A NEW calculus of contexts

Murdoch J. Gabbay

Paris, INRIA and PPS, 1/2005

Thank you for fitting me in on short notice!
The issue

Context=‘term with a hole’. E.g. $C[-] = \lambda x.[-]$.

[-] may be filled, e.g. $C[t] = \lambda x.t$, in a capturing manner, e.g. $C[x] = \lambda x.x$.

This is not modelled by $\beta$-reduction since it avoids capture; consider $(\lambda y.\lambda x.y)x \rightsquigarrow^* \lambda x'.x$.

Our vision: suppose a hierarchy of levels of variables of increasing strength. Abstraction and application are (more-or-less) as before. However, substitution for a variable avoids capture for variables of the same strength or stronger, and does not avoid capture for weaker variables.

For example, if $x$ is weak (level 1, say) and $X$ is stronger (level 2, say), then $(\lambda X.\lambda x.X)x \rightsquigarrow \lambda x.x$. 
The issue

Problem: $\alpha$-equivalence.

If $\lambda x.X = \lambda y.X$ then $(\lambda X.\lambda x.X)x \rightsquigarrow \lambda x.y$. This would be bad!

Dropping $\alpha$-equivalence entirely is too drastic. Some capture-avoidance, as in $(\lambda y.\lambda x.y)x$, should be legitimate.

Our answer: Separate scope ($\lambda$) and binding ($\Pi$). Introduce freshness context to manage their interaction. Also use explicit substitution, because it is easy to express the reductions.

Result: Not solely a ‘calculus for contexts’, but also a calculus with good meta-properties and unexpected expressivity including things we might not have expected to have anything to do with contexts.
Suppose countably infinite set of disjoint infinite sets of variables $a_i, b_i, c_i, n_i, \ldots$ for $i \geq 1$. Say $a_i$ has level $i$. Syntax is given by:

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

Call $s[a_i \mapsto t]$ an explicit substitution, $\lambda a_i.t$ an abstraction, and $\forall a_i.t$ a binder.

Terms are equated up to binding by $\forall$ and nothing else.

Call a variable $b_j$ stronger than another $a_i$ when $j > i$ (when it has strictly higher level). $b_3$ is stronger than $a_1$. 
Example terms and reductions

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

\[
(\lambda x.x)y \leadsto x[x \mapsto y] \leadsto y
\]
Ordinary reduction

\[
(\lambda x.X)[X \mapsto x] \leadsto \lambda x.(X[X \mapsto x]) \leadsto \lambda x.x
\]
Context substitution

\[
x[X \mapsto t] \leadsto x
\]
$X$ stronger than $x$

\[
x[x' \mapsto t] \leadsto x
\]
Ordinary substitution

\[
x[x \mapsto t] \leadsto t
\]
Ordinary substitution

\[
X[x \mapsto t] \not\leadsto
\]
Suspended substitution
Let $t$ have level 3.

$$X[x \mapsto t][X \mapsto x] \leadsto X[X \mapsto x][x \mapsto t[X \mapsto x]]$$

$$\leadsto x[x \mapsto t[X \mapsto x]] \leadsto t[X \mapsto x].$$

$b_j[a_i \mapsto t]$ with $i < j$ is a strong hole $b_j$ waiting to be filled so a weaker $a_i$ can substitute in it.

$[a_i \mapsto t]$ is not a term and cannot be made an argument of a function. Using suspensions we can express it:

$$\lambda X.(X[x \mapsto y]) \text{ encodes } [x \mapsto y].$$

Reduction is not possible because $X$ is stronger than $x$. Contrast this with $\lambda X.(X[X \mapsto y])$, which reduces in one step to $\lambda X.y$. 
Example of substitution-as-a-term

Let \( \text{true} \equiv \lambda x, y.x, \text{false} \equiv \lambda x, y.y, \) and \( \text{ld} \equiv \lambda x.x. \)

\( f \) takes a substitution-as-a-term, a truth value, and an argument, and applies the substitution or not according to the truth value:

\[
f \equiv \lambda x, y, z.(yx \text{ld})z
\]

For example:

\[
f (\lambda X.X[x\rightarrow y]) \text{true} x \rightsquigarrow \text{true}(\lambda X.X[x\rightarrow y])\text{ld}x
\]

\[
\rightsquigarrow^* (\lambda X.X[x\rightarrow y])x \rightsquigarrow^* X[x\rightarrow y][X\leftarrow x] \rightsquigarrow^* y
\]

Similarly, \( f (\lambda X.X[x\rightarrow y]) \text{false} x \rightsquigarrow^* x. \)
\( \lambda \) does not bind. It is an abstractor but not a binder. So we introduce \( \mathcal{N} \) to get \( \alpha \)-equivalence (so, intuitively, \( \mathcal{N}x.\lambda x.blah \) is what we traditionally understand by ‘lambda \( x \) blah’).

(\( \mathcal{N} \) scope-extrudes like in the \( \pi \)-calculus. \( \lambda \) stays put!)

So consider \( \lambda X.\mathcal{N}y.\lambda y.(X[y\leftarrow 0]). \)

Here, \( y \) is generated inside the function call, so we should know \( y \) is not in \( X \) — and reduce to \( \lambda X.\mathcal{N}y.\lambda y.X \).

Freshness contexts \( y \# X \) take care of that. We assume rewrites occur in a context of freshness information with enough fresh variables to satisfy all our needs. Because \( \mathcal{N} \) binds, we can rename \( y \) to some fresh variable which the context says is fresh for \( X \). We then proceed.

Confused? *Let’s make sure you are…*
Reduction rules

\[(\beta) \quad (\lambda a_i.s)u \rightsquigarrow s[a_i \leftarrow u]\]

\[(\sigma a) \quad a_i[a_i \leftarrow u] \rightsquigarrow u \quad \forall c. \ c \# a_i \Rightarrow c \# u\]

\[(\sigma \#) \quad s[a_i \leftarrow u] \rightsquigarrow s \quad a_i \# s\]

\[(\sigma p) \quad (a_i t_1 \ldots t_n)[b_j \leftarrow u] \rightsquigarrow (a_i[b_j \leftarrow u]) \ldots (t_n[b_j \leftarrow u])\]

\[(\sigma \sigma) \quad s[a_i \leftarrow u][b_j \leftarrow v] \rightsquigarrow s[b_j \leftarrow v][a_i \leftarrow u[b_j \leftarrow v]] \quad j > i\]

\[(\sigma \lambda) \quad (\lambda a_i.s)[c_k \leftarrow u] \rightsquigarrow \lambda a_i.(s[c_k \leftarrow u]) \quad a_i \# u, c_k k \leq i\]

\[(\sigma \lambda') \quad (\lambda a_i.s)[b_j \leftarrow u] \rightsquigarrow \lambda a_i.(s[b_j \leftarrow u]) \quad j > i\]

\[(\sigma tr) \quad s[a_i \leftarrow a_i] \rightsquigarrow s\]

\[(\forall p) \quad (\forall n_j.s)t \rightsquigarrow \forall n_j.(st) \quad n_j \not\in t\]

\[(\forall \lambda) \quad \lambda a_i.\forall n_j.s \rightsquigarrow \forall n_j.\lambda a_i.s \quad n_j \neq a_i\]

\[(\forall \sigma) \quad (\forall n_j.s)[a_i \leftarrow u] \rightsquigarrow \forall n_j.(s[a_i \leftarrow u]) \quad n_j \not\in u \ n_j \neq a_i\]

\[(\forall \not\in) \quad \forall n_j.s \rightsquigarrow s \quad n_j \not\in s\]
WHY??
Here is a fun NEW calculus of contexts program (hope you like it):

$$s = \lambda X.((X[x\mapsto y])(X[y\mapsto x])).$$

Observe $s(xy) \sim^\ast (yy)(xx)$.

Thus, the hierarchy of variables allows us to ‘inject’ terms into positions where their variables with be captured, either by a lambda or by an explicit substitution.

Well yes, that’s what you’d expect of a ($\lambda$-)calculus of contexts, really — isn’t it?

We can use this power to model many things.
Records

A record is $b_j[a_{i_1}^1 \leftarrow t_1] \cdots [a_{i_n}^n \leftarrow t_n]$ where $j > i_k$ for $1 \leq k \leq n$.

In words, a record is a set of substitutions suspended on a ‘big hole’ $b_j$.

Let’s be concrete. $R = X[l \leftarrow t_l][p \leftarrow t_p]$ stores $t_l$ at $l$ and $t_p$ at $p$.

We retrieve $t_l$ with $[X \leftarrow l]$ and update it with $[X \leftarrow X[l \leftarrow \text{newval}]]$.

Call $\forall \lambda a_3.a_3[X\leftarrow l]$ record lookup at $l$ and $\forall \lambda a_3.a_3[X\leftarrow X[l\leftarrow t'_l]]$ record update at $l$.

$$
(\forall \lambda a_3.a_3[X\leftarrow l]) R \xrightarrow{(\beta)} \forall a_3.a_3[X\leftarrow l][a_3\leftarrow R]
$$
$$
(\sigma \sigma), (\sigma a)
\xrightarrow{\ast}
\forall a_3.R[X\leftarrow l]
$$
$$
(\sigma \sigma)
\xrightarrow{\ast}
\forall a_3.l[l\leftarrow t_l[X\leftarrow l]][p \leftarrow t_p[X\leftarrow l]]
$$
$$
(\sigma a), (\forall \#)
\xrightarrow{\ast} t_l[X\leftarrow l].
$$

Similar reductions for record update.
Protected records

In-place update is also possible. E.g. \([X \mapsto X[l \mapsto l + 1]]\) (or as a term, \(\forall \lambda a_3.a_3[X \mapsto X[l \mapsto l + 1]]\)) adds 1 to \(t_l\) in-place.

Note that \(X\) is free in \(R\). If we do not like that, we can use protected records.

\[ R' = \forall \lambda X. (X[l \mapsto t_l][p \mapsto t_p]) \]

Protected record lookup at \(l\) is encoded by \(\forall \lambda a_3.a_3 l\) and protected record update at \(l\) by \(\forall \lambda a_3. \forall \lambda X. a_3(X[l \mapsto t'_l])\).

Note there is no possibility of substitution for some stronger variable than \(X\) being captured by \(\lambda X\), because it is protected by \(\forall\), which explicitly marks the scope of \(X\).
Global state, and object-oriented programming

in the sense of general references and Abadi-Cardelli imp-$\varepsilon$ respectively, are easy to encode. For details, see the paper.

Possible applications to the theory of both, e.g. applicative characterisation of contextual equivalence obtained from a theorem which holds of the NEW context calculus (in the paper, a non-trivial result). This is speculative but if it works, it would be great because that kind of theorem is generally very hard.
Partial evaluation

Write

\[
\text{if} = \lambda a, b, c. a b c \quad \text{true} = \lambda a b. a \quad \text{false} = \lambda a b. b \\
\text{not} = \lambda a. \text{if } a \text{ false true }.
\]

in untyped $\lambda$-calculus. Then calculate

\[
s = \lambda f, a. \text{if } a (f a) a \quad \text{specialised to } \quad s \text{ not}
\]

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

A more intelligent method may recognise that the program will always return $\text{false}$ (with types etc.).
Partial evaluation

Choose level 1 variables \( a, b \) and level 2 variables and \( B, C \) and define

\[
\begin{align*}
\text{true} &= \lambda ab.a \\
\text{false} &= \lambda ab.b \\
\text{if} &= \lambda a, B, C. a(B[a\mapsto\text{true}])(C[a\mapsto\text{false}]) \\
\text{not} &= \lambda a.\text{if } a \text{ false true}.
\end{align*}
\]

So if we get to \( B, a = \text{true} \). Consider

\[
s = \lambda f, a.\text{if } a (f a) a \quad \text{specialised to} \quad s \text{ not}.
\]

We obtain:

\[
\begin{align*}
s \text{ not} & \quad \rightsquigarrow^* \lambda a.a ((\text{not}B)[a\mapsto\text{true}][B\mapsto a]) (C[a\mapsto\text{false}][C\mapsto a]) \\
& \quad \rightsquigarrow^* \lambda a.a ((\text{nota})[a\mapsto\text{true}]) (a[a\mapsto\text{false}]) \\
& \quad \rightsquigarrow^* \lambda a.(a \text{ false false}).
\end{align*}
\]

More efficient!
Other applications

**Staged computation** (MetaML, Template Haskell, Converge) offer control execution; a program can suspend its own execution, compose suspended programs into larger (suspended) programs, pass suspended programs as arguments to functions, and evaluate them. This raises issues similar to those surrounding contexts.

Our calculus is a pure rewrite system. *However*, a *programming language* based on it *can* model staged computation.
Other applications

Just the idea: in $s[a \mapsto t]$ restrict evaluations in $t$ to those involving variables at least as strong as $a$. For example $a_3[a_1 \mapsto (\lambda a_2.1)0] \rightsquigarrow a_3[a_1 \mapsto 1]$, and $a_3[a_2 \mapsto (\lambda a_1.1)0]$ does not reduce. $a_3[a_2 \mapsto (\lambda a_2.\lambda a_1.a_2)11 \rightsquigarrow a_3[a_2 \mapsto \lambda a_1.1]$, because $a_2$ is strong enough to reduce under a substitution by $a_2$, but $a_1$ is not.

This to give enough control of execution flow to encode the brackets, escape, and run of MetaML (as well as other less exotic constructs, such as call-by-name and call-by-value versions of function application). Perhaps also retain good meta-theory. Perhaps compare different staged computation (or other) calculi in common language.

We can explore how well provable properties of the NEW context calculus transfer back along maps into it.
There is \( a \)-logic, which has a predicate \( \text{var} \) such that \( \text{var}(x) \) holds when \( x \) is a variable. There is the NEW context calculus, which has contexts. If we put them together, do we get a calculus/logic which can program unification of its own terms?
Meta-properties

- Confluence.
- Preservation of strong normalisation for untyped lambda-calculus.
- Hindley-Milner type system.
- Applicative characterisation of contextual equivalence.
Conclusions

We have ideas of scope as a separate entity from abstraction, from Nominal research, as well as the idea of a freshness context. The calculus can be thought of an operational semantics for heavily souped-up Nominal Terms.

We have a hierarchy of strengths of variables, in common with work by Sato et al.

We have an explicit substitution calculus. This calculus is deliberately simple-minded treating substitutions, e.g. \((\sigma\sigma)\) and \((\sigma p)\). However, the interaction with the hierarchy of variables seems interesting.
Conclusions

Our meta-properties are good and in some directions non-trivial, for example the applicative characterisation of contextual equivalence. The applications seem new, for example our suggestion that calculi of contexts may model state, objects, and even staged computation. Though we make these claims formal, more work clearly remains to be done.

Technically, this work is a logical extension and application of long traditions and techniques in lambda-calculi, explicit substitution calculi, and calculi of contexts, with Nominal techniques applied in a non-trivial but reasonable manner consistent with obtaining certain desired meta-properties.