Nominal techniques and the meta-level

Murdoch J. Gabbay

24/1/2006, Venzia, Italia

Thanks for having me in this wonderful city.

Thanks for all the coffee.

This will be a general talk about what it is I do (because you so kindly keep asking).

It will not be technical: but see my webpage www.gabbay.org.uk

Abstraction

Computer science is always about representation.

Do you disagree?

We do not lack data nowadays, but information is always precious. Mathematics has historically been very successful at representing information (based on data).

Abstraction

A historical example: Astronomical observations, to Kepler's laws, to Newton's laws.

A biological example: The eye receives much information, which is abstracted by the retina to a collection of lines and passed down the optic nerve (then unpacked and re-abstracted as objects; when this goes wrong, see 'the man who mistook his wife for a hat', or forms of autism).

A contemporary example: Data mining converts data into information. Also, how is that data represented? Prolog? First-order logic? Raw numbers?

Representing information

Theoretical computer science wants to represent information.

Many motivations for this (rigour, efficiency, automation).

An antique example: The zero and positional notation. Algebra. Maple is 'just an application' of an explicit representation of cardinality-with-unknowns, as in $2.x = y$.

A semi-historical example: Databases are 'just an application' of fragments of first-order logic.

A contemporary example: XML is 'just' trees.

Representing information

Theoretical computer science wants to represent information.

If you disagree, are you sure you're not a mathematician?

I don't mind if you are. I'm a mathematician myself — though I apply my mathematics to theoretical computer science.

So what is the mathematics of representing information?

λ-calculus and logic

The classic! The only!

They have in common: abstraction.

$\lambda x.t$, $\forall x.P$, or even just $t(x)$ and $P(x)$.

This makes sense; they represent information.

Information abstracts from data and represents, in one way or another, a parametrisation over many individual cases.

The meta-level/object-level split

One of the most significant discoveries of the 20th century was the meta-level. This arose from paradoxes related to naïve formalisations, but also from physics (quantum mechanics).

Russell's celebrated paradox only the most well-known of many.

Illative combinatory logic, stratification, types, ZF, NF,...

All restrict language to allow to represent 'data' and 'information', while avoiding paradoxes.

The λ-calculus has simple types. (Function at higher type)=information vs (element of lower type)=data.

Logic has predicates vs. terms. Predicate=information, term=data.

The meta-level/object-level split: philosophy thereof

It’s perhaps enlightening to consider what happens when the meta-level/object-level split is misapplied.

Just once case: Psychology.

The ‘subject’ of a psychological experiment makes theories about the ‘scientist’ running the experiment.

Child psychology is particularly prone to this, since adults like to think they’re smarter than kids.

A different approach is necessary.

The meta-level/object-level split

The meta-level we choose, or choose to pretend exists, has a profound influence on our formalisms, and thus (via computers) on our tools. Just like other basic cultural assumptions.

Indeed, this split is far greater than solely mathematics.

But the formalism of mathematics (and economic importance of computers) threw a particularly harsh light on some aspects of the issue.

The issue

Consider the term $\lambda x.t$.

x is a variable symbol and t is a meta-level variable, ranging over λ-terms.

Instantiation of t does not avoid capture: if we set t to be zx, we get $\lambda x.zx$.

Consider the predicate $\forall x.P$.

x is a variable symbol and P is a meta-level variable, ranging over predicates.

Instantiation of P does not avoid capture: if we set P to be $x = z$, we get $\forall x.x = z$.

The issue

Can we model this within the object-level, which is pre-equipped with notions of abstraction (λ and $\forall$)?

Consider the term $(\lambda t.\lambda x.t)(zx)$.

This reduces

$$(\lambda t.\lambda x.t)(zx) \rightsquigarrow \lambda x'.zx$$

Oops!

This is not wrong — it just does not accurately reflect the instantiation behaviour of meta-level variable symbols.

The issue

The problem is that the model of abstraction based on λ (and its friends, e.g. $\forall$) assumes that any interaction with the environment is explicitly represented in a fully-specified interface.

This issue turns up also in software engineering, for example.

Compare enterprise systems architecture, where it is important to assume that interactions between blocks can only occur along specified channels (but components are relatively macroscopic. . .

. . . with other kinds of systems architecture, where it may be important to have complex interconnections ‘across interfaces’, since otherwise the explicit representation of every connection as it traverses components (from A to Z via C,D,. . .) inflates state-space and specification.

The issue

This issue turns up in ‘exceptions’ in programming, which violate normal control flow. cf. use of monads in Haskell to explicitly represent this in control flow in a pure functional language.

The issue appears in so-called ‘incomplete proofs’ and ‘existential variables’ (my claim) where variables are introduced to represent complex objects whose instantiation may involve capture of object-level variable symbols present in the context.

The issue appears in parallelism and concurrency, where it may be desirable to substitute a program fragment with bindings (dynamic linking) or reason about scope and movement of names. Existing concrete models of this include graphs, subject to a notion of substitution in which links may be made with the context.

The issue

Still, back to the original motivation. What is the mathematics of instantiating t in

$$\lambda x.t?$$

Note that t itself is not a λ-term! t is a meta-level variable ranging over λ-terms.

Indeed, x is not an object-level variable symbol. x is a meta-level variable symbol ranging over λ-term variables. (In an implementation, there would be an 'identifier space' or 'name space' to which x is linked at runtime.)

The issue

Substitution of ‘strong’ (meta-level; t) variables for ‘weak’ (object-level; x) variables does not avoid capture.

Substitution of variables of the same level does avoid capture. That’s capture-avoiding substitution $t\left[x \mapsto u\right]$.

Nominal terms [Urban Pitts Gabbay 03]

Nominal terms are a framework which faithfully represents the intuition and informal practice of writing $\lambda x.t$ (and co.) including the capturing behaviour of instantiation of t.

They are abstract syntax, with sorts and term-formers.

$$t, u ::= a, b, c, \ldots \mid X, Y, Z, \ldots \mid [a]t \mid \mathsf{f}(t, \ldots, t) \mid \ldots$$

$a, b, c, \ldots$ are atoms. They represent object-level variable symbols. They have a sort of . . . 'object-level variable symbols'. So object-level variable symbols are data.

$X, Y, Z, \ldots$ are variables or unknowns. They represent unknowns and may have any sort (usually elided).

$[a]t$ is an abstraction. Think of it as $\lambda a.t$, but without β-equivalence.

Nominal terms [Urban Pitts Gabbay 03]

Nominal terms arose from the Fraenkel-Mostowski set semantics of abstraction, which for the first time allowed a semantics specification of abstraction independently of a commitment to functional abstraction, or any other concrete model (graphs, syntax, de Bruijn, α-equivalence, etcetera).

But once you have a term-language you can do all kinds of things with it; operational semantics, logic and proof-theory, λ-calculi, and denotational semantics.

Nominal rewrite system for the λ-calculus

Take $\cdot$ (application) a binary term-former arity $(\mathbb{T},\mathbb{T})\mathbb{T}$.

Write $\cdot(t,u)$ as tu and associate to the left, as usual.

Take λ (abstraction) arity $([\mathbb{A}]\mathbb{T})\mathbb{T}$.

Write $\lambda([a]t)$ as $\lambda[a]t$.

Take sub (explicit substitution) arity $([\mathbb{A}]\mathbb{T},\mathbb{T})\mathbb{T}$.

Write $\mathsf{sub}([a]t,u)$ as $t[a{\mapsto}u]$.

Nominal rewrite system for the λ-calculus

Rewrite rules are:

$$(\lambda[a]X)Y \;\rightarrow\; X[a{\mapsto}Y] \qquad \big(\cdot(\lambda[a]X, Y) \;\rightarrow\; \mathsf{sub}([a]X, Y)\big)$$

and...

Explicit substitution

$$
\begin{array}{lcl}
a[a{\mapsto}X] & \rightarrow & X \\
a\#Z \vdash Z[a{\mapsto}X] & \rightarrow & Z \\
\mathsf{f}(X_1,\ldots,X_n)[a{\mapsto}X] & \rightarrow & \mathsf{f}(X_1[a{\mapsto}X],\ldots,X_n[a{\mapsto}X]) \\
b\#X \vdash ([b]Y)[a{\mapsto}X] & \rightarrow & [b](Y[a{\mapsto}X])
\end{array}
$$

For example:

$$(\lambda[a]a)b \;\longrightarrow\; a[a{\mapsto}b] \;\longrightarrow\; b$$

$$(\lambda[a]aab)b \;\longrightarrow\; (aab)[a{\mapsto}b] \;\longrightarrow\; (aa)[a{\mapsto}b](b[a{\mapsto}b]) \;\longrightarrow^{*}\; bbb$$

For example:

$$(\lambda[a]\lambda[b]a)b \;\longrightarrow\; (\lambda[b]a)[a{\mapsto}b] \;\longrightarrow\; \lambda(([b']a)[a{\mapsto}b]) \overset{b'\#b}{\longrightarrow} \lambda[b'](a[a{\mapsto}b]) \;\longrightarrow\; \lambda[b']b$$

$$(\lambda[a]\lambda[b]Z)X \;\longrightarrow\; (\lambda[b]Z)[a{\mapsto}X] \;\longrightarrow\; \lambda(([b'](b'\ b)\cdot Z)[a{\mapsto}X]) \overset{b'\#X,Z}{\longrightarrow} \lambda[b']((b'\ b)\cdot Z[a{\mapsto}X]).$$

If we also know $a\#Z$ we can further reduce

$$\lambda[b']((b'\ b)\cdot Z[a{\mapsto}X]) \;\longrightarrow\; \lambda[b'](b'\ b)\cdot Z.$$

α-equality and freshness

What is $a\#t$?

$$\frac{a\#t_1 \;\cdots\; a\#t_n}{a\#\mathsf{f}(s_1,\ldots,t_n)} \qquad \frac{a\#t}{a\#[b]t} \qquad \frac{}{a\#b} \qquad \frac{}{a\#[a]t} \qquad \frac{\pi^{-1}(a)\#X}{a\#\pi\cdot X}$$

$a\#[a]t$ always holds.

$a\#X$ only holds if you've assumed $\ldots a\#X$.

$b\#a$ always holds.

$a\#a$ never holds.

$a\#\pi\cdot X$ holds if and only if $\pi^{-1}(a)\#X$ holds.

α-equality and freshness

What is $(a\ b) \cdot X$?

Well, note that it is not possible for $[a]X \approx_\alpha [b]X$.

Then (since rewrites and thus equality should be closed under instantiating unknowns) $[a]a \approx_\alpha [b]a$, which is like $\lambda a.a = \lambda b.a$ (but without the functions, i.e. β-equivalence!).

But we still want to rename atoms, to avoid capture, etc.

So we write $[a]X \approx_\alpha [b](b\ a) \cdot X$.

Nominal rewriting is such that rewrites are equivalent up to the least symmetric transitive reflexive congruence $\approx_\alpha$ such that

$$a, b\#t \vdash (a\ b) \cdot t \approx_\alpha t.$$

Example derivation

$$\dfrac{\dfrac{}{a\#\lambda[a]ba} \qquad \dfrac{\dfrac{a \approx_\alpha a \quad b \approx_\alpha b}{ab \approx_\alpha ab}}{\lambda[b]ab \approx_\alpha (b\,a)\cdot(\lambda[a]ba) \equiv \lambda[b]ab}}{\lambda[a]\lambda[b]ab \approx_\alpha \lambda[b]\lambda[a]ba}$$

Looks like $\lambda f.\lambda x.fx = \lambda x.\lambda f.xf$.

Example derivation

$$\dfrac{a\#\lambda[a](b\ a)\cdot X \qquad \dfrac{\dfrac{}{X \approx_\alpha (b\ a)\circ(b\ a)\cdot X}\,(\#X)}{\lambda[b]X \approx_\alpha (b\ a)\cdot(\lambda[a](b\ a)\cdot X) \equiv \lambda[b](b\ a)\circ(b\ a)\cdot X}}{\lambda[a]\lambda[b]X \approx_\alpha \lambda[b]\lambda[a](b\ a)\cdot X}$$

Looks like?

Note permutation treats open terms (terms with unknowns). Parametric treatment of abstraction.

Equality

We can throw out the directionality of rewriting and consider nominal algebra (Nominal Algebraic Specifications).

Substitution (again)

$$
\begin{array}{rlll}
(\#\mapsto) & a\#X \vdash & X[a\mapsto T] & = X \\
(f\mapsto) & \vdash & \mathsf{f}(X_1,\ldots,X_n)[a\mapsto T] & = \mathsf{f}(X_1[a\mapsto T],\ldots,X_n[a\mapsto T]) \\
(abs\mapsto) & b\#T \vdash & ([b]X)[a\mapsto T] & = [b](X[a\mapsto T]) \\
(var\mapsto) & \vdash & \mathsf{var}(a)[a\mapsto T] & = T \\
(ren\mapsto) & b\#X \vdash & X[a\mapsto \mathsf{var}(b)] & = (b\,a)\cdot X
\end{array}
$$

(var has sort $(\mathbb{A})\mathbb{T}$.)

These axioms are ω-complete — if $t\sigma = u\sigma$ for all closing σ then $t = u$.

This is not at all an easy result.

Logic (first-order)

(Props) $P \Rightarrow Q \Rightarrow P = \top \qquad \neg\neg P \Rightarrow P = \top$

$(P \Rightarrow Q) \Rightarrow (Q \Rightarrow R) \Rightarrow (P \Rightarrow R) = \top \qquad \bot \Rightarrow P = \top$

(Quants)

$\forall[a]P \Rightarrow P[a{\mapsto}T] = \top \qquad \forall[a](P \wedge Q) \Leftrightarrow \forall[a]P \wedge \forall[a]Q = \top$

$a\#P \vdash \forall[a](P \Rightarrow Q) \Leftrightarrow P \Rightarrow \forall[a]Q = \top$

(Eq) $T \approx T = \top \qquad T \approx U \Rightarrow P[a{\mapsto}T] \Leftrightarrow P[a{\mapsto}U] = \top$

One-and-a-halfth order logic.

A programming language with a hierarchy of meta-levels

Suppose sets of variables $a_i, b_i, c_i, n_i, \ldots$ for $i \geq 1$.

a_i has level i. Syntax is given by:

$$s, t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i \mapsto t] \mid \ldots$$

- $s[a_i \mapsto t]$ is explicit substitution.
- $\lambda a_i.t$ is abstraction.

Call b_j stronger than a_i when $j > i$.

E.g. b_3 is stronger than a_1.

Example terms and reductions

x, y, z have level 1. X, Y, Z have level 2.

$(\lambda x.x)y \rightsquigarrow x[x \mapsto y] \rightsquigarrow y$	Ordinary reduction
$(\lambda x.X)[X \mapsto x] \rightsquigarrow \lambda x.(X[X \mapsto x]) \rightsquigarrow \lambda x.x$	Context substitution
$x[X \mapsto t] \rightsquigarrow x$	X stronger than x
$x[x' \mapsto t] \rightsquigarrow x$	Ordinary substitution
$x[x \mapsto t] \rightsquigarrow t$	Ordinary substitution
$X[x \mapsto t] \not\rightsquigarrow$	Suspended substitution

Records

Fix constants 1 and 2.

l and m have level 1, X has level 2.

A record:

$$X[l \mapsto 1][m \mapsto 2]$$

A record lookup:

$$\begin{aligned} X[l \mapsto 1][m \mapsto 2][X \mapsto m] &\rightsquigarrow X[l \mapsto 1][X \mapsto m][m \mapsto 2] \\ &\rightsquigarrow X[X \mapsto m][l \mapsto 1][m \mapsto 2] \\ &\rightsquigarrow m[l \mapsto 1][m \mapsto 2] \\ &\rightsquigarrow m[m \mapsto 2] \\ &\rightsquigarrow 2. \end{aligned}$$

In-place update

$$
\begin{aligned}
X[l{\mapsto}1][m{\mapsto}2][X{\mapsto}X[l{\mapsto}2]] &\rightsquigarrow X[l{\mapsto}1][X{\mapsto}X[l{\mapsto}2]][m{\mapsto}2] \\
&\rightsquigarrow X[X{\mapsto}X[l{\mapsto}2]][l{\mapsto}1][m{\mapsto}2] \\
&\rightsquigarrow X[l{\mapsto}2][l{\mapsto}1][m{\mapsto}2] \\
&\rightsquigarrow X[l{\mapsto}2][m{\mapsto}2]
\end{aligned}
$$

Substitution-as-a-term

$(\lambda X.X[l \mapsto \lambda n.n])$ applied to lm

$$(\lambda X.X[l \mapsto \lambda n.n])lm \rightsquigarrow X[l \mapsto \lambda n.n][X \mapsto lm] \rightsquigarrow^* (\lambda n.n)m$$

Records (again, using λ)

Fix constants 1 and 2.

l and m have level 1. X has level 2.

A record:

$$\lambda X.X[l \mapsto 1][m \mapsto 2].$$

Now we use application to retrieve the value stored at m:

$$(\lambda X.X[l \mapsto 1][m \mapsto 2])m \rightsquigarrow X[l \mapsto 1][m \mapsto 2][X \mapsto m]$$

Records (again, using λ)

$$\lambda X.X[l{\mapsto}\mathcal{W}][m{\mapsto}2]$$

Here $\mathcal{W}$ has level 3. It beats X, l, and m.

Apply $[\mathcal{W}{\mapsto}X]$:

$$\Big(\lambda X.X[l{\mapsto}\mathcal{W}][m{\mapsto}2]\Big)[\mathcal{W}{\mapsto}X] \rightsquigarrow^* \lambda X.X[l{\mapsto}X][m{\mapsto}2].$$

Apply to (lm) and obtain $(l2)2$:

$$\Big(\lambda X.X[l{\mapsto}X][m{\mapsto}2]\Big)(lm) \rightsquigarrow^* lm[l{\mapsto}lm][m{\mapsto}2] \rightsquigarrow^* (l2)2$$

Conclusions

There is a whole other notion of instantiation, which captures — to be compared with substitution, which does not (my terminology).

Instantiation is difficult to manage, because e.g. we do not want $[a]X = [b]X$ but we do want $[a]a = [b]b$.

NEW technology helps to investigate these ideas.