Nominal foundations of mathematics

Murdoch J. Gabbay

30 March 2016

Introduction

Thanks to Jessica for organising this talk. Thank you all for coming.

Why foundations?

Foundations provide the datatypes, abstract machines, and languages without which maths and computing would be impossible:

- Turing machines; λ-calculus.
- Natural numbers; structured datatypes; relations.
- Real numbers and analysis.
- Logic, topology, set theory, type theory . . .
- . . . the list goes on.

Three pillars of mathematics

Most of us live in these:

- First-order logic.
- λ-calculus.
- Set theory.

When we need to form an abstraction, or express a definition, we reach to the systems above.

Even if we don't necessarily realise it! I taught Python last year: the influence of these three pillars on Python's design is quite direct.

Foundations matter: they bring clarity, reliability, uniformity, and speed.

Let's study these foundations from a nominal perspective.

On nominal techniques

Nominal techniques go back to my PhD thesis. We differ from 'ordinary' mathematics in postulating a datatype of names or atoms.

Names are unordered, atomic (have no internal structure), infinite, and permutable. Examples of names include:

- Pointers.
- Variable symbols a and b.
- Channel names.
- Variables (the object of mathematical study in this talk).

Ordinary maths assumes a datatype of numbers. Nominal techniques assume numbers and names.

Nominal techniques = ordinary sets/types + datatype of names.

Adding names

As it turns out, adding names as a datatype is fruitful.

Applications are numerous: in formal methods and mechanised theorem-proving, concurrency, rewriting, automata theory, and much more.

I want today to pull out one thread: using names to model the variables in first-order logic, the λ-calculus, and set theory.

Syntax vs semantics

Variable symbol does not equal variable.

A variable symbol appears in a string, like the symbol 'a' in the strings $\lambda a.a$ and $\forall a.a{=}a$. Similarly the equality symbol and the lambda symbol appear.

$=$ and λ have intended behaviour, for instance: we might expect a property like congruence

$$u = u' \Rightarrow s[a{:=}u] = s[a{:=}u']$$

or a property like β-reduction

$$(\lambda a.a)b \rightarrow b.$$

Variables a and b are also symbols. They too have intended behaviour.

Studying variables

Question: can we use nominal techniques to understand the intended behaviour of the variable symbols that feature in first-order logic, λ-calculus, and set theory?

These have the following intended behaviour in common: they can be substituted. Usually written $[a{:=}u]$ or $[u/a]$ or $[a/u]$ or $[a{\mapsto}u]$.

(This is common, but not universal. Not all variable symbols behave like this, for instance: pointers get dereferenced, not substituted.)

Studying variables

There are also differences:

- In first-order logic, variables can be universally quantified: $\forall a.\phi$.
- In λ-calculus they get β-reduced and possibly η-expanded: $(\lambda a.s)t \to s[a{:=}t]$ and $s \to \lambda a.(sa)$.
- Variables in set theory are a special case of variables in first-order logic, with the added property of being able to form set comprehension $\{a \mid \phi\}$ (the set of a such that ϕ).

Still, these languages have substitutable variables in common, so to understand variables, we must understand substitution.

It will turn out that other behaviour, as listed above, naturally 'pops out of' the model of substitution. I will discuss just $\forall$ and $=$ below (leaving λ and $\{a \mid \phi\}$ to other talks).

Substitution vs meaning of substitution

Substitution is simple on syntax, but the meaning of that substitution may be complex.

Substituting p for $2^{42643801} - 1$ in the syntax 'p is prime' is simple. We obtain '$2^{42643801} - 1$ is prime'.

The meaning of this is non-trivial: we must check primality.

Likewise a multiplication like $131 * 429083 * 812052162169$ is easy to write, but the computational content of what it means is (mildly) non-trivial.

In this talk we are studying the meaning of $[a \mapsto u]$; what it is to substitute a for u—not the syntactic operation $[a{:=}u]$ (syntax was a topic of my PhD).

It turns out that substitution can be axiomatised, just like groups, rings, and fields can be axiomatised:

Nominal algebra axioms of substitution

$$\begin{aligned}
a\#Z \Rightarrow \quad & Z[a\mapsto X] = Z \\
& Z[a\mapsto a] = Z \\
a\#Y \Rightarrow \quad Z[a\mapsto X][b\mapsto Y] &= Z[b\mapsto Y][a\mapsto X[b\mapsto Y]] \\
b\#Z \Rightarrow \quad & Z[a\mapsto X] = ((b\ a)\cdot Z)[b\mapsto X]
\end{aligned}$$

These are axioms in nominal algebra (Gabbay & Mathijssen 2006), which is like ordinary algebra but enriched with names.

The above is an algebraic system; substitution $Z[a\mapsto X]$ is an algebraic operation, just like group multiplication.

Call a set with an operation satisfying the axioms above, a sigma-algebra.

Sigma-algebra

$$\begin{aligned}
a\#Z \Rightarrow \quad & Z[a\mapsto X] = Z \\
& Z[a\mapsto a] = Z \\
a\#Y \Rightarrow \quad & Z[a\mapsto X][b\mapsto Y] = Z[b\mapsto Y][a\mapsto X[b\mapsto Y]] \\
b\#Z \Rightarrow \quad & Z[a\mapsto X] = ((b\ a)\cdot Z)[b\mapsto X]
\end{aligned}$$

$a\#Z$ is a freshness side-condition. It corresponds to saying 'if a is not free in Z' (it requires nominal foundations to be interpreted).

$(b\ a)\cdot Z$ is a permutation. It corresponds to 'swap b and a in Z'. If $b\#Z$ then $(b\ a)\cdot Z$ means 'replace b by a in Z', so this axiom

$$b\#Z \Rightarrow Z[a\mapsto X] = ((b\ a)\cdot Z)[b\mapsto X]$$

is an axiomatic version of this lemma

$$b \notin fv(s) \Rightarrow s[a\mapsto u] = s[a\mapsto b][b\mapsto u].$$

Permutations are better because they form a group.

How to understand variables: from sigma to amgis

We are now ready to understand variables in first-order logic.

Assume a sigma-algebra $x, y, z, u, v \in \mathcal{X}$.
Consider its powerset $p, q \in pow(\mathcal{X})$.

Define an amgis-action on sets by:

$$\begin{aligned} x \in p[u \leftarrow\!\shortmid a] \Leftrightarrow x[a \mapsto u] \in p \quad \text{so} \\ p[u \leftarrow\!\shortmid a] = \{x \mid x[a \mapsto u] \in p\}. \end{aligned}$$

The amgis-action is the functional preimage of the sigma-action.

(Amgis-algebras can be axiomatised, as sigma-algebras were axiomatised above. See e.g. [Gabbay semooc 2016].)

Think of $p[u \leftarrow\!\shortmid a]$ as

"p reprogrammed to believe that a is equal to u."

From amgis back to sigma

Assume an amgis-algebra $p, q \in \mathcal{P}$.
Consider its powerset $X, Y \in pow(\mathcal{P})$.

Define an action by:

$$\begin{aligned} &p \in X[a{\mapsto}u] \Leftrightarrow \text{И}b.p[u{\leftarrow}b] \in (b\ a)\cdot X \quad \text{so} \\ &X[a{\mapsto}u] = \{p \mid \text{И}b.(p[u{\leftarrow}b] \in (b\ a)\cdot X)\}. \end{aligned}$$

The И is the new-quantifier (Gabbay & Pitts 1999). It means 'for a fresh name'.

This generates a sigma-algebra on $pow(\mathcal{P})$!

So taking powersets we alternate: sigma, amgis, sigma.
If $\mathcal{X}$ is a sigma-algebra, then
$pow(\mathcal{X})$ is an amgis-algebra, and
$pow(pow(\mathcal{X}))$ is a sigma-algebra.

Think of $X[a{\mapsto}u]$ as

"X reprogrammed to believe that a is equal to u — then hide/bind a."

Who's asking ... why bother?

If we can go from sigma in $\mathcal{X}$ to amgis is $pow(\mathcal{X})$ back to sigma in $pow(pow(\mathcal{X}))$, then why not just stay in $\mathcal{X}$?

Because $pow(pow(\mathcal{X}))$ has rich sets structure and $\mathcal{X}$ doesn't. In fact:

- It has all the structure of a model of first-order logic.
- It is a sound and complete model of first-order logic.
- We can prove a particularly strong completeness property called Stone duality.

Interlude: duality theory

Consider Boolean algebra: the logic of $\wedge$, $\vee$, and $\neg$, satisfying axioms such as

$$\neg\neg\phi = \phi \quad \text{and} \quad \phi \wedge (\psi \vee \psi') = (\phi \vee \psi) \wedge (\phi \vee \psi').$$

Clearly conjunction $\wedge$ 'looks like' sets intersection $\cap$ and disjunction $\vee$ 'looks like' sets union $\cup$ and negation $\neg$ 'looks like' sets complement $\setminus$.

But is there some model of Boolean algebras that is so wild that it cannot be presented in these terms; so $\wedge$ cannot mean $\cap$ and $\vee$ cannot mean $\cup$ and $\neg$ cannot mean $\setminus$?

Interlude: duality theory

Stone duality for Boolean algebra says: no, there is no such model.

In fact, every Boolean algebra $\mathcal{B}$ can be presented as a subset of $pow(pow(\mathcal{B}))$ where $\wedge$ is $\cap$ and $\vee$ is $\cup$ and $\neg$ is $pow(pow(\mathcal{B})) \setminus$ -.

And every map of Boolean algebras extends to a map of this presentation.

(More technically: Boolean algebras correspond to compact totally disconnected Hausdorff spaces; maps of Boolean algebras correspond to continuous functions.)

Stone duality is a strong sanity guarantee, that logical symbols correspond to sets operations. It gives one very precise, fine-grained account of what a logic 'really means'.

A full Stone duality result is typically hard work ($\geq$50 pages). You don't need to understand the proof, to use the result!

Interlude: duality theory

In brief, my three most recent papers work by giving Stone representations/dualities for first-order logic, the λ-calculus, and set theory (modulo $\approx$90 pages of maths per paper!).

- No Stone duality result for the λ-calculus had previously been known. It was just not possible to engineer the proofs without the fine control of names given by nominal techniques.
- The set theory I consider, Quine's NF, could not be proved consistent using ordinary mathematics. It consistency has been an open problem since the 1930s.
  (NF is fascinating in its own right. This is for another talk.)

So this isn't just a new way of looking at old and well-understood systems.

The nominal models help us to see and prove new things we couldn't see and prove before.

Duality for FOL in $pow(pow(\mathcal{X}))$

Let me indicate the representation of first-order logic.

Consider 'predicates' $X, Y \in pow(pow(\mathcal{X}))$ over a base sigma-algebra $\mathcal{X}$:

Logical conjunction $\wedge$ becomes sets intersection $X \cap Y$.

Negation $\neg$ becomes complement $pow(pow(\mathcal{X})) \setminus X$.

So far, just like Boolean algebras.

First-order logic extends with: variables a, quantification $\forall a$, and equality $=$.

Variables are handled by the sigma- and amgis-actions. Recall:

$$\begin{aligned} p[u \leftarrow\!\shortmid a] &= \{x{\in}\mathcal{X} \mid x[a{\mapsto}u] \in p\} & &\in pow(\mathcal{X}) \\ X[a{\mapsto}u] &= \{p{\in}pow(\mathcal{X}) \mid \reflectbox{N} b.(p[u \leftarrow\!\shortmid b] \in (b\ a){\cdot}X)\} & &\in pow(pow(\mathcal{X})) \end{aligned}$$

Duality for FOL in $pow(pow(\mathcal{X}))$

Universal quantification becomes the following (they are equal):

$$\forall a.X \;=\; \bigcap_u X[a{\mapsto}u] \;=\; \bigvee\{X' \mid X'{\subseteq}X,\; a\#X'\}.$$

- $\bigcap_u X[a{\mapsto}u]$ means "$X(u)$, for all u".
- $\bigvee\{X' \mid X'{\subseteq}X,\; a\#X'\}$ means "the greatest predicate for which a is fresh, and implying X".

Logicians recall from proof-theory ($\forall$**E**) and ($\forall$**I**):

$$\frac{\Gamma \vdash \forall a.\phi}{\Gamma \vdash \phi[a{:=}u]}\,(\forall\mathbf{E}) \qquad \frac{\Gamma \vdash \phi \;(a{\notin}fv(\Gamma))}{\Gamma \vdash \forall a.\phi}\,(\forall\mathbf{I})$$

Model of FOL in $pow(pow(\mathcal{X}))$

Equality becomes this:

$$u{=}v = \{p{\in}pow(\mathcal{X}) \mid \text{И}c.p[u{\leftarrow\!\dashv}c] = p[v{\leftarrow\!\dashv}c]\}.$$

$\text{И}c.p[u{\leftarrow\!\dashv}c] = p[v{\leftarrow\!\dashv}c]\}$ is a congruence property: it is precisely that which is necessary to derive

$$p \in X[a{\mapsto}u] \quad \text{if and only if} \quad p \in X[a{\mapsto}v].$$

Bearing in mind that $p \in X[a{\mapsto}u] \Leftrightarrow \text{И}c.(p[u{\leftarrow\!\dashv}c] \in (c\ a){\cdot}X)$.

λ-calculus and Quine's NF

The models of the λ-calculus and Quine's NF differ in significant details, but they also share a core pattern—a pattern which is accessible using a nominal universe.

With this trio of papers we have covered a swathe of mathematical foundations.

Applications

- Automata: generalised notions of finiteness-with-names.
- Echoes of this in SAT solving. Symmetry breaking. So far unexplored.
- Programming languages with names. (FreshML and FreshOCaml.)
- Theorem-provers. (Nominal Isabelle.)
- Computational aspects of substitution, and intermediate calculi (with substitution, without λ).
- Other systems with name-binding, many of which are less well-understood than the classic systems considered above.
- Meta-variables and meta-programming.
- Logic and programming constructs for handling pointers, locations, and processes; either verification or programming tools.
- Applications to physics (particle creation/destruction)?

Applications

Binders typically ‘create’ and ‘destroy’ a resource which may be linked to within some scope in space or time. For instance:

$$\forall a.(a{=}a)$$

Here resource ‘a’ is created by $\forall$. The occurrences of a in its scope $(a{=}a)$ point to that resource. The resource is destroyed when we leave the scope of the binder.

The meanings vary depending on whether we have $\forall$, λ, $\{a \mid \text{-}\}$, $\int$, ν, threads, processes, memory, or whatever. This is a common situation which appears in diverse areas:

- A pointer- or link-like structure, and a resource which can be dynamically created and destroyed.
- A potentially infinite symmetry up to some noncanonical choice of naming, or basis.

Nominal techniques are a toolset for modelling just this.