Behind the name: the many faces of atomic terms

Marino Miculan

Department of Mathematics and Computer Science
University of Udine, Italy

Theory Days, Koke, 3-5 February 2006

Names and variables are everywhere. . .

```
int fib(int n)  {
  if (n <= 1)
    return(1);
  else {
    int n1, n2;
    n1 = fib(n-1);  n2 = fib(n-2);
    return(n1+n2);
  }
}

int main(int ac, char *av[])  {
  int n;
  n = atoi(av[1]);
  printf("Fibonacci(%d) is %d\n", n, fib(n));
}
```

Many different uses of atomic symbols.

. . . but with several and different uses

Names are important for handling conceptual complexity

- by decomposing a task into named subtasks
- by hiding irrelevant details (of code, data, terms, types. . .)
- by parametrizing other phrases
- . . .

We will call names, variables. . . , *atomic terms* or *atoms*.

Names and variables in programming languages

In programming languages, names and variables are governed by well-known “laws”, or *principles*. First and foremost:

The Abstraction principle

The phrases of any semantically meaningful syntactic class may be named.

Any construct (or better, any meaning) may be named.
Consequence: use *different* names for different meanings.

Names and variables in programming languages

The Qualification Principle

Any semantically meaningful syntactic class may admit local definition.

qualification = **abstraction** restricted to local scope

Consequence: names have a scope!

The Parameterization Principle

The phrases of any semantically meaningful syntactic class may be parameters.

parameterization = "λ-**abstraction** principle"

Consequence: meaning of *formal* parameters can be bound to that of *actual* parameters.

Does this contradict Abstraction Principle?

No, they are just different kinds of names!

Names and variables in logics

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\ \forall R \quad x \text{ not free in } \Gamma \qquad \frac{\Gamma, A[t/x] \vdash B}{\Gamma, \forall x.A \vdash B}\ \forall L$$

Name x = placeholder to be replaced $\Rightarrow$ similar to formal parameters in parameterization principle.
Here, the name does not carry any meaning on its own — and actually, the type x is ranging over may be empty.
But also, consider the role of axioms in proof theory:

$$EM : A \vee \neg A$$

EM is a name for something we assume to exist and to match the right specification $\Rightarrow$ similar to definitions in abstraction principle
Names = syntactic device to denote semantic objects.

π-calculus [Milner et al 1992]: executive summary

Slogan: take names seriously!

The π-calculus is a process calculus (=small language intended to be a model) for communicating systems where mobility is modeled through name passing

In the π-calculus we can:

- Create new channels (which are names)
- Do I/O over channels (synchronous and asynchronous) including passing channels over channels
- Define processes recursively
- Fork new processes

We cannot (but we can simulate):

- pass processes over channels
- define procedures and λ-abstractions

π-calculus: syntax

- Terms are only names $a, b, x, y \ldots$ - subject of communications
- Processes $P, Q, \ldots$ - components of a system

Processes are defined as follows:

0	the process that does nothing
$\bar{a}b.P$	the process that outputs b on channel a (and then does P)
$a(x).P$	the process that inputs x on channel a (and then does $P\{x\}$)
$P\|Q$	the process made of subprocesses P and Q running concurrently
$!P$	the process that behaves like unboundedly many copies of P
$\nu x.P$	the process that creates a new channel x (and then does $P\{x\}$) - useful for private interactions

x is bound in $a(x).P$ and $\nu x.P$.
Processes are taken up-to α-equivalence.

π-calculus: operational semantics

Dynamics of the calculus is given in terms of a *structural congruence* and a *reaction relation*. Some rules:

$$\nu x.0 \equiv 0 \qquad \nu x.\nu y.P \equiv \nu y.\nu x P$$

$$(\nu x.P)|Q \equiv \nu x.(P|Q) \quad x \notin FN(Q)$$

$$\bar{a}b.P|a(x).Q \rightarrow P|Q\{b/x\}$$

$$\frac{P \rightarrow Q}{P|R \rightarrow Q|R}$$

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$$

$$\frac{P \rightarrow Q}{\nu x.P \rightarrow \nu x.Q} \ (x \text{ fresh})$$

α-conversion of bound variables can be used to generate fresh names.

Example: how a process can learn an hidden name

$$
\begin{aligned}
(\nu x.\bar{a}x.P)|a(y).Q &\equiv (\nu x.\bar{a}x.P|a(y).Q)\\
&\to \nu x.P|Q\{x/y\}
\end{aligned}
$$

What if x is free in Q? Just convert it to something not free in Q:

$$
\begin{aligned}
(\nu x.\bar{a}x.P)|a(y).Q &\equiv (\nu z.\bar{a}z.P\{z/x\})|a(y).Q\\
&\equiv (\nu z.\bar{a}z.P\{z/x\}|a(y).Q)\\
&\to \nu z.P\{z/x\}|Q\{z/y\}
\end{aligned}
$$

Semantically equivalent to the previous one.

Notice

Restricted names *are not* like local parameters; instead, they are bound variables ranging over *fresh (i.e., not used) names*.

Notice: Q acquires *knowledge*, as it receives the name previously private to $P \Rightarrow P$ and Q now share a secret.

Derivation: spi-calculus [Abadi and Gordon 1998]

- π-calculus extended with "cryptographic" operations
- the objects of communications are terms, not only names:

$$M, N ::= n \mid 0 \mid succ(M) \mid x \mid \{M\}_N$$

- Names are essential for representing nonces, keys (and channels). E.g.: $\nu x.\bar{y}\{M\}_x$.
- Processes: as before, plus:

$case\ M\ of\ 0 : P\ succ(x) : Q$	integer case
$case\ M\ of\ \{x\}_N\ in\ P$	shared-key decryption

- Semantics: shared key decryption

$$case\ \{M\}_N\ of\ \{x\}_N\ in\ P > P\{M/x\}$$

The point so far

- Atomic symbols are fundamental tools for representing abstract notions of knowledge.
- Their behaviour may change, but in general they can be (locally) created and passed around. (Sometimes also unified or substituted with terms).
- It is important to have general and uniform tools and methodologies for dealing with these aspects.
- Three main fields:
  - Logics for reasoning with and about names and other atomic symbols
  - Semantic model constructions for modelling the knowledge represented by names
  - Programming languages for writing programs about data with (bound) names (consider, e.g., a compiler)

Logics for reasoning with names

Many logics have been introduced as *metalogical specification systems:*

- a formalism (*metalanguage*) equipped with an *encoding methodology*
- a given object system (e.g., λ-calculus, π-calculus, FOL, ...) can be encoded in the metalanguage
- as a result, we get a logic for reasoning with and about the object system
- (often) implemented in proof assistants/theorem provers
    - useful for quick implementations/prototyping

Logics for reasoning with names

Two general approaches:

- Try to extend existing logics without changing syntax and proof systems
  - Allows to reuse existing implementations and techniques
  - Modular and extensible
  - May be not expressive enough
- Develop new, special-purpose logics
  - customizable to specific expressivity issues
  - Various degrees of "exotic" aspects: in terms, formulas, judgments, sequents,...
  - Need to (re-)implement specific proof assistants/theorem provers

Logics for names: A non-exhaustive list

- $FO\lambda^{\mathbb{N}}$ [Miller and McDowell 1997]
- Nominal Logics [Gabbay and Pitts 1999,. . . , Cheney 2005]
- Theory of Contexts [HMS 1999, 2001,. . .]
- Fresh Logic [Gabbay 2003. . .]
- $FO\lambda^{\nabla}$ [Miller and Tiu 2003]

They differ in many aspects, in particular for the intended nature of (bound) symbols.

	$FO\lambda^{\mathbb{N}}$	HOL/ToC	Nominal Log.	Fresh Log.	$FO\lambda^{\nabla}$
basic logic	FOL	HOL	FOL	FOL/HOL	FOL
terms	λ-calc	λ-calc	ν / λ-calc	ν	λ-calc
formulas	standard	standard	И	ν	∇
judgments	standard	standard	standard	standard	$\sigma \triangleright A$
sequents	standard	standard	std / typed	std / typed	typed
model	IPA	tripos	FM-sets	FM-sets	?

HOL + Theory of Contexts

HOL/ToC is a higher order logic over simply typed λ-calculus with constants, extended with axioms.

- Maximum reuse, lowest re-implementation overhead.

HOL/ToC(Σ)

Simple Theory of Types for a given system Σ
+ (Classical) Higher Order Logic
+ Theory of Contexts

The language of terms allow to represent faithfully the object language, taking care of binders as *functions*.

HOL + Theory of Contexts

- Each syntactic sort is represented by a distinct type
- each term constructors is represented by a (typed) constants
- Binders are represented by *higher-order* constructors: they take *functions* as arguments. For instance

$$nu : (Name \rightarrow Proc) \rightarrow Proc$$
$$in : Name \rightarrow (Name \rightarrow Proc) \rightarrow Proc$$

  $\nu x.\bar{x}y$ is represented as $nu(\lambda x : Name.out(x, y))$
  Thus, objects of type $Name \rightarrow Proc$ (i.e., functions) represent terms with *holes*, i.e. *term contexts*.
- Freshness is rendered by non-occurrence predicates.
  Example: the rule for ν is encoded as

$$\frac{\forall x.x \notin P(\cdot) \wedge x \notin Q(\cdot) \supset P(x) \rightarrow Q(x)}{nu(\lambda x.P(x)) \rightarrow nu(\lambda x.Q(x))}$$

Axioms of the Theory of Contexts

However not all functions in $Name \to Proc$ are suitable (no case analysis over names is allowed)

And we need to assume something about $Name$, after all.

Axiomatic approach

Take the needed properties as *axioms*

- **Fresh:** $Fresh: \quad \forall M : A. \exists a : Name. a \notin M$
- **Extensionality of contexts:**

$$\frac{M(x) = N(x)}{M = N} \; x \notin FN(M, N)$$

- β-exp : $\forall M : A. \forall a : Name. \exists C : Name \to A. C(a) = M \wedge a \notin C$
- **Decidability of occurrence:** (Not needed in classical logic).

$$DEC : \quad \forall M' \forall a.\; a \in M \vee a \notin M$$

HOL/ToC: pros

- Simple
- Successfully applied to many nominal calculi: π-calculus, λ-calculus, Ambients, spi-calculus, ...
- Powerful on propositions: e.g., it allows to derive new induction principles on the structure of the syntax "up to α-conversion"
- Flexible: not committed to a single meaning of atomic symbols
- Easily implemented in existing proof assistants (e.g., Coq), without changing anything of the underlying environment

HOL/ToC: cons

Proposition

The Axiom of Unique Choice ("every functional relation can be turned into a function") is inconsistent with the Theory of Contexts.

Consequences:

- Functional language is "poor": not all functional relations can be turned into functions $\Rightarrow$ good for logic programming, not for functional programming
- Cannot be used in logics with AC or $AC!$ (like, Isabelle/HOL)
- Since $AC!$ holds in any topos, giving a model for HOL with these axioms is not easy (e.g. *Set* is not enough!)
  (but the theory *is* consistent: there is a "tripos" model...)

Nominal Logic

NL(Σ)

Simple Theory of Types with special types and constructors
+ First Order Logic with special quantifier
+ Axioms about swapping, freshness...

Binders are represented as *quotient classes*, not as functions.
Special term constructors:

- *swapping of a and b in M*: $(a\ b) \cdot M$,
- *abstraction of a in M*: $a.M$, of type $\langle Name \rangle \tau$

Notice that a is *not* bound in $a.M$ — actually a can be any term.
For instance, for the π-calculus:

- Types: $Proc$, $Name$, $\langle Name \rangle Proc$
- Term constructors: $in : Name \to \langle Name \rangle Proc \to Proc$, $nu : \langle Name \rangle Proc \to Proc \ldots$

$\nu x.\bar{x}y$ is represented as $nu(x.out(x, y))$.

Nominal Logic: formulas

Formulas: first order logic with a special quantifier $\text{И}a{:}\nu.\phi$.
Intuitive meaning: "ϕ holds for all/any a".
Well-formedness of $\text{И}a.\phi$ is subject to a *freshness condition* about the bound variable:

$$\frac{\Sigma \# a{:}\nu \vdash \phi \ \mathit{form}}{\Sigma \vdash \text{И}a{:}\nu.\phi \ \mathit{form}}$$

Thus, the *(typing) contexts* may contain variables (of names) subject to freshness informations:

$$\Sigma ::= \langle\rangle \mid \Sigma, x{:}\tau \mid \Sigma \# a{:}\nu$$

$\Sigma \# a{:}\nu$ means "a is a variable to be instantiated with names different from those used in Σ".
Example: the rule for ν is encoded as

$$\frac{\text{И}a.(P(a) \to Q(a))}{nu(a.P(a)) \to nu(a.Q(a))}$$

Nominal Logic: axioms. . .

(S_1) $(a\ a) \cdot x \approx x$
(S_2) $(a\ b) \cdot (a\ b) \cdot x \approx x$
(S_3) $(a\ b) \cdot a \approx b$
(E_1) $(a\ b) \cdot c \approx c$
(E_2) $(a\ b) \cdot (t\ u) \approx ((a\ b) \cdot t)((a\ b) \cdot u)$
(E_3) $p(\vec{x}) \supset p((a\ b) \cdot \vec{x})$
(E_4) $(a\ b) \cdot \lambda x{:}\tau.t \approx \lambda x{:}\tau.(a\ b) \cdot t[((a\ b) \cdot x)/x]$
(F_1) $a\#x \wedge b\#x \supset (a\ b) \cdot x \approx x$
(F_2) $a\#b \quad (a{:}\nu, b{:}\nu', \nu \neq \nu')$
(F_3) $a\#a \supset \bot$
(F_4) $a\#b \vee a \approx b$
(A_1) $a\#y \wedge x \approx (a\ b) \cdot y \supset \langle a \rangle x \approx \langle b \rangle y$
(A_2) $\langle a \rangle x \approx \langle b \rangle y \supset (a \approx b \wedge x \approx y) \vee (a\#y \wedge x \approx (a\ b) \cdot y)$
(A_3) $\forall y : \langle \nu \rangle \tau \exists a : \nu \exists x : \tau. y \approx \langle a \rangle x$

Nominal Logic:. . . and some special rules

$$\frac{\Sigma\#a{:}\nu : \Gamma \Rightarrow \phi}{\Sigma : \Gamma \Rightarrow \phi}\ \mathit{Fresh}$$

$$\frac{\Sigma\#a{:}\nu : \Gamma \Rightarrow \phi}{\Sigma : \Gamma \Rightarrow \text{И}a.\phi}\ \text{И}\mathcal{I}$$

$$\frac{\Sigma : \Gamma \Rightarrow \text{И}a.\phi \quad \Sigma\#a{:}\nu : \Gamma, \phi \Rightarrow \psi}{\Sigma : \Gamma \Rightarrow \psi}\ \text{И}\mathcal{E}$$

Intersting properties about И:

$$\text{И}x.\neg\phi \equiv \neg\text{И}x.\phi \qquad \forall x.\phi \supset \text{И}x.\phi \supset \exists x.\phi$$

Nominal Logic: pro and cons

Pros:

- First order logic
- Good proof theory (enjoys cut elimination, ...)
- Validity is decidable
- Model based on (non-standard) set theory
- Consistent with AC! (but not AC) $\Rightarrow$ expressive functional language ($\Rightarrow$ basis for languages as FreshML and CαML.)

Cons:

- "Exotic" quantifier and term constructors (may be confusing at first)
- Typing context with freshness informations
- Not easily implemented (must change existing systems to accomodate permutation axioms and new quantifier)

The point about Logics

- We start having quite several logics for reasoning explicitly with names and binders.
- But none of them is fully satisfactory.
- And no general methodology for developing new logics for different notions of names, has clearly emerged yet.

Models of varying knowledge

Names and variables represent knowledge which may change.
Changes on knowledge must be reflected coherently on data: e.g, unification of variables:

$$x, y \vdash (x\ y) \quad \xrightarrow{\{x/y\}} \quad x \vdash (x\ x)$$

Functor categories

Take an index category whose object represent degree of information, and stratify your basic datatypes (e.g. sets, cpo's,. . .) and proposition according to this structure.

Some recurrent index categories (others are possible)

$\mathbb{F}$

finite sets and functions between them. Given a set n, we can

- add more symbols $w : n \to n+1$ (weakening)
- permute symbols $p : n \to n$ (swapping)
- unify symbols $c : n+1 \to n$ (contraction)

These are the laws of standard variables $\Rightarrow$ $\mathbb{F}$ is good for *variables*

$\mathbb{I}$

Finite sets and *injective* functions only. We still can add and swap symbols, but we cannot contract anymore $\Rightarrow$ $\mathbb{I}$ is good for *names* like in π-calculus, or *locations*

$\mathbb{P}$

Finite sets and *bijective* functions. We can only swap symbols $\Rightarrow$ $\mathbb{P}$ is good for *linear variables*.

Example: Presheaves over $\mathbb{F}$

Structure of $Set^{\mathbb{F}}$: there is:

- A presheaf of *variables* $Var \in Set^{\mathbb{F}}$, $Var = \mathbf{y}(1)$. The action on objects is $Var(n) = n$: the set of allocated variables.
- *Products and coproducts*, which are computed pointwise; the terminal object is the constant functor $\mathcal{K}_1 = \mathbf{y}(0)$: $\mathcal{K}_1(n) = 1$. Exponential and finite powerset functors also.
- A *dynamic allocation* functor $\delta : Set^{\mathbb{F}} \to Set^{\mathbb{F}}$: given $A : \mathbb{F} \to Set$, it is $\delta(A)_n = A_{n+1}$.

Proposition

$(\_)^{Var} \cong \delta$, and hence $\_ \times Var \dashv \delta$.

A similar situation holds for $\mathbb{I}$,. . .

Syntax with variable binders as initial algebras

Using these constructors, we can define endofunctors over $Set^{\mathbb{F}}$.
For instance, for the π-calculus:

$$\Sigma_\pi(A) = \overbrace{1}^{0} + \overbrace{A \times A}^{P|Q} + \overbrace{Var \times Var \times A}^{\bar{x}y.P} + \overbrace{Var \times \delta A}^{x(y).P} + \overbrace{\delta A}^{\nu x.P}$$
$$\Sigma_\pi(A)_n = 1 + A_n + A_n \times A_n + n \times n \times A + n \times A_{n+1}$$

This functor has an initial algebra,

$$Proc \cong \Sigma_\pi(Proc)$$

which corresponds exactly to the syntax of π-calculus.

Other kind of atomic symbols

- In $Set^{\mathbb{I}}, Set^{\mathbb{P}}$ we can do pretty the same constructions. In particular, $Set^{\mathbb{I}}$ is used for semantics of names, locations, etc. Operational semantics of π-calculus proceses can be rendered as coalgebras in $Set^{\mathbb{I}}$ of the "behaviour" functor:

$$BP \triangleq \wp_f(\overbrace{N \times P^N}^{\text{input}} + \overbrace{N \times N \times P}^{\text{output}} + \overbrace{N \times \delta P}^{\text{bound output}} + \overbrace{P}^{\tau})$$

$$(BP)_n = \wp_f(n \times (P_n)^n \times P_{n+1} + n \times n \times P_n + n \times P_{n+1} + P_n).$$

- Can be generalized further, with index categories which allow to deal with different kinds of binders/variables *at once*.

The point about models

We have good techniques to build models for varying knowledge.
But, what these models are useful for?

For proving soundness of logical systems

- HOL/ToC has a model using both $Set^{\mathbb{F}}$ (for representing syntax with variables) and $Set^{\mathbb{I}}$ (for meaning of names);
- Nominal Logic has a model in the full subcategory of $Set^{\mathbb{I}}$ of pullback preserving functors (the *Schanuel topos*, or FM-sets)

For justifying and inspire new principles

- case analysis, pattern matching with bound variables (useful for new programming languages like Cαml and FreshML)
- induction and recursion over syntax with binders (by initiality).
- general forms of substitutions (nice cat theory there)
- bisimulation principles (by finality)....

Conclusions

The situation

- Names, variables are strong devices to represent abstract notions of knowledge, used in many contexts: logics, programming languages, mobility calculi, security. . .
- It is important to have strong tools for reasoning and programming with atomic terms.

But we are on the right way (maybe)

- We start having some good logical systems (for some specific notions of atomic terms), but no general methodology has emerged yet.
- Construction of suitable models is quite streamlined (cf. [Power Tanaka 2003-05])
- New (extensions of) programming languages are on the way

$FO\lambda^{\nabla}$ [Miller&Tiu, LICS 2003]

Motivated by proof theoretical arguments, rather than semantics.

$FO\lambda^{\nabla}(\Sigma)$

Simple Theory of Types without special types and constructors
+ First Order Logic with special quantifier
+ Special proof system

Binders are represented as *functions*, as in HOL/ToC.
But names which are intended to be "fresh" are introduced by a special quantifier $\nabla x.\phi$.
Intuitive meaning: "ϕ holds *uniformly* over x"
For instance, for the π-calculus:

- $\nu x.\bar{x}y$ is represented as $nu(\lambda x.out(x, y))$.
- The rule for ν is rendered as

$$\frac{\nabla x.(P \rightarrow Q)}{\nu(\lambda x.P) \rightarrow \nu(\lambda x.Q)}$$

$FO\lambda^{\nabla}$: syntax

- Types: usual simple types: $\tau ::= o \mid \gamma \mid \tau_1 \rightarrow \tau_2$
- Terms: usual simply typed λ-calculus: $\Sigma \vdash t : \tau$
- Object-level datatypes can be represented by adding types and constructors (even higher-order)
- (Basic) Formulas: standard FOL, plus the special quantifier $\nabla_{\gamma} x.A$
- Generic Judgments:

$$\mathcal{A}, \mathcal{B} ::= \overbrace{(x_1 : \tau_1, \ldots, x_n : \tau_n)}^{\sigma} \triangleright B$$

Think of $x_1, \ldots, x_n$ as *locally scoped constants*.
Local signature cannot be weakened nor contracted!

$FO\lambda^{\nabla}$: Proof system (some rules)

Propositional connectives are "stratified" by local signatures:

$$\frac{\Sigma : \Gamma, \sigma \rhd A \Rightarrow \mathcal{C}}{\Sigma : \Gamma, \sigma \rhd A \wedge B \Rightarrow \mathcal{C}} \wedge\mathcal{L}1 \qquad \frac{\Sigma : \Gamma \Rightarrow \sigma \rhd A \quad \Sigma : \Gamma \Rightarrow \sigma \rhd B}{\Sigma : \Gamma \Rightarrow \sigma \rhd A \wedge B} \wedge\mathcal{R}$$

∇ internalizes local signatures into formulas

$$\frac{\Sigma : \Gamma, \sigma \rhd \nabla_{\gamma} x.B \Rightarrow \mathcal{C}}{\Sigma : \Gamma, \sigma, x{:}\gamma \rhd B \Rightarrow \mathcal{C}} \nabla\mathcal{L} \qquad \frac{\Sigma : \Gamma \Rightarrow \sigma, x{:}\gamma \rhd B}{\Sigma : \Gamma \Rightarrow \sigma \rhd \nabla_{\gamma} x.B} \nabla\mathcal{R}$$

Compare with quantifiers rules:

$$\frac{\Sigma, h : |\sigma| \rightarrow \gamma : \Gamma \Rightarrow \sigma \rhd B[(h\ \sigma)/x]}{\Sigma : \Gamma \Rightarrow \sigma \rhd \forall_{\gamma} x.B} \forall\mathcal{R}$$

$$\frac{\Sigma, \sigma \vdash t : \gamma \quad \Sigma : \sigma \rhd B[t/x] \Rightarrow \mathcal{C}}{\Sigma : \sigma \rhd \forall_{\gamma} x.B \Rightarrow \mathcal{C}} \forall\mathcal{L}$$

$FO\lambda^{\nabla}$: pros and cons

Pros:

- First order logic
- Good proof theory (enjoys cut elimination, ...)
- Validity is decidable

Cons:

- Not easily implemented (must modify existing systems to accomodate local signatures)
- "Exotic" quantifier and term constructors (may be confusing at first)
- Meaning of local symbols different than "fresh names". In fact, ∇ is self-dual (like И), but

$$\forall x.\phi \not\supset \nabla x.\phi \not\supset \exists x.\phi$$

- Model: unknown