Functional programming and unification with meta-variables

Murdoch J. Gabbay

Universidad Politecnica de Madrid, 30/3/2006

Thanks to Jim Lipton and Julio Mariño

The purpose of this talk. . .

. . . is to publicise two separate threads of ideas which (perhaps) we might use as the germ of a mathematical discussion in the future.

In other words, I think this is totally cool stuff and I want to tell you all about it.

This is a research seminar, so reserve I the perogative my ideas in the wrong order to put.

But you Spanish probably think that sentence is perfectly grammatical, don't you?

Nominal Terms

Nominal Terms model abstraction (in the sense of α-equivalence) without functional abstraction (in the sense of β-equivalence).

Mathematically this is quite interesting!

It brings things into balance: β-equivalence has a well-understood semantics (sets and functions between them). Nominal Terms also have a well-understood semantics (Fraenkel-Mostowski sets).

Substitution

Substitution seems to sit in between α-equivalence and β-equivalence.
It uses abstraction

$$a[a \mapsto t] = b[b \mapsto t]$$

in the sense that atoms ‘bound’ by substitution.

But it is not as powerful as β-equivalence: it cannot represent functions.

I will talk about the theory and semantics of substitution, approached in ‘nominal’ style.

Syntax of nominal terms

Fix countably many atoms $a, b, c, \ldots$. Fix some term-formers f. Let π vary over fi nitely-supported bijections on atoms.

π is fi nitely supported when $\pi(a) = a$ for all atoms except for fi nitely many atoms.

Fix countably many unknowns X. These are morally 'unknown terms', but represented in the syntax. In short, X is a variable.

The syntax of nominal terms is given by:

$$t ::= a \quad | \quad \pi X \quad | \quad [a]t \quad | \quad \mathsf{f}(t, \ldots, t).$$

a is an atom. πX is an unknown term with a suspended permutation. $[a]t$ is an abstraction. $\mathsf{f}(t, \ldots, t)$ is a term!

λ-calculus

We can specify a signature of the λ-calculus with just three constants:

$$\lambda \qquad \mathsf{app} \qquad \mathsf{sub}.$$

(Sugar $\mathsf{app}(t, u)$ as tu and $\mathsf{sub}([a]t, u)$ as $t[a \mapsto u]$.)

We use these to write the rule

$$(\lambda[a]t)u \;=\; t[a \mapsto u].$$

Why use abstraction

The standard minimal theory of equality on nominal terms is given by the rule:

$$a\#t,\ b\#t \Rightarrow (a\ b)t = t.$$

Here $a\#t$ means that if a occurs in t then it does so under an abstraction. Think of $a\#t$ as a generalisation of $a \notin fn(t)$.

$fn(t)$ equals 'free names of t'.

$(a\ b)t$ is a and b swapped in t.

Call this theory of equality CORE.

For example:

The canonical example is

$$b\#X \Rightarrow [a]X = [b](b\ a)X.$$

Note that $(b\ a)[a]X = [b](b\ a)X$.

That is, we need permutations to handle renaming in the presence of unknowns X.

CORE is α-equivalence.

β

If we add this equality

$$(\lambda[a]t)u \;=\; t[a{\mapsto}u]$$

we get LAMBDA.

But of course there's a bit missing — the theory of substitution.

Theory of substitution SUB

$$
\begin{array}{lcll}
a\#Z & \Rightarrow & Z[a{\mapsto}X] & = Z \\
 & & \mathsf{f}(Z_1,\ldots,Z_n)[a{\mapsto}X] & = \mathsf{f}(Z_1[a{\mapsto}X],\ldots,Z_n[a{\mapsto}X]) \\
b\#X & \Rightarrow & ([b]Y)[a{\mapsto}X] & = [b](Y[a{\mapsto}X]) \\
b\#X & \Rightarrow & X[a{\mapsto}b] & = (b\,a)X \\
 & & a[a{\mapsto}X] = X &
\end{array}
$$

Some notes

I switched from using meta-variables t, u to unknowns X, Y, Z. The axioms of the last slide should be understood as valid also for instantiating unknowns.

(This is no big deal.)

Note also the axiom $b\#X \Rightarrow X[a{\mapsto}b] = (b\,a)X$. This raises the question

> Q. If you can express swapping using substitution, why bother with all this swapping nonsense?
> A. Because $X[a{\mapsto}b]$ has one more term-former than X whereas $(b\,a)X$ does not. With swapping we can rename atoms to avoid capture, without increasing the size of a term.

Of course we could base an entire theory on substitution rather than renaming — but that would defeat our purpose.

Example equality

Suppose $a\#u$. Then

$$v[a{\mapsto}t][b{\mapsto}u] = v[b{\mapsto}u][a{\mapsto}t[b{\mapsto}u]]$$

is derivable. Remember that sub is just another term-former!

$$\mathsf{sub}([b](\mathsf{sub}([a]v, t)), u) = \mathsf{sub}([a](\mathsf{sub}([b]v, u)), \mathsf{sub}([b]t, u)).$$

Computability of substitution

As these equalities suggest:

$$v[a{\mapsto}t][b{\mapsto}u] = v[b{\mapsto}u][a{\mapsto}t[b{\mapsto}u]] \quad b\#v \Rightarrow v[a{\mapsto}u] = ((b\,a)v)[b{\mapsto}u]$$

it is not immediately obvious that equality up to SUB is decidable. In fact, this is actually quite hard to prove!

But we did, and it is.

Cool questions (set 1)

What about unifi cation up to SUB? What is its relation to higher-order unifi cation. Is it decidable?

What lambda-calculi (and logics) can we build to program on this substitution?

The NEW calculus of contexts: 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 x, we get $\lambda x.x$.

The issue

Claim: This is the essence of the meta-level.

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

Substitution of variables of the same level does avoid capture.

Let’s base a calculus on this idea.

The issue

Suppose x is weak (level 1, say) and X is stronger (level 2, say), then

$$(\lambda X.\lambda x.X)x \rightsquigarrow (\lambda x.X)[X \mapsto x]$$
$$\rightsquigarrow \lambda x.(X[X \mapsto x]) \rightsquigarrow \lambda x.x.$$

This is important because it captures some part of meta-programming.

The syntax

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:

$$s, t ::= a_i \quad | \quad tt \quad | \quad \lambda a_i.t \quad | \quad t[a_i \mapsto t] \quad | \quad И a_i.\, t.$$

- $s[a_i \mapsto t]$ is explicit substitution.
- $\lambda a_i.t$ is abstraction.
- $И a_i.\, t$ a binder.

Equate up to И-binding, nothing else.

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

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

In-place update as a term

$\lambda\mathcal{W}.\mathcal{W}[X \mapsto X[l \mapsto 2]]$ applied to $X[l \mapsto 1][m \mapsto 2]$

. . . and so on ($\mathcal{W}$ has level 3).

Likewise global state (world = a big hole), and Abadi-Cardelli imp-ε object calculus.

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 $(l2)2$:

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

Records (again, using λ)

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

Is that wrong?

Depends what you want.

This kind of thing makes the Abadi-Cardelli ‘self’ variable work. The issue is that λ does not bind — it abstracts.

И

$$\text{И}X.\ \big(\lambda X.X[l{\mapsto}\mathcal{W}][m{\mapsto}2]\big).$$

Then

$$\begin{aligned}
&\big(\text{И}X.\ \lambda X.X[l{\mapsto}\mathcal{W}][m{\mapsto}2]\big)[\mathcal{W}{\mapsto}X] \\
&\qquad \rightsquigarrow^* \ \text{И}X'.\ (\lambda X'.X'[l{\mapsto}\mathcal{W}][m{\mapsto}2][\mathcal{W}{\mapsto}X]) \\
&\qquad\qquad \rightsquigarrow^* \ \text{И}X'.\ \lambda X'.X'[l{\mapsto}X][m{\mapsto}2]
\end{aligned}$$

Apply to lm:

$$\begin{aligned}
&\text{И}X'.\ (\lambda X'.X'[l{\mapsto}X][m{\mapsto}2])\ (lm) \\
&\qquad \rightsquigarrow \text{И}X'.\ ((\lambda X'.X'[l{\mapsto}X][m{\mapsto}2])\ (lm)) \\
&\qquad \rightsquigarrow \text{И}X'.\ X'[l{\mapsto}X][m{\mapsto}2][X'{\mapsto}lm] \rightsquigarrow^* (X[m{\mapsto}2])2
\end{aligned}$$

И behaves like the π-calculus ν; it floats to the top (extrudes scope).

Summary

1. λ abstracts — it stays put and β-reduces.
2. $\left[x \mapsto s\right]$ substitutes — it floats downwards capturing x until it runs out of term or gets stuck on a stronger variable.
3. И binds — it floats upwards avoiding capture.

Reduction rules

$$
\begin{array}{lll}
(\beta) & (\lambda a_i.s)u \rightsquigarrow s[a_i \mapsto u] & \\
(\sigma a) & a_i[a_i \mapsto u] \rightsquigarrow u & \forall c.\ c\#a_i \Rightarrow c\#u \\
(\sigma\#) & s[a_i \mapsto u] \rightsquigarrow s & a_i\#s \\
(\sigma p) & (a_i t_1 \ldots t_n)[b_j \mapsto u] \rightsquigarrow (a_i[b_j \mapsto u]) \ldots (t_n[b_j \mapsto u]) & \\
(\sigma\sigma) & s[a_i \mapsto u][b_j \mapsto v] \rightsquigarrow s[b_j \mapsto v][a_i \mapsto u[b_j \mapsto v]] & j > i \\
(\sigma\lambda) & (\lambda a_i.s)[c_k \mapsto u] \rightsquigarrow \lambda a_i.(s[c_k \mapsto u]) & a_i\#u, c_k\ k \leq i \\
(\sigma\lambda') & (\lambda a_i.s)[b_j \mapsto u] \rightsquigarrow \lambda a_i.(s[b_j \mapsto u]) & j > i \\
(\sigma tr) & s[a_i \mapsto a_i] \rightsquigarrow s & \\
(\mathcal{И}p) & (\mathcal{И}n_j.\ s)t \rightsquigarrow \mathcal{И}n_j.\ (st) & n_j \notin t \\
(\mathcal{И}\lambda) & \lambda a_i.\mathcal{И}n_j.\ s \rightsquigarrow \mathcal{И}n_j.\ \lambda a_i.s & n_j \neq a_i \\
(\mathcal{И}\sigma) & (\mathcal{И}n_j.\ s)[a_i \mapsto u] \rightsquigarrow \mathcal{И}n_j.\ (s[a_i \mapsto u]) & n_j \notin u\ n_j \neq a_i \\
(\mathcal{И}\notin) & \mathcal{И}n_j.\ s \rightsquigarrow s & n_j \notin s
\end{array}
$$

Partial evaluation (if I have time)

Write

$$\texttt{if} = \lambda a, b, c.abc \quad \texttt{true} = \lambda ab.a \quad \texttt{false} = \lambda ab.b$$
$$\texttt{not} = \lambda a.\texttt{if}\ a\ \texttt{false}\ \texttt{true}.$$

in untyped λ-calculus. Then calculate

$$s = \lambda f, a.\texttt{if}\ a\ (f\ a)\ a \qquad \text{specialised to} \qquad s\ \texttt{not}$$

by β-reduction. We obtain $\lambda a.\texttt{if}\ a\ (\texttt{not}\ a)\ a$.

A more intelligent method may recognise that the program will always return `false` (with types etc.).

Partial evaluation (if I have time)

Choose level 1 variables a, b and level 2 variables and B, C and defi ne

$$\texttt{true} = \lambda ab.a \quad \texttt{false} = \lambda ab.b$$
$$\texttt{if} = \lambda a, B, C.\, a(B[a{\mapsto}\texttt{true}])(C[a{\mapsto}\texttt{false}])$$
$$\texttt{not} = \lambda a.\texttt{if}\ a\ \texttt{false}\ \texttt{true}.$$

So if we get to B, $a = \texttt{true}$. Consider

$$s = \lambda f, a.\texttt{if}\ a\ (f\ a)\ a \qquad \text{specialised to} \qquad s\ \texttt{not}.$$

We obtain:

$$\begin{aligned} s\,\texttt{not} &\rightsquigarrow^* \lambda a.a\ ((\texttt{not}B)[a{\mapsto}\texttt{true}][B{\mapsto}a])\ (C[a{\mapsto}\texttt{false}][C{\mapsto}a]) \\ &\rightsquigarrow^* \lambda a.a\ ((\texttt{not}a)[a{\mapsto}\texttt{true}])\ (a[a{\mapsto}\texttt{false}]) \\ &\rightsquigarrow^* \lambda a.(a\ \texttt{false}\ \texttt{false}). \end{aligned}$$

More effi cient!

Meta-properties

- Confluence.
- Preservation of strong normalisation (for untyped lambda-calculus).
- Hindley-Milner type system. Explicit substitution rule is like that for `let`.
- Applicative characterisation of contextual equivalence.

Conclusions

The meta-level lives in the same world as the object calculus. So does the meta-meta-level. And so on.

This is not achieved by a type-hierarchy: it is achieved by literally insisting on variables of different strengths.

Scope separate from abstraction; necessary for proper control of α-equivalence in the presence of the hierarchy.

Hierarchy of strengths of variables in common with work by Sato et al. But we have different control of α-equivalence.

Explicit substitution calculus.

Model of state, unordered datatypes, and objects. Probably more.

Bringing together substitution and the NEW calculus of contexts

Very simple question:

What about a programming language with meta-programming ideas taken from the NEWcc, and unifi cation ideas taken from nominal unifi cation or variants thereof?

In other words: what logic programming languages can we get out of this stuff?