Programming with nominal techniques

Murdoch J. Gabbay

13 August 2019

Thanks

Thank you to the organisers Yukiyoshi Kameyama, Ohad Kammar, and Jeremy Yallop for organising this school, for allowing me to participate – and for asking me to give this talk.

Thank you to the organisers Yukiyoshi Kameyama, Ohad Kammar, and Jeremy Yallop for organising this school, for allowing me to participate – and for asking me to give this talk.

Advert: Nominal Techniques Summer School

3rd School on Foundations of Programming and Software Systems

(FoPSS 2019, co-located with HIGHLIGHTS 2019)

Warsaw, 10–15 September 2019

`https://www.mimuw.edu.pl/~fopss19/`

Johannes Borgström	*Nominal Process Calculi and Modal Logics*
Bartek Klin	*Basic Nominal Techniques*
Jamie Gabbay	*Advanced Nominal Techniques*
Andrew Pitts	*Nominal Sets and Functional Programming*
Maribel Fernández	*Nominal Rewriting and Unification*
Mikołaj Bojańczyk	*Computation Theory with Atoms*
Sławomir Lasota	*Computation Theory with Atoms II*
Andrzej Murawski	*Nominal Game Semantics*

Nominal techniques ...

... are an approach to **names** and **name-binding/abstraction** based on semantics in sets with atoms and Choice (ZFAC) [gabbay:equzfn].

Examples of names in TCS include:

- Variable symbols: the 'x' in $\lambda x.x$.
- Pointers l (pointer = name + deref).
- Variables x (variable = name + substitution).
- Channel names a (cf. π-calculus).
- Thread IDs, file handles, & similar.
- Meta-variables (= name + capturing substitution).
- Wires in diagrams (cf. Ghica).
- Orbit-finite sets.
- ... and *much* more.

Nominal techniques ...

... are an approach to names and name-binding/abstraction based on semantics in **sets with atoms and Choice** [gabbay:equzfn] (ZFAC set theory).

It's a foundational theory.

This lends itself to an account of what names *are* — which may be distinct from an account of how names are *programmed on*.

Analogy: arguably numbers *are* the smallest set closed under 0 and succ — but implementations use e.g. binary strings.

That's why it's perfectly possible for nominal techniques to exist at the theory level, and even the user model level — yet for datatypes be implemented e.g. using de Bruijn indexes.

Other name-carrying structures, such as orbit-finite sets, might require some other concrete implementation.

Nominal techniques

I won't go into the nominal sets model of names and binding in this talk. That's for another lecture series. (Go to the nominal summer school advertised above if you like!)

For this talk, I want to focus on implementation.

Previous efforts, such as in my PhD thesis, FreshML, and FreshOCaml, have either tried to extend a language systematically with nominal constructs, or to deeply embed names and permutations inside an existing language.

I want to talk about how to host nominal techniques inside a language, as a guest, in a new way.

Key feature of nominal techniques

The nominal model of names is polymorphic over types.

Thus: implementation of nominal ideas should exist as a package which provides constructs which

- may be slightly weaker than full nominal techniques, and may be unsafe (i.e. raise runtime errors if abused) but
- can be 'just loaded' and
- are polymorphic over all types in the language.

Unclear how to do this. I will first:

- propose solution (high-level, language-independent), then
- suggest implementations.

Your challenge:

- Make it real; implement it.

My proposal follows:

Type-formers

$$\text{Name} : * \rightarrow *$$
$$\text{Nom} : * \rightarrow * \rightarrow *$$

- $a : \text{Name}\,\tau$ says

  a is a name.
  a carries a label of type τ (like 'τ-ref').

- $x : \text{Nom}\,\tau\,\alpha$ says

  x is an element in α.
  Some τ-labelled names may be abstracted in x.

τ-labels as object-level typing info

τ-labels are optional, but convenient. Write $()$ for the unique element of the unit type. Then:

- $a : \mathsf{Name}\,()$ says

  I am a name.
  I have a label, but it's trivial so call me a ***pure name****.*

- $x : \mathsf{Nom}\,()\,(\mathsf{Name}\,())$ says

  I am a pure name. I may be bound.

τ-labels are convenient because many TCS applications involve type environments, as in '$a : \mathtt{N}$' — intuitively, $\mathtt{N} : \tau$ and $(a : \mathtt{N}) \in \mathsf{Name}\,\tau$.

So: τ-labels save us threading an environment of type annotations.

Constructors and destructors

$$\begin{aligned}
\mathsf{Name} &: * \to * \\
\mathsf{Nom} &: * \to * \to *
\end{aligned}$$

$$\begin{aligned}
\mathsf{fresh} &: \forall\tau.\ \tau \to \mathsf{Nom}\,\tau\,(\mathsf{Name}\,\tau) \\
\mathsf{res} &: \forall\tau,\alpha.\ [\mathsf{Name}\,\tau] \times \alpha \to \mathsf{Nom}\,\tau\,\alpha \\
\mathsf{label} &: \forall\tau,\alpha.\ \mathsf{Name}\,\tau \to \tau
\end{aligned}$$

$[\ldots]$ denotes lists.

- fresh(t) generates a fresh t-labelled name n and wraps it in an n-binding.
- res(l, a) (where res = 'restrict') inputs a list of names l and $a : \alpha$, and wraps a in an l-binding.
- label(n) returns whatever value n points at.

Constructors and destructors

$$\begin{aligned} \mathsf{Name} &: * \to * \\ \mathsf{Nom} &: * \to * \to * \end{aligned}$$

$$\begin{aligned} \mathsf{fresh} &: \forall\tau.\ \tau \to \mathsf{Nom}\,\tau\,(\mathsf{Name}\,\tau) \\ \mathsf{res} &: \forall\tau,\alpha.\ [\mathsf{Name}\,\tau] \times \alpha \to \mathsf{Nom}\,\tau\,\alpha \\ \mathsf{label} &: \forall\tau,\alpha.\ \mathsf{Name}\,\tau \to \tau \end{aligned}$$

res is associative, the order of the elements in l doesn't matter, and we have weakening. The list is just a convenience.

res is *dynamic*, or *capturing*. E.g.

$$(\lambda a.\mathsf{res}([n], x))\ n \to \mathsf{res}([n], n).$$

Nom is a monad

Omit top-level type quantifiers henceforth.

$$
\begin{aligned}
&\mathsf{Name} : * \to * \\
&\mathsf{Nom} : \; * \to * \to * \\[1em]
&\mathsf{fresh} : \; \tau \to \mathsf{Nom}\,\tau\,(\mathsf{Name}\,\tau) \\
&\mathsf{res} : \; [\mathsf{Name}\,\tau] \times \alpha \to \mathsf{Nom}\,\tau\,\alpha \\
&\mathsf{label} : \; \mathsf{Name}\,\tau \to \tau \\[1em]
&\mathsf{return} : \alpha \to \mathsf{Nom}\,\tau\,\alpha \\
&>>= : \; \mathsf{Nom}\,\tau\,\alpha \to (\alpha \to \mathsf{Nom}\,\tau\,\beta) \to \mathsf{Nom}\,\tau\,\beta
\end{aligned}
$$

- $\mathsf{N}om\,\tau$ - is a monad, for each τ.
- $\mathsf{return} = \lambda a.\mathsf{res}([], a)$ wraps $a : \alpha$ in an empty binding context.
- $>>=$ is capture-avoiding.
  E.g. monadic combination of $\mathsf{res}([n], n)$ with $\mathsf{res}([n], n)$ is $\mathsf{res}([n_1, n_2], (n_1, n_2))$. See 'Nom equality' slide below.

Unsafe operations on Nom

$$
\begin{aligned}
\mathsf{unNom} &: \mathsf{Nom}\,\tau\,\alpha \to \alpha \\
\mathsf{fuse} &: \mathsf{Name}\,\tau \to \mathsf{Name}\,\tau \to \mathsf{Nom}\,\tau\,()
\end{aligned}
$$

unNom **destroys binding.** Unsafe because names can escape scope:

$$
\begin{aligned}
&\mathsf{unNom}\,\mathsf{res}(n, a) \to a \quad \text{so that} \\
&\mathsf{unNom}\,(\mathsf{fresh}\,()) \to \mathsf{unNom}\,(\mathsf{res}([n], n)) \to n
\end{aligned}
$$

Feature, or bug?

1. Feature! We create a new unique ID.
2. Bug! Bound name has escaped context. Runtime error. (Trigger exception if n evaluated outside scope.)
3. Both! Name n *is* an exception, and res is its handler!

fuse **is effectful.** If $\mathsf{fuse}\,n\,n'$ is called inside $\mathsf{Nom}\,\tau$ monad, it fuses n and n', making them equal; more on this later.

Nom equality

Equality $==$ on $\mathrm{Nom}\,\tau$ follows the monadic structure and is capture-avoiding. Illustrates safe use of unNom.

Let \$ denote right-associative application. For $x, y : \mathrm{Nom}\,\tau\,\alpha$,

$$x == y \quad = \quad \mathrm{unNom}\ \$\ \begin{array}{l} x >>= \lambda a. \\ y >>= \lambda b. \\ \mathrm{return}(a == b). \end{array}$$

Interesting facts. Recall $\mathrm{fresh}() \rightarrow \mathrm{res}([n], n)$ for suitable n. Then:

$$\begin{array}{lcl} \mathrm{fresh}() == \mathrm{fresh}() & \rightarrow & ???? \\ \mathrm{fresh}() >>= \lambda a.a == a & \rightarrow & ???? \end{array}$$

Nom equality

Equality $==$ on $\mathsf{Nom}\,\tau$ follows the monadic structure and is capture-avoiding. Illustrates safe use of unNom.

Let \$ denote right-associative application. For $x, y : \mathsf{Nom}\,\tau\,\alpha$,

$$x == y \quad = \quad \mathsf{unNom}\ \$\ \begin{array}{l} x >>= \lambda a. \\ y >>= \lambda b. \\ \mathsf{return}(a == b). \end{array}$$

Interesting facts. Recall $\mathsf{fresh}() \rightarrow \mathsf{res}([n], n)$ for suitable n. Then:

$$\begin{array}{lcl} \mathsf{fresh}() == \mathsf{fresh}() & \rightarrow & \mathsf{False} \\ \mathsf{fresh}() >>= \lambda a.a == a & \rightarrow & ???? \end{array}$$

Nom equality

Equality $==$ on $\mathsf{Nom}\,\tau$ follows the monadic structure and is capture-avoiding. Illustrates safe use of unNom.

Let \$ denote right-associative application. For $x, y : \mathsf{Nom}\,\tau\,\alpha$,

$$x == y \quad = \quad \mathsf{unNom}\ \$\ \begin{array}{l} x \mathrel{>\!\!>\!=} \lambda a. \\ y \mathrel{>\!\!>\!=} \lambda b. \\ \mathsf{return}(a == b). \end{array}$$

Interesting facts. Recall $\mathsf{fresh}() \rightarrow \mathsf{res}([n], n)$ for suitable n. Then:

$$\begin{array}{lcl} \mathsf{fresh}() == \mathsf{fresh}() & \rightarrow & \mathsf{False} \\ \mathsf{fresh}() \mathrel{>\!\!>\!=} \lambda a.a == a & \rightarrow & \mathsf{True} \end{array}$$

Abstraction

$$\begin{aligned}
\mathsf{Abs} : \quad & * \to * \to * \\[1em]
\langle \text{-} \rangle \text{-} : \quad & \tau \to \alpha \to \mathsf{Nom}\,\tau\,(\tau \times \alpha) \\
\langle n \rangle a = \quad & \mathsf{res}([n], (n, a)) \\[1em]
@@: \quad & \mathsf{Nom}\,\tau\,\alpha \to (\tau \times \alpha \to \beta) \to \beta \\
\langle n \rangle a \; @@ \; f = \quad & f(a, x)
\end{aligned}$$

- In words, $\langle n \rangle a$ is '(n, a) in an n-binding'. Call this *abstraction*.
- @@ is *concretion*. It unpacks an abstraction and applies f:

$$\langle n \rangle a \; @@ \; f \to f(n, a).$$

n can escape scope, e.g. $\langle n \rangle n \; @@ \; \lambda n, a.a \to n$.

$$@@ = \lambda x, f.\mathsf{unNom}\ \$\ x >\!>= \mathsf{return} \circ f$$

A simple program

$$\langle n\rangle n \;@@\; \lambda n, a.\langle n\rangle(\langle n\rangle a,\; a) \rightarrow \;\; ????$$

Let's mark the bindings:

$$\langle n^1\rangle n^1 \;@@\; \lambda n, a.\langle n^1\rangle(\langle n^2\rangle a^2,\; a^1)$$

So:

$$\langle n\rangle n \;@@\; \lambda n, a.\langle n\rangle(\langle n\rangle a,\; a) \quad \rightarrow \quad \langle n^1\rangle(\langle n^2\rangle n^2,\; n^1)$$

Abs equality

Abs τ - equality is *not* monadic! For x, y : Abs τ:

$$\begin{array}{rcl} x == x' & = & \mathrm{unNom}\ \$\ \ x >>= \lambda n, a. \\ & & \phantom{\mathrm{unNom}\ \$\ \ } x' >>= \lambda n', a'. \\ & & \phantom{\mathrm{unNom}\ \$\ \ } \mathrm{fuse}(n, n') >> \\ & & \phantom{\mathrm{unNom}\ \$\ \ } \mathrm{return}(a == a') \end{array}$$

Above, $>> t$ is $>>= \lambda a.t$ where a is not free in t.

So in particular,

$$\begin{array}{rcl} \langle n \rangle a == \langle n \rangle a & \rightarrow & \mathrm{True} \\ \langle n \rangle a == \langle n' \rangle a' & \rightarrow & \mathrm{True} \end{array}$$

A useful test program

```
test6 = unNom $ do -- Nom monad
    -- make a fresh name
    n <- fresh ()
    -- create two abstractions
    let (x1,x2) = (res [n] n
                  ,res [n] n)
    -- unpack them
    y1 <- x1
    y2 <- x2
    -- check for equality
    return $ y1 == y2
```

Should this compute True or False?

A useful test program

```
test6 = unNom $ do -- Nom monad
    -- make a fresh name
    n <- fresh ()
    -- create two abstractions
    let (x1,x2) = (res [n] n
                  ,res [n] n)
    -- unpack them
    y1 <- x1
    y2 <- x2
    -- check for equality
    return $ y1 == y2
```

This should compute False.

Equality is **capture-avoiding** and restriction captures **dynamically**. Each of the two restrictions res$([n], n)$ 'owns' its own local copy of n.

Abs-by-name vs. abs-by-function

```
-- fresh f returns the value of f at a
   fresh name
atFresh :: t -> (Name t -> a) -> Nom t a
atFresh t f = f <$> fresh t

-- Abstract a name in an element
absByName :: Name t -> a -> Abs t a
absByName n a = Abs $ res [n] (n, a)

-- Apply f to a fresh element of type t
absFresh :: t -> (Name t -> a) -> Abs t a
absFresh t f = Abs .
               atFresh t $ \m -> (m, f m)
```

Characteristic property of nominal abstraction

```
-- Concretion of an abstraction at a name.
     Unsafe if name is not fresh.
conc :: Abs m a -> Name m -> a
conc a' m' = a' @@ \m a -> unsafeUnNom $
                 fuseLeft [(m, m')]
              >> return a

-- near inverse to <*>;
-- absFuncIn . absFuncOut = id but
--     not necessarily other way around
absFuncOut :: Default t =>
               (Abs t a -> Abs t b) ->
                Abs t (a -> b)
absFuncOut f = absFresh def
    (\n a -> conc (f (absByName n a)) n)
```

This suggests our constructs are relatively powerful.

Let's look at some code

```
Nominal_IOref.hs
SystemF.hs
Nominal_resumable_exceptions.hs
```

What's a name?

Mathematically, a name n is a datum that is:

- dynamically bindable,
- testable for equality, and
- can be generated fresh.

But this doesn't directly help write a Nominal package for Haskell, Scala, Perl, OCaml, and so forth.

What's a name?

I propose that a name n is a **resumable exception**.

A name is a widget that just holds a **private ID**. It remains passive until triggered with one of two questions:

- A. What is your public ID?
  E.g. $== :: \mathsf{Name}\,\tau \to \mathsf{Name}\,\tau \to \mathsf{Bool}$ tests for equality of public IDs.
- B. What is your label?
  $\mathsf{label} :: \mathsf{Name}\,\tau \to \tau$ queries this.

On query, a name raises an exception labelled with its private ID, which is caught by the innermost handler holding the name's private ID – if this exists!

I recommend not insisting that a handler always exists, so that names are data, not data-in-a-monadic-context.

The handler calculates and answer and then **returns** flow of control to the query site, and execution resumes.

What's a name?

I propose that a name n is a **UNIX channel** (e.g. stdio).

A name is a widget that just holds the **private ID** of the channel. It remains passive until triggered with one of two questions:

- A. What is your public ID? E.g. $==$ tests for equality of public IDs.
- B. What is your label? label queries this.

In both cases it queries a channel handler on the private ID.

The handler calculates and answer and **returns** an answer.

Like `cat "Hello world" > filename.txt`.