

Boxology using (hierarchical) nominal techniques

Murdoch J. Gabbay, Heriot-Watt University, Scotland

*St Andrew's University, Scotland
Tuesday 19 December 2006*

Thanks to Kevin Hammond

Boxes

I have been conversing with the Very Excellent Greg Michaelson. I told him about one of my recent grant proposals:

My claim in that proposal was that theoretical computer science is lagging behind current practice, because it does not provide a sufficiently good model of **boxes**.

In the grant proposal I called it “the problem of compositionality”.
Boxology ... composition ... think hi-tec lego.

Greg replied “but yes of course Jamie ... that’s Boxology! Talk to Kevin.”

So I did.

Example: Calculus of Communicating Systems

Notion of box given indirectly by **action prefix** $\alpha.P$. α is an action, $\alpha.$ is the action prefix. P is ‘trapped’ under α .

Explicit notion of composition given by **parallel reduction**.

$$\alpha.P \mid \bar{\alpha}.Q \longrightarrow P \mid Q \quad (\text{Communication}).$$

Here α ‘executes’ by interacting with its **anti-action** $\bar{\alpha}$ (think synchronous communication). This frees P and Q and the system continues as $P \mid Q$.

Example: λ -calculus

Notion of box given indirectly by **function abstraction** $\lambda x.t$. λx is an input to t . t is ‘trapped’ under λx waiting for that input to arrive.

Explicit notion of composition given by **function application**.

$$(\lambda x.t)u \longrightarrow t[u/x] \quad (\beta\text{-reduction})$$

Problems with boxes

CCS: no explicit notion of **ambient**.

λ -calculus: notion of ambient given indirectly by reduction under λ .

λ -calculus: splendid theory of models, given by functions. Splendid logical properties. However, inflexible notion of composition!

CCS: flexible notion of composition. However, models are not hugely interesting; logical properties rather banal.

CCS and λ -calculus: no access to names, in the sense of pointers.

... but you missed out ...

The π -calculus.

The ambient calculus (and ambient logic).

Monads in the typed λ -calculus.

System F.

Dependent types.

Go on! Shout more names at me! Let's do a few rounds!

Further problems with boxes

When is a function that transforms one box into another box, itself a box? When is a collection of boxes a box? What is it to have a name in a box? What is it to move a box from one box to another? When can this happen? How are connections between boxes updated?

Even quite small differences in answers to this question can make massive differences to mathematical properties of the system.

In **syntax** this is not a problem. Just define a convenient syntax and operational semantics, and away you go.

A million operational equivalences and billions of hours of PhD time later, what have we **really** gained?

Syntax of hierarchical nominal terms

For each number $i \geq 1$ fix disjoint countably infinite sets of atoms a_i, b_i, c_i, \dots . Say that a_i has level i .

The syntax of hierarchical nominal terms is inductively defined by

$$t ::= a_i \mid [a_i]t \mid f(t_1, \dots, t_n).$$

f are term-formers.

Syntax of hierarchical nominal terms

Call a_i an **atom of level i** .

a_i can be viewed from level $\leq i$ (below), or from level $> i$ (above).

From below a_i looks like an unknown — a box (‘an unknown term t ’).

From above a_i looks like a constant — a datum (‘the name ‘ t ’ ’).

Syntax of hierarchical nominal terms

$[a_i]t$ is an **abstraction**.

If $j > i$ then $[a_i]b_j$ looks like $\lambda x.t$ or $\forall x.\phi$ or $\int t dx$.

If $j = i$ then $[a_i]b_i$ looks like $\lambda x.y$ or $\bar{a}b$ or $\int y dx$.

if $j < i$ then $[a_i]b_j$ looks like $\lambda x.2$ or $\bar{a}2$ or $\int 2 dx$.

Syntax of hierarchical nominal terms

$f(t_1, \dots, t_n)$ is how we combine elements.

For example: $\lambda, \forall, +, \int$.

Example: substitution

Assume a binary term-former **sub** and sugar **sub** $([a]u, t)$ to $u[a \mapsto t]$.

$$\text{(Sa)} \quad a_i[a_i \mapsto t] \rightsquigarrow t$$

$$\text{(S\#)} \quad a_i \# v \vdash v[a_i \mapsto t] \rightsquigarrow v$$

$$\text{(Saa)} \quad v[a_i \mapsto a_i] \rightsquigarrow v$$

$$\text{(Sf)} \quad f(v_1, \dots, v_n)[a_i \mapsto t] \rightsquigarrow f(v_1[a_i \mapsto t], \dots, v_n[a_i \mapsto t])$$

$$\text{(Sabs>)} \quad ([c_k]v)[a_i \mapsto t] \rightsquigarrow [c_k](v[a_i \mapsto t]) \quad (i > k)$$

$$\text{(Sabs}\leq) \quad b_j \# t \vdash ([b_j]v)[a_i \mapsto t] \rightsquigarrow [b_j](v[a_i \mapsto t]) \quad (i \leq j)$$

Here $k \leq i < j$.

f ranges over all term-formers.

Example: λ

Assume a unary term-former λ and a binary term-former app . Sugar $\text{app}(t, u)$ to tu .

$$(\beta) \quad (\lambda[a_i]t)u \rightsquigarrow t[a_i \mapsto u]$$

I think that a concurrent process calculus is just a feasible.

Example data

Assume a binary term-former \approx and constants \top, \perp .

$$a_i \approx a_i \rightsquigarrow \top \quad a_i \approx b_i \rightsquigarrow \perp$$

However, what $a_i \approx b_j$ reduces to depends on what substitution $[b_j \mapsto t]$ might arrive later.

$b_j[a_i \mapsto t]$ is an explicit substitution, which is itself data.

$\lambda b_j. b_j[a_i \mapsto t]$ is an explicit substitution as a program.

Assume a binary pairing term-former. Then $(b_j, b'_j)[a_i \mapsto t]$ is a synchronisation.

Some conventions

i, j, k will range over nonzero natural numbers.

a_i, b_i, c_i range **permutatively** over atoms of level i .

That is, $a_i \neq b_j$ necessarily. a_i and b_i represent two **distinct** atoms of the same level.

$k \leq i < j$ unless stated otherwise.

Freshness assertions

Call a pair $a_i \# t$ a **freshness assertion**.

The intuition is ' a_i is not free in t '.

When is $a_i \# t$ **true**?

Freshness assertions

Q. When is $a_i \# b_i$?

A. Always. $(x \notin fv(y))$ is always true

Q. When is $a_i \# c_k$?

A. Always. $(x \notin fv(0))$ is always true

Q. When is $a_i \# a_i$?

A. Never. $(x \notin fv(x))$ is never true

Freshness judgements

Q. When is a_i not free in b_j , for $i < j$?

A. When we say so. Because b_j is stronger than a_i , it behaves like t does with respect to a variable symbol x : it depends on what t is.

Freshness assertions

Q. When is $a_i \# f(t_1, \dots, t_n)$?

A. When $a_i \# t_1$ and ... and $a_i \# t_n$.

(Obvious motivation from syntax here.)

Q. When is $a_i \#[a_i]t$?

A. Always.

$(x \notin fv(\forall x.\phi), x \notin fv(\lambda x.t))$

Freshness assertions

Q. When is $a_i \#[b_j]t$ (for $i < j$)?

A. When $a_i \#t$ assuming $a_i \#b_j$.

This has no obvious analogue in ordinary syntax, but consider this: we expect

$$a_i \#[b_j]b_j,$$

but $a_i \#b_j$ is only true if we assume it is true.

Derivation rules for valid freshness assertions

$$\frac{k \leq i}{a_i \# c_k} (\#diff) \quad \frac{a_i \# t_1 \dots a_i \# t_n}{a_i \# f(t_1, \dots, t_n)} (\#f)$$

$$\frac{a_i \# t}{a_i \# [c_k]t} (\#abs \geq) \quad (i \geq k)$$

$$\frac{}{a_i \# [a_i]t} (\#abs =) \quad \frac{\begin{array}{c} [a_i \# b_j] \\ \vdots \\ a_i \# t \end{array}}{a_i \# [b_j]t} (\#abs <) \quad (i < j)$$

Freshness assertions

$[a_i \# b_j]$ denotes **discharge** in the natural deduction sense.

In sequent style (**#abs**<) would be

$$\frac{\nabla, a_i \# b_j \vdash a_i \# t}{\nabla \vdash a_i \#[b_j]t}.$$

Call a freshness assertion **primitive** if it is of the form $a_i \# b_j$ (yes, for $i < j$). Write ∇ for a set of primitive freshness assertions and call it a **(primitive) freshness context**.

Hierarchical nominal rewrite rules

A **hierarchical nominal rewrite rule** is a triple

$$\nabla \vdash l \rightsquigarrow r$$

where ∇ is a primitive freshness context and l and r are terms such that r and ∇ mention only unknowns in l .

The rules for substitution are rewrite rules in this sense.

They give an interesting semantics for atoms as boxes that can mention other boxes — by name.

Conclusions

The NEW calculus of contexts [PPDP'04] is a lambda-calculus with a hierarchy of variable symbols based on similar ideas.

Using the three ideas of

- A hierarchy of variables.
- Freshness.
- Abstraction.

we can represent a great richness of structure.

Conclusions

“When is a function that transforms one box into another box, itself a box? When is a collection of boxes a box? What is it to have a name in a box? What is it to move a box from one box to another? When can this happen? How are connections between boxes updated?”

Depends on the rewrite rules. But we now have a powerful mathematical handle on what we mean when we say ‘box’.