious' foundation in set theory, and the only innovations are a few conventions we think make thing smoother or more natural.

- Work in a fairly standard (ZFC...?) universe of sets and construct number systems and mathematical objects in one of the 'usual' ways, probably in fairly standard first-order logic.

Set theory as a foundation

We propose in some sense the 'obvious' foundation in set theory, and the only innovations are a few conventions we think make thing smoother or more natural.

- Work in a fairly standard (ZFC...?) universe of sets and construct number systems and mathematical objects in one of the 'usual' ways, probably in fairly standard first-order logic.
- Things you would express as type constraints in typed systems are usually expressed as set membership: $x : \mathbb{R}$ becomes $x \in \mathbb{R}$ etc.

Set theory as a foundation

We propose in some sense the 'obvious' foundation in set theory, and the only innovations are a few conventions we think make thing smoother or more natural.

- Work in a fairly standard (ZFC...?) universe of sets and construct number systems and mathematical objects in one of the 'usual' ways, probably in fairly standard first-order logic.
- Things you would express as type constraints in typed systems are usually expressed as set membership: $x : \mathbb{R}$ becomes $x \in \mathbb{R}$ etc.
- Constraints that quantify over 'large' collections like $w : \mathrm{ordinal}$ become applications of predicates $\mathrm{ordinal}(w)$, though we could support syntactic sugar like $x \in On$.

Set theory as a machine code

The philosophy is to use set theory act as a simple, well-understood foundation but leave the theorem proving to layers of code, which the foundations don't help but also don't hinder.

- Can do some kind of 'type checking' for catching errors, encouraging a disciplined style, and do some inference more efficiently.
- Wiedijk's paper "Mizar's soft type theory" shows how in principle Mizar's type system can be understood this way, even though in practice it's coded separately.

Set theory as a machine code

The philosophy is to use set theory act as a simple, well-understood foundation but leave the theorem proving to layers of code, which the foundations don't help but also don't hinder.

- Can do some kind of 'type checking' for catching errors, encouraging a disciplined style, and do some inference more efficiently.
- Wiedijk's paper "Mizar's soft type theory" shows how in principle Mizar's type system can be understood this way, even though in practice it's coded separately.
- Other convenient 'magic' like using symmetries, transferring results via isomorphisms, homotopy equivalence or elementary equivalence (Urban's Ultraviolence Axiom) is done by theorem proving, not the foundations.

This is a computer science view, analogous to starting with machine code as the foundation and building higher-level layers on top.

Avoiding fake theorems

- Set theory is sometimes criticized because you get too many identifications or spurious theorems from the constructions: 'zero is a subset of a line'
- We propose to use definitional extension principles that merely require a consistency proof (analogous to type definition rules in HOL) but don't necessarily tie
- You still get some 'fake theorems' if you consider everything as a set: $\emptyset \subseteq$ anything.
- Even those can be avoided by starting with a set theory allowing urelements (not everything has to be a set).

Numeric subtypes

The idea that the usual number systems are all overlaid with the obvious subset relations is ubiquitous in the mathematical literature.

- We don't necessarily propose to help out with other analogous conventions: 0 can also be the trivial group, 2 can be $1_R +_R 1_R$ in a ring, ...
- But the number system inclusions are so ingrained in informal mathematics, and the profusion of different number systems is so inconvenient, that it's worth the effort to make this literally true.
- Each time a new number system is constructed we show that we could make it a superset ($\mathbb{Q} \subseteq \mathbb{R}$ etc.) even if it doesn't arise naturally that way.
- If all else fails, just take the union of the smaller structure and the new elements minus the isomorphic image of the smaller one.

Encoding undefinedness (1)

There are a number of common conventions around 'undefinedness' in mathematics, which arguably don't fit well with typcial formal treatments.
Often equations are taken implicitly to include definedness: $s = t$ means 'either both s and t are both undefined, or they are both defined and equal'.

Encoding undefinedness (1)

There are a number of common conventions around 'undefinedness' in mathematics, which arguably don't fit well with typcial formal treatments.

Often equations are taken implicitly to include definedness: $s = t$ means 'either both s and t are both undefined, or they are both defined and equal'.

So for instance this equation includes the assertion that the sum converges

$$\sum_{n=1}^{\infty} 1/n^2 = \pi^2/6$$

Encoding undefinedness (1)

There are a number of common conventions around 'undefinedness' in mathematics, which arguably don't fit well with typcial formal treatments.

Often equations are taken implicitly to include definedness: $s = t$ means 'either both s and t are both undefined, or they are both defined and equal'.

So for instance this equation includes the assertion that the sum converges

$$\sum_{n=1}^{\infty} 1/n^2 = \pi^2/6$$

And this one holds over $\mathbb{R}$ regardless of whether x and y are zero

$$(xy)^{-1} = x^{-1}y^{-1}$$

Encoding undefinedness (2)

There are a number of formal approaches, which require a lot of complexity or a lot of radical logical changes:

- Every type is lifted and includes an 'undefined' element $\perp$ (LCF)
- The logic explicitly supports partial terms (IMPS) or even three-valued predicates (VDM)

Encoding undefinedness (2)

There are a number of formal approaches, which require a lot of complexity or a lot of radical logical changes:

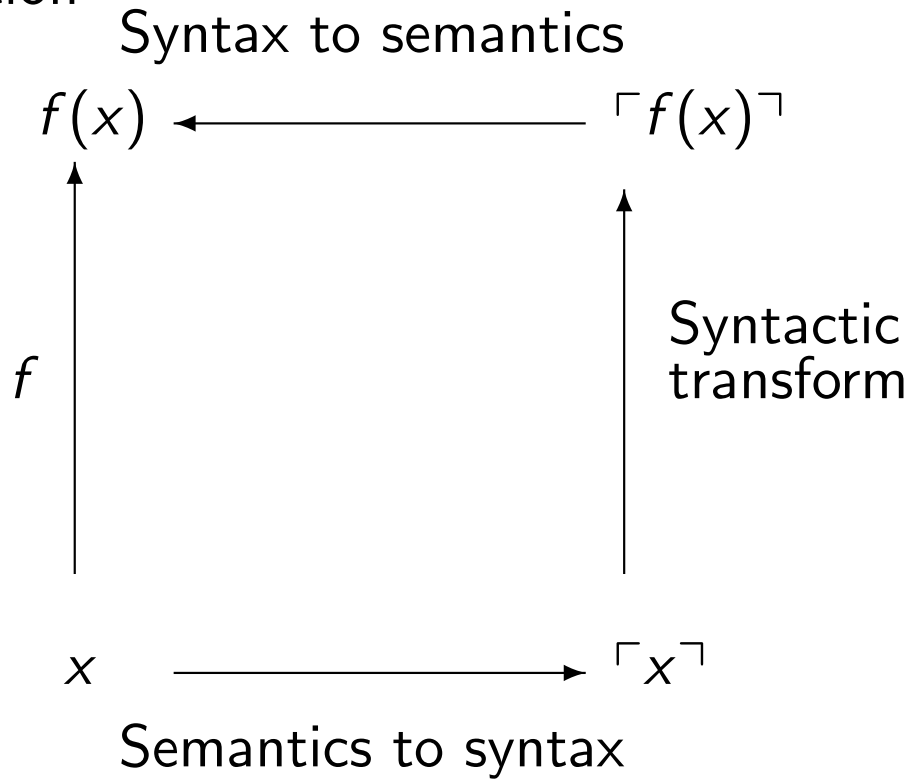
- Every type is lifted and includes an 'undefined' element $\perp$ (LCF)
- The logic explicitly supports partial terms (IMPS) or even three-valued predicates (VDM)

In set theory we can get much of this with one trivial convention:

- Every function $f : A \to B$ explicitly contains a domain A and codomain B.
- Function application is defined to map $f(x) = B$ (the set B itself) if $x \notin A$. So $f(x) \in B \Leftrightarrow x \in A$ (since $B \notin B$ in ZF).
- This amounts to using the codomain itself as a kind of bottom element, rather like LCF
- No theorem proving obligations we didn't have before, and a simple encoding of 'undefined' terms

Reflection (1)

A common pattern in theorem proving is the following, often called (small-scale) reflection



The idea is to do most of the work in the 'syntactic' representation, because you can prove a more generic theorem in this context or (in Coq) because proof/evaluation is faster there.

Reflection (2)

What about reflection in set theory?

- The basic pattern of small-scale reflection is equally applicable in set theory; in fact the absence of types may make evaluation functions easier

Reflection (2)

What about reflection in set theory?

- The basic pattern of small-scale reflection is equally applicable in set theory; in fact the absence of types may make evaluation functions easier
- Unlike constructive type theories, there isn't any built-in notion of efficient evaluation, definitional equality etc., but one could consider defining one

Reflection (2)

What about reflection in set theory?

- The basic pattern of small-scale reflection is equally applicable in set theory; in fact the absence of types may make evaluation functions easier
- Unlike constructive type theories, there isn't any built-in notion of efficient evaluation, definitional equality etc., but one could consider defining one

ZFC offers a more interesting large-scale principle in the 'reflection theorem': if ϕ is any formula of first-order ZFC, then there exists a *set* V in which ϕ holds with all quantifiers relativized to V.

Reflection (2)

What about reflection in set theory?

- The basic pattern of small-scale reflection is equally applicable in set theory; in fact the absence of types may make evaluation functions easier
- Unlike constructive type theories, there isn't any built-in notion of efficient evaluation, definitional equality etc., but one could consider defining one

ZFC offers a more interesting large-scale principle in the 'reflection theorem': if ϕ is any formula of first-order ZFC, then there exists a *set* V in which ϕ holds with all quantifiers relativized to V.

- May allow one to perform dynamic or large-scale reflection.
- Apossible approach to using higher-order notions, category theory etc. without the complication of universes.

Relevance to AITP

Maybe thinking about foundations is not the first priority for people interested in applying AI methods, but I would argue that it may give a closer correspondence with informal texts, which might help in projects to exploit that correspondence.

Relevance to AITP

Maybe thinking about foundations is not the first priority for people interested in applying AI methods, but I would argue that it may give a closer correspondence with informal texts, which might help in projects to exploit that correspondence.

> *The original aim of the writer was to take mathematical textbooks such as Landau on the number system, Hardy-Wright on number theory, Hardy on the calculus, Veblen-Young on projective geometry, the volumes by Bourbaki, as outlines and make the machine formalize all the proofs (fill in the gaps).*

Wang "Toward Mechanical Mathematics", 1960.

Questions?