FM Techniques for Syntax-with-Binding

Murdoch J. Gabbay, October 2002

Substitution as computation

The syntax of a simple λ-calculus:

(1) $$t ::= x \in \mathbb{A} \mid t\,t \mid \lambda x.t \mid 0 \mid \mathrm{Succ}(t) \mid t + t.$$

Evaluation rules:

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t}$$

$$\frac{t_1 \Downarrow \lambda x.s \quad t_2 \Downarrow V \quad s[V/x] \Downarrow W}{t_1 t_2 \Downarrow W} \qquad \ldots$$

Thus we see that substitution models computation. Do we have a good definition of substitution?

Capture-avoiding substitution (not rigorous!)

$$
\begin{array}{lll}
[s/x]x & \stackrel{\text{def}}{=} & s \\
[s/x]y & \stackrel{\text{def}}{=} & y \\
[s/x](t_1\ t_2) & \stackrel{\text{def}}{=} & [s/x]t_1\ [s/x]t_2 \\
[s/x]\lambda x.t & \stackrel{\text{def}}{=} & \lambda x.t \\
[s/x]\lambda y.t & \stackrel{\text{def}}{=} & \lambda y'.[s/x][y'/y]t \quad \text{for } y' \text{ fresh.}
\end{array}
\tag{2}
$$

Last clause: substitution capture-avoiding. Thus for $t = x$ and $s = y$,

$$[y/x]\lambda y.x \neq \lambda y.y \quad \text{but} \quad [y/x]\lambda y.x = \lambda y'.y$$

for some/any fresh y'. What does 'fresh' mean? Not rigorous mathematical specification, and cannot be programmed.

Problem appears elsewhere

Similar phenomenon for, say, inlining optimiser:

```
let y=5 in
   let x=f(y) in
      let y=7 in <x,x,y>
```

rewrites to

```
let y=5 in
   let y'=7 in <f(y),f(y),y'>
```

Note here how we rename $\mathtt{y}$ to $\mathtt{y}'$ in the inner binding, so as not to accidentally capture it.

One solution: de Bruijn

Usual solution: de Bruijn indeces. Variables represented by numbers with (what I shall call) a "λ-offset". Thus $\lambda x.x$ (and $\lambda y.y$, $\lambda z.z$, ...) represented by $\lambda 1$; 1 is a pointer to the first λ above it. $\lambda x.\lambda y.\lambda z.y$ is represented by $\lambda\lambda\lambda 2$.

De Bruijn presupposes a canonical ordering on the variable names; $x_1, x_2, x_3, \ldots$. Thus $\lambda 4$ represents $\lambda x.x_3$.

Thus α-equivalence is literal equality and the problems of the last slides are eliminated.

Problem with de Bruijn

. . . to be replaced by anotehr: we sacrifice the natural inductive structure of terms. E.g. a subterm of $\lambda 4$ is 4, which represents x_4 on its own but x_3 as it is under a single λ, but x_2 when under two, and so on. Papers and programs using de Bruijn use (choke on) "shift functions" to correctively increment and decrement indeces.

For example substitution $[s/x]t$ becomes a complicated and possibly computationally very expensive function, since precisely what gets substituted for x in t depends very much on how many λs we have traversed. Indeed, the index representing x itself in not uniform.

Binding is everywhere

- Syntax and operational semantics (e.g. λ-calculus, PCF, $\lambda\sigma$-calculus, π-calculus . . .).
  Difficulties with defs: capture-avoiding substitution thus evaluation (worse in $\lambda\sigma$ because internalised), type weakening, . . .
- Implementations of logics and calculi (Isabelle, HOL, abstract machines for calculi above . . .).
  Difficulties with defs & algorithms: de Bruijn datatypes with shift functions make papers and programs unreadable and possibly bugged, name-carrying terms force hand-coding of α-equivalence, abstract machines rendered complex, . . .
- Denotational semantics (models of π-calculus processes).

We are all *familiar* with binding, but familiarity is not understanding.

Syntax and Semantics

תורה	$\longrightarrow$	גמרא
Syntax	$\longrightarrow$	Semantics

Syntax is the object of study here, and we are trying to choose a good one: if The Word is lousy, we're at best wasting our time and at worst damned to eternal torture (this point also observed by other authors).

Usual Model of Syntax

The usual model of syntax is labelled trees. Consider the λ-calculus for arithmetic in (1):

$$t::= x \in \mathbb{A} \mid t\,t \mid \lambda x.t \mid 0 \mid \mathrm{Succ}(t) \mid t + t.$$

We list the constructors var, app (both sugared above to be nameless), λ, 0, Succ, $+$ and assign $\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inl}(-)))))$ to var, $\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inr}(\langle - : T, - : T\rangle)))))$ to app, $\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inl}(\mathbf{Inr}(\mathbf{Inr}(\langle - : \mathbb{A}, - : T\rangle)))))$, and so on. Here T is the type of syntax trees which we are defining.

This is an inductive definition of syntax, standard practice. As we have observed, it does not interact well with binding.

We need:

Henceforth **variable names** will be called **atoms**. We require support for:

- "Choose a fresh atom" (recall, 'for y' fresh' in the informal definition of substitution).
- "Call the bound atom in an abstraction x" (recall, '$\lambda 1$ represents $\lambda x.x$'; we are actually choosing to call it x).
- "Bind x in t" (recall, 'form $\lambda x.t$ from t').

This allows us to, for example, give semantics to a program such as

```
case x' of <n>x => <n>f(x)
```

This is the capture-avoiding application of `f` under the binder: n is chosen fresh, f applied to the body, then n is rebound.

Capture-avoiding substiution (rigorous)

Capture-avoiding substitution becomes:

```
sub s a var(a) => s
sub s a var(b) => b
sub s a app(t1,t2)
                => app(sub s a t1,sub s a t2)
sub s a lam(<n>t)
                => lam(<n>(sub s a t))
```

But what is the type system of such a language; we must ensure the fresh choice of n does not escape its scope. What about the interaction with state? This is the FreshML project.

FM offers a semantics for binding which illuminates the phenomenon it models. It guides us to type systems, logics, and other formal environments, with facilities such as the three listed in Slide 10.

What is FM?

- "FM sets" stands for "Fraenkel-Mostowski set theory". They are innocent; presented in [GabbayMJ:thesis], [GabbayMJ:newaas], [GabbayMJ:newaas-jv].
- FM has become an overall label for logics and programming languages developed using FM sets. Thus for example FreshML and Pitts' Nominal Sets.
- My latest baby: FMG. Stands for Fraenkel-Mostowski Generalised. Primitive version presented in [GabbayMJ:hotn], full version in [GabbayMJ:picfm] (pending publication) with more to follow.
- FMG is a Higher-Order Logic (HOL) and can be thought of as a HOL version of FM sets. However, the presentation is much cleaner and the system strictly more powerful.

These slides present the special case of FMG corresponding to FM.

The Six Rules of the Righteous Binder

Hypothesise **type of atoms** $\mathbb{A}$. Definitionally extend with type of **permutations** $P_{\mathbb{A}}$, bijective $f : \mathbb{A} \to \mathbb{A}$. Axioms:

(Act-$\mathbb{A}$)	$\pi \cdot a$	$= \pi(a : \mathbb{A})$
(Act-$\circ$)	$\pi \cdot \pi' \cdot x$	$= (\pi \circ \pi') \cdot x$
(Act-**Id**)	**Id** $\cdot\, x$	$= x$
(Eqv1)	$\pi \cdot f(x)$	$= (\pi \cdot f)(\pi \cdot x)$
(Eqv2)	$\pi \cdot c$	$= c$ $\quad c$ a closed term

(Small) $\quad \exists A \in \mathbb{A}^{\mathcal{S}}.\ A\ supports\ x$

Here $\circ$ denotes function composition. $\pi(a)$ denotes value of π as a function at a. $\mathbb{A}^{\mathcal{S}} \subseteq \mathcal{P}(\mathbb{A}) \stackrel{\text{def}}{=} \mathbb{A} \to \mathbb{B}$ is finite sets $\mathcal{P}_{fin}(\mathbb{A})$.

The Plan

The axioms above define a Higher-Order Logic (set-theoretic version exists too, if we prefer). These foundational systems can be used to interpret functions, state spaces, models, graphs, algorithms, automata, programming languages, logics, and so on. We interpret (say) a programming language in an FM universe and use the extra FM structure to encode binding. We use this encoding of binding to guide us in the design of, say, a pattern-matching discipline and type system which provides, in a safe and useful way, the facilities of Slide 10. From the theory emerges useful practice.

It seems to work. For example, FreshML. Other work being developed too, for example Pitts' Nominal Logic, my recent work on the π-calculus, and work by Urban Pitts and myself on unification of logics-with-binding.

Permutation

We return to the axioms, addressing the first five. $\mathbb{A}$ now has a distinguished existence, it contains the variable names $x, y, z, a, b, c, \aleph, \beth, \ldots$. For any element t we can apply a permutation π to t: $\pi \cdot t$. We do not know what this is, but we do know it is a permutation action from (Act-$\circ$) and (Act-**Id**). Thus $\pi \cdot \pi^{-1} \cdot t = t$:

$$\pi \cdot \pi^{-1} \cdot t = (\pi \circ \pi^{-1}) \cdot t = \mathbf{Id} \cdot t = t.$$

This action is coherent with the natural permutation action on $\mathbb{A}$ by (Act-$\mathbb{A}$). Thus

$$[a \mapsto b, b \mapsto a] \cdot a : \mathbb{A} = b.$$

The permutation $[a \mapsto b, b \mapsto a]$ is written $(a\ b)$ in FM work, and called **transposition**.

Equivariance

(Eqv1) and (Eqv2) together ensure **equivariance**: if F has free variables $x_1, \ldots, x_n$ (*different* xs from those in $\mathbb{A}$!) then

$$\pi \cdot F(x_1, \ldots, x_n) = F(\pi \cdot x_1, \ldots, \pi \cdot x_n).$$

For example:

$$\begin{aligned} \pi \cdot \langle x_1, x_2 \rangle &= \pi \cdot \big((\lambda x, y.\langle x, y \rangle x_1) x_2\big) \\ &= (\pi \cdot (\lambda x, y.\langle x, y \rangle x_1))(\pi \cdot x_2) \\ &= (\pi \cdot \lambda x, y.\langle x, y \rangle)(\pi \cdot x_1)(\pi \cdot x_2) \\ &= (\lambda x, y.\langle x, y \rangle)(\pi \cdot x_1)(\pi \cdot x_2) \\ &= \langle \pi \cdot x_1, \pi \cdot x_2 \rangle \end{aligned} \tag{3}$$

Similarly for **Inl**$(-)$ and **Inr**$(-)$. Thus these axioms accurately model the way substitution distributes through tree structure of formal grammar.

Support

Let's use these axioms to build a notion of "the free variables of x":

$$A\ supports\ x \stackrel{\text{def}}{=} \forall \pi.\ \pi \in Fix(A) \implies \pi \cdot x = x$$

$$Fix(A \subseteq \mathbb{A}) \stackrel{\text{def}}{=} \left\{ \pi \mid \forall a \in A.\ \pi \cdot a = a \right\}$$

$$S(x) \stackrel{\text{def}}{=} \bigcap \left\{ A \in \mathbb{A}^{\mathcal{S}} \mid A\ supports\ x \right\}.$$

Theorem 1: $S(x)\ supports\ x.$

Example

1. Syntax: $S(\mathbf{Inl}(z)) = S(z),\ S(\langle z, z' \rangle) = S(z) \cup S(z')$.
2. Syntax: $S(\lambda a.t) = \{a, S(t)\}$, where here λ is treated as an ordinary constructor, so $\lambda a.t$ is modelled by $\mathbf{Inl}(\ldots \langle a, t \rangle)$ and $(c\ a) \cdot \lambda a.t = \lambda c.(c\ a) \cdot t$.
3. $\{b\}$ does not support $\lambda a.b$.
4. $\{b\}$ *does* support $[\lambda a.b]_{=_\alpha}$, α-equivalence class, because $\lambda a.b =_\alpha \lambda a'.b$.

Of course $(a\ b) \cdot \lambda a.b = \lambda b.a \neq_\alpha \lambda a.b$ but $(a\ b) \notin Fix(\{b\})$.

The axiom (Small)

Note we have the notion of support of functions, algorithms, models, and so on, because permutation is defined on everything. Axiom (Small) insists that every element has a finite number of free variable names:

(Small) $$\exists A \in \mathbb{A}^{\mathcal{S}}.\ A\ supports\ x$$

Recall $\mathbb{A}^{\mathcal{S}} = \mathcal{P}_{fin}(\mathbb{A})$. Theorem 1 states that there exists a unique smallest such A and this is $S(x)$ the support of x.

We can now give a semantics to 'fresh for': $a\#t$ when $a \notin S(t)$. In this talk however we pursue the И-quantifier, a binder for 'choose fresh n'.

И Quantifier

The И ('new') quantifier in FM is defined by:

$$И a.\, P(a) \stackrel{\mathrm{def}}{=} \exists L \in \mathbb{A}^{\mathcal{L}}.\, \forall a \in L.\, P(a).$$

Here $\mathbb{A}^{\mathcal{L}}$ is $\mathcal{P}_{cofin}(\mathbb{A})$, the set of cofinite sets of atoms. L is cofinite when $\mathbb{A} \setminus L$ is finite.

From properties of finite and cofinite sets it follows

(4) $$И a.\, P(a)\ op\ И a.\, Q(a) \iff И a.\, (P(a)\ op\ Q(a))$$

for op one of $\wedge$ and $\vee$. Consider for example $P(a) = a \in FV(s)$ and $Q(a) = a \in FV(t)$ for s, t in some datatype of terms. Note how semantics gives rise to a new entity И in a new logic.

Sets of atoms

Lemma 2: For $X \subseteq \mathbb{A}$, $S(X) = S(\mathbb{A} \setminus X)$.

Proof. $\pi \cdot X = X$ if and only if $\pi \cdot (\mathbb{A} \setminus X) = (\mathbb{A} \setminus X)$.

Lemma 3: For $X \subseteq \mathbb{A}$, X and $\mathbb{A} \setminus X$ support X.

Proof. If π fixes every element of X pointwise then clearly $\pi \cdot X = X$. If π fixes every element outside X, then it can only permute elements within X so $\pi \cdot X = X$.

Corollary 4: Every $X \subseteq \mathbb{A}$ is either finite or cofinite:

$$\mathcal{P}(\mathbb{A}) = \mathcal{P}_{fin}(\mathbb{A}) + \mathcal{P}_{cofin}(\mathbb{A}).$$

И and Negation

People often find Corollary 4 counter-intuitive: “what happened to the rest of the powerset then?”. It does not exist in the FM universe, though of course it exists in a model of FM inside another, traditional, universe.

This ensures that

(5) $$И a.\, P(a) \Rightarrow И a.\, Q(a) \iff И a.\, (P(a) \Rightarrow Q(a))$$

(6) $$И a.\, \neg P(a) \iff \neg И a.\, P(a).$$

This forms part of a series of excellent properties enjoyed by И. We have a new logic, and we used FM semantics to develop it. A simple example of the joys possible.

Example

We now apply this to give an improved definition of α-equivalence $=_\alpha$ for a datatype of λ-terms

$$\Lambda \stackrel{\text{def}}{=} \mathtt{V}\ \textbf{of}\ \mathbb{A} + \mathtt{A}\ \textbf{of}\ \Lambda \times \Lambda + \mathtt{L}\ \textbf{of}\ \mathbb{A} \times \Lambda.$$

$=_\alpha$ is given by

$$\begin{array}{lll} & \mathtt{V}(a) =_\alpha \mathtt{V}(b) & \Leftrightarrow \quad a = b \\ (7) \quad & \mathtt{A}(s_1, s_2) =_\alpha \mathtt{A}(t_1, t_2) & \Leftrightarrow \quad s_1 =_\alpha t_1 \wedge s_2 =_\alpha t_2 \\ & \mathtt{L}(a, s) =_\alpha \mathtt{L}(b, t) & \Leftrightarrow \quad И c.\ (c\ a) \cdot s =_\alpha (c\ b) \cdot t. \end{array}$$

(Extra joy)

Transposition

We used transposition to define $=_\alpha$. We could use substitution $[n_2/n_1]$ in (7) but transposition has better properties. For example, it respects $=_\alpha$:

$$s =_\alpha t \iff (a\ b) \cdot s =_\alpha (a\ b) \cdot t.$$

For example:

$$\begin{array}{lll} & \mathtt{L}(f, \mathtt{L}(x, \mathtt{A}(f, x))) =_\alpha \mathtt{L}(f', \mathtt{L}(x', \mathtt{A}(f', x'))) & \\ (8) \quad & \mathtt{L}(x, \mathtt{L}(f, \mathtt{A}(x, f))) =_\alpha \mathtt{L}(f', \mathtt{L}(x', \mathtt{A}(f', x'))) & (f\ x) \\ & \mathtt{L}(f, \mathtt{L}(f, \mathtt{A}(f, f))) \neq_\alpha \mathtt{L}(f', \mathtt{L}(x', \mathtt{A}(f', x'))) & [f/x] \end{array}$$

Not just λ-calculus

We have used the λ-calculus as a running example. Do not leave with the impression that this talk is about the λ-calculus. Far from it. In my most recent paper (with the referees) I apply sophisticated versions of these techniques to analyse the syntax and semantics of the π-calculus, a process calculus for modelling distributed, communicating systems which generate fresh tokens during computation.

We can analyse syntax *and* semantics because the semantics of the π-calculus is syntax-based, and can suffer very badly from problems caused by binding. The semantics in question are designed for model-checking and verification (e.g. of bisimulations).

In my paper I claim to have brought order to the confusion, and to have opened the door to using FM languages to code novel and effective algorithms on the models I build.

Example $\neq$ the λ-calculus:

Proving weakening for some type system:

(9) $\forall \Gamma, t, x.\ \Gamma \vdash t : T \ \wedge\ x \notin Dom(\Gamma) \implies \Gamma, x : X \vdash t : T.$

An inductive proof on $\vdash$ interacts badly with a rule such as

$$\frac{\Gamma, x : X \vdash t : T}{\Gamma \vdash \mathtt{L}(x, t) : X \to T} \quad x \notin Dom(\Gamma)$$

because we can weaken the conclusion with x but not the assumption.

(10) $\forall \Gamma, t.\ И x.\ \Gamma \vdash t : T \implies \Gamma, x : X \vdash t : T$

is easy to prove (using properties of И from Slides 20 and 22). Γ is finite so И provides a 'context-less' version of $x \notin Dom(\Gamma)$. (9) iff (10) thanks to *equivariance*, which for predicates is $\Phi((a\ b) \cdot t) \iff \Phi(t)$.

Final slide

We can also improve on the rules themselves. Γ is finite (of course) so И provides a 'context-less' version of $x \notin Dom(\Gamma)$:

$$\forall \Gamma.\ И x.\ \forall t.\ \frac{\Gamma, x : X \vdash t : T}{\Gamma \vdash \mathtt{L}(x,t) : X \to T}$$

This system is even easier to work with, as I verify for the π-calculus.

I have not mentioned at all how to improve on the datatypes themselves, that is, produce a model such that $\lambda x.x = \lambda y.y$, like de Bruijn, and yet where t is a subterm of $\lambda x.t$ (up to the choice of name for the bound variable). These are abstraction types $[\mathbb{A}]T$. These are wonderful, but they belong to another talk.

Take-home message

- I upheld a noble tradition and lied about the final slide.
- Binding is everywhere. Not all people who have problems with binding identify this as the problem. Anywhere where we create new tokens (nonces, memory locations, etc.), use them, then throw them away, there is a possibility to apply FM techniques.