Nominal programming for the working programmer

Murdoch J. Gabbay

30 June 2021

Thanks

Thank you to the SPLS organisers for the invitation to speak.

I'll try to keep this talk short and simple, so if you don't follow please just ask a question.

Names are everywhere

If you just look, names are everywhere. Here's a partial list:

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

Name-handling packages (Haskell-centric)

I know of two Haskell packages for name-management in nominal style:

- One by myself:
  `https://hackage.haskell.org/package/nom`
- One by Peter Selinger:
  `https://hackage.haskell.org/package/nominal`

Nominal packages (Haskell-centric)

Notable non-nominal packages include:

- The *hobbits* package by Eddy Westbrook, Nicolas Frisby, and Paul Brauner
  `https://hackage.haskell.org/package/hobbits`
- The *bound* package by Edward Kmett
  `https://hackage.haskell.org/package/bound`
  (fabulous slogan: *making de Briujn succ less*).
- The *unbound* package by Stephanie Weirich
  `https://hackage.haskell.org/package/unbound`.

Hobbits started in 2011; *bound* in 2012; *unbound* in 2018; *nominal* in 2019; and *nom* in 2020.

There are others (and cf. Conor and others' work in Agda).

History

Nominal techniques were introduced in 2001 in my PhD. A FreshML — slogan *programming with binders made simple* — was there too.

A determined but unsuccessful attempt was made to integrate these ideas into OCaml (`http://www.fresh-ocaml.org/`).

An issue with FreshML may have been that the ideas were ahead of their time (discuss -mjg). So it's interesting to note the recent rash of nominal Haskell packages. Perhaps the world is readier for this now than it was.

Today

There's much to discuss about the nominal package . . . so let's not do that!

In this talk, I'd like to talk about *all the other* languages.

If the time is right to take a fresh look (pun intended) at names in programming — then what would it take to put principled yet practical name-handling on the working programmer's workbench?

It helps to work from a set of canonical programs that we want to implement. I present . . .

. . . the canonical nominal program

I propose there's just one program to look at:

- In a λ-calculus:

$$\lambda x : [\mathbb{A}]\mathbb{A}.\ \text{new}\, a.\ ([a](x@a), [a](x@a))$$

- In imperative style:

```
1  input x     % Secretly, x = [a]x'
2  a = fresh
3  x' = x @ a
4  x1 = [a]x' % if x'=a then x1=[a]a=[b]b
5  x2 = [a]x' % if x'=a then x2=[a]a=[b]b
6  return (x1, x2)
```

(Actually there may be two or three. But this one is a great start. Ping me for more details.)

The canonical nominal program

Note of our canonical program

$$\lambda x : [\mathbb{A}]\mathbb{A}.\ \text{new}\, a.\ ([a](x@a), [a](x@a))$$

that we create *two* abstraction scopes from *one* scope:

- we start with $x = [a]a$ and
- end with two copies $([a]a, [a]a)$.

The abstraction here is *capturing*, or *dynamic*. The a in the left-hand $[a]a$ in $([a]a, [a]a)$ is independent from the a in the right-hand $[a]a$.

Thus we expect

$$([a]a, [a]a) =_\alpha ([a]a, [b]b) =_\alpha ([a'']a'', [a''']a''')$$

where $=_\alpha$ (alpha-equivalence) implies an observational equivalence.

The canonical nominal program

There are plenty of *fresh* commands to generate a fresh unique id, but I am not aware of many polymorphic dynamic abstraction constructs in the style above.

Let's look at that again, slightly differently:

$$[a]a \xrightarrow{\text{fresh } a} a \longrightarrow (a, a) \longrightarrow ([a]a, a) \longrightarrow ([a]a, [a]a).$$

Or again, choosing explicitly fresh atoms:

$$[a]a \xrightarrow{\text{fresh } a'} a' \longrightarrow (a', a') \longrightarrow ([a'']a'', a') \longrightarrow ([a'']a'', [a''']a''').$$

The canonical nominal program

Note we intend x to be polymorphic (any input):

$$\lambda x : [\mathbb{A}]\mathbb{A}.\ \text{new}\, a.\ ([a](x@a), [a](x@a))$$

If you know specifically what type x is — e.g. an element of an inductive datatype — then you can use generics to automate a pertinent inductive definition.

This may be concretely helpful in specific circumstances, but for this talk it misses the point: we want a practical understanding of names for *any* x (including e.g. x a function).

How to model this

I see *two* natural ways to model this:

- *Assume a permutation action.*
- *Assume resumable exceptions.*

We'll consider each option in turn.

Option 1: Permutation action

One method is to assume a permutation action. Perhaps easiest to express via code:

```
1  input x     % Secretly, x = (a0, x0)
2              % Abstraction written as
3              % pair, for compactness
4  a  = fresh % new unique id
5  x' = perm(a, fst x, snd x)
6              % (a a0).x0
7  a1 = fresh
8  x1 = (a1, perm(a1, a, x'))
9              % (a1 a).(a a0).x0 = (a1 a0).x0
10 a2 = fresh
11 x2 = (a2, perm(a2, a, x'))
12             % (a2 a).(a a0).x0 = (a2 a0).x0
13 return (x1, x2)
```

[talk this through; mention how done in Haskell package; access to compiler would be nice -mjg].

Option 2: Resumable exceptions

A *resumable exception* is an exception such that flow of control returns to the *raise* after some action by the *handler*.

Concretely: assume a type populated by resumable exceptions `a`.

Evaluation of **`raise`** `a` raises an exception call to the innermost **`resume`** `a` **`as`** `val` handler, at which point `val` gets inlined in place of `a`, and computation continues from the call site, which is now replaced with `val`.

Then we can write:

Option 2: Resumable exceptions

```
1  input x     % x = (raise a0, x0)
2  a  = fresh exception
3  x' = resume exn_of(fst x) as raise a in snd x
4  a1 = fresh exception
5  x1 = resume a as raise a1 in (raise a, x')
6  a2 = fresh exception
7  x2 = resume a as raise a2 in (raise a, x')
8  return (x1, x2)
```

[talk this through; mention Perl; again access to compiler would be nice -mjg].

[slight simplification: we might want to be able to compare `a` and `b` for equality]

Conclusions

I suggest: let's write packages that engineer one or other of the approaches above, in various languages.

This captures the essence of the name-handling functionality required (plus a few other small bits).

Conclusions

I shall speak for my own Haskell package: it's *subtle*; the types are *fascinating*; the documentation is *comprehensive* . . .

. . . but is this a compliment? Perhaps this just obfuscates the underlying *operational simplicity*.

Let's write simpler, stupider, better, more accessible packages.

I'm glad I developed the package, but now that I scratched that itch, I want to see (for example) the short, brutal, unsafe — and useful — version in Perl!

I feel quite naughty saying it like that, but it's just a matter of *simplifying*.

Conclusions

There are two slogans here:

1. Names-as-permutable-data.
2. Names-as-resumable-exceptions.

Both seem plausible (the nominal Haskell packages are based on the first).

Question: Are you a hacker? Do you fancy spending a weekend coding up one of the above in your favourite language? If so, then I'd love to talk: please send me an e-mail.

The end! Thank you for listening.