Metamathematics based on nominal terms and nominal logic: foundations of a nominal theorem-prover

Murdoch J. Gabbay[⋆]

[⋆] www.gabbay.org.uk

Abstract

We propose nominal terms as the syntax of a theorem-prover, instead of first- or higher-order terms.

Then term-formers can bind, so we can handle binding, but we do not take syntax up to β-equivalence or put functions directly into our syntax.

This gives us expressivity to axiomatise and reason about higher-order data, the π-calculus, locality, substitution, specification languages with binding—whose common attribute is the use of binding—but our theorem-prover remains based on a first-order framework.

1 Nominal terms

Binding naturally appears for instance in axiomatisations involving locality, name-restriction, functions, substitution, and instantiation. Here are examples:

$$\begin{array}{ll}\int_x f(x) = \int_y f(y) & (\forall x.\phi)[x{::=}t] = \forall x.(\phi[x{::=}t]) \quad \text{provided that } x \text{ is not free in } t \\ \lambda x.f(x) = \lambda y.f(y) & P \mid \nu a.Q = \nu a.(P \mid Q) \quad \text{provided that } a \text{ is not free in } P\end{array}$$

In addition, we often want to reason intensionally about inequality of names (e.g. when specifying α-inequality) or about renaming names (e.g. when α-converting).

If we use higher-order terms then binding is identified with functional abstraction via λ-abstraction. Thus $\lambda x.f(x)$ is a term denoting 'input x, then output $f(x)$', and $\int$, $\forall$, and ν are considered as higher-order constants. This is the approach taken e.g. by Isabelle, HOL, and mCRL2 (and also by Coq). But this introduces functions directly as primitive in the meta-language. Furthermore, if names are identified with functional abstraction then we have no intensional access to the name of a variable; only to the functional argument it represents. We invited binding into our syntax; we did not necessarily want to invite its friends, simple type theory and even set theory, to the party too.

I argue that nominal terms give us the power to reason intuitively about binding, and intensionally on names, while remaining first-order. Recent advances have extended nominal terms with first-order quantification. Thus, a theorem-prover based on nominal terms would be a practical basis for mechanised mathematics.

2 An example: axiomatise functional abstraction

Here, for instance, are axioms for the λ-calculus as they might be specified in a nominal terms theorem-prover e.g. based on nominal algebra Gabbay and Mathijssen (2009):

$$\begin{array}{lrll}(\beta\mathbf{var}) & & (\lambda a.a)X & = X \\ (\beta\#) & a\#Z \vdash & (\lambda a.Z)X & = Z \\ (\beta\mathbf{app}) & & (\lambda a.(Z'Z))X & = ((\lambda a.Z')X)((\lambda a.Z)X) \\ (\beta\mathbf{abs}) & b\#X \vdash & (\lambda a.(\lambda b.Z))X & = \lambda b.((\lambda a.Z)X) \\ (\beta\mathbf{id}) & & (\lambda a.Z)a & = Z\end{array}$$

Points to note here are:

- λ is just a term-former, like application, and $\lambda a.X$ decomposes as λ applied to a *nominal atoms-abstraction* $a.x$ which is a mathematically strictly smaller and simpler structure than a function.
- We have just axiomatised functions. From a 'nominal' point of view, this is just an axiomatic theory. Axioms can express arbitrary complexity, if we want it.
- $a\#Z$ and $b\#X$ are freshness side-conditions corresponding to conditions like 'provided x is not free in t'. Nominal terms axioms tend to closely parallel informal axioms; reasoning in a theorem-prover based on nominal terms is likely to be intuitive.

For completeness, here are the axioms above written out again as they might be written informally:

$$\begin{array}{lll}(\lambda x.x)r & = r & \\ (\lambda x.t)r & = t & \text{provided that } x \text{ is not free in } t \\ (\lambda x.(t't))r & = ((\lambda x.t')r)((\lambda x.t)r) & \\ (\lambda x.(\lambda y.t))r & = \lambda y.((\lambda x.t)r) & \text{provided that } y \text{ is not free in } r \\ (\lambda x.t)x & = t & \end{array}$$

This kind of example invites us to think afresh about what it is we are doing when we axiomatise a system with binding, and about the relationship between syntax and semantics. Axiomatisations in nominal terms are different from those using higher-order terms, but in what I would argue is a good way: unfamiliar perhaps, but reasonable.

3 Two more examples: inequality of names and on-the-fly renaming

I mentioned that nominal terms are more expressive than higher-order terms. Here are two simple examples of how:

$$a \neq b \qquad\qquad \phi \Rightarrow \pi{\cdot}\phi$$

The first example observes that the name a is not equal to the name b. Nominal terms permit direct intensional reasoning on names. Higher-order terms cannot not permit this, since variables can always be bound by an outer λ-abstraction (in other words: 'variables vary; names do not').

The second example is a form of α-equivalence, or *equivariance*. Whenever we write 'choose some variables x and y, and prove ϕ', we are using equivariance implicitly to say 'where it does not matter what the names of x and y are'. Higher-order terms cannot permit direct reasoning on renaming variables—once the user has chosen names for variables those names are fixed and they cannot be accessed from within the system.[1]

4 The theorems so far

So far, we have rewriting, equality, and first-order frameworks, called *nominal rewriting*, *nominal algebra*, and *permissive-nominal logic* (Fernández and Gabbay, 2007; Gabbay and Mathijssen, 2009; Dowek and Gabbay, 2010).

We have used them to axiomatise substitution, the λ-calculus, first-order logic, and arithmetic. But that is not all: we have also proved these theories *correct* (Gabbay and Mathijssen, 2008a, 2010, 2008b; Dowek and Gabbay, 2010). That is, the axioms do not just *look* like they axiomatise what they look like they axiomatise, but it has also been proved by arguments on models that they *do* axiomatise what they look like they axiomatise.

Since substitution, the λ-calculus, first-order logic, and arithmetic are a basis for computer science—at least in theory—this can be read as a proof-of-concept for nominal mechanised mathematics. Put another way: a nominal theorem-prover is feasible.

References

Gilles Dowek and Murdoch J. Gabbay. Permissive Nominal Logic. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2010)*, pages 165–176, 2010. doi: http://dx.doi.org/10.1145/1836089.1836111.

Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting (journal version). *Information and Computation*, 205(6): 917–965, June 2007. doi: http://dx.doi.org/10.1016/j.ic.2006.12.002.

Murdoch J. Gabbay and Aad Mathijssen. Capture-Avoiding Substitution as a Nominal Algebra. *Formal Aspects of Computing*, 20(4-5):451–479, June 2008a. doi: http://dx.doi.org/10.1007/11921240_14.

Murdoch J. Gabbay and Aad Mathijssen. One-and-a-halfth-order Logic. *Journal of Logic and Computation*, 18(4):521–562, August 2008b. doi: http://dx.doi.org/10.1093/logcom/exm064.

Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: equational logic with names and binding. *Journal of Logic and Computation*, 19(6):1455–1508, December 2009. doi: http://dx.doi.org/10.1093/logcom/exp033.

Murdoch J. Gabbay and Aad Mathijssen. A nominal axiomatisation of the lambda-calculus. *Journal of Logic and Computation*, 20(2):501–531, April 2010. doi: http://dx.doi.org/10.1093/logcom/exp049.

[1] A solution to this is *raising*. Raising identifies a name with an explicitly numbered functional abstraction. From that point of view, the advantage of nominal techniques is that its naming is 'lazy' and 'global', whereas the naming in raised higher-order terms is 'eager' and 'local'.