A Metalanguage for Programming with Bound Names Modulo Renaming

Andrew Pitts   Jamie Gabbay

UNIVERSITY OF CAMBRIDGE

Sources: paper in proceedings volume; Gabbay-Pitts paper in Proc. LICS'99; Gabbay's PhD thesis (forthcoming).

The problem: representing languages with binders in functional programming languages

Classical theory of algebraic data types (incl. de Bruijn ‘nameless’ terms) is too concrete and user-unfriendly.

Use of typed lambda calculus (à la Martin-Löf, HOAS, etc) is too abstract: one looses (simple) concepts of structural recursion/induction.

`FreshML`—novel features

- Type of atoms whose values represent object-level variables (cf. `ML`'s `unit ref`) and types of atom-abstractions—more concrete than function abstractions (they support pattern-matching and reasoning by structural induction).

- 1-1 representation of alpha-equivalence classes of object-level terms as values of *concrete* data types. 'Nameful' rather than nameless representations, with type system ensuring that well-typed programs respect renaming of bound object-level variables (via freshness inference).

There's a simple underlying **mathematics** of these new features for **program construction**: FM-sets (= sets equipped with an abstract notion of swapping atoms, roughly speaking).

A paradigmatic example: abstract syntax trees of lambda-terms and capture-free substitution in ML

```
datatype  ltree  =  Vr of string
                 |  Ap of ltree*ltree
                 |  Lm of string*ltree
```

```
fun  sb t x (Vr y) = if y=x  then t
                             else Vr y
 |   sb t x (Ap(u,v)) = Ap(sb t x u,
                           sb t x v)
 |   sb t x (Lm(y,u)) = Lm(y, sb t x u)
```

`sb` codes the essence of capture-free substitution for lambda-terms, but is only partially correct (because . . .).

As an implementation of *capture-free substitution* of `t` for `(Var x)` in `u`, `sb t x u` is only partially correct because of the last clause: it doesn't take into account the condition which we always arrange to hold in practice when making a substitution—namely that `y` is 'fresh' for `x` and `t`.

By contrast, the `FreshML` language computes freshness information and ensures that it's version of `sb` respects the above convention. Here is the `FreshML` version of the code on Slide 4...

A paradigmatic example: alpha-equivalence classes of lambda-terms and capture-free substitution in FreshML

```
datatype lam  =  Var of atm
              |  App of lam*lam
              |  Lam of [atm]lam
```

```
fun  sub t x (Var y) = ···
 |   sub t x (App(u,v)) = ···
 |   sub t x (Lam a.u) = Lam a.(sub t x u)
```

Claim: values of type `lam` denote lambda-terms modulo alpha-conversion; and `sub:lam -> atm -> lam -> lam` implements capture-free substitution.

`FreshML` code for `lam` and `sub` contains examples of the following `FreshML` features:

- a built-in type of atoms, `atm`
- atom-abstraction types, like `[atm]lam`
- abstraction expressions, like `a.(sub t x u)`
- matching with abstraction patterns like `a.u`

Will explain their meaning in terms of the FM-sets model and sketch proof of **Claim**.

Recall usual definition of alpha-conversion, $=_\alpha$, as smallest congruence relation on values of $\mathtt{ltree}$ containing $\mathtt{Lm(x,V)} =_\alpha \mathtt{Lm(y,V[y/x])}$. (where $\mathtt{V}[\mathtt{y}/\mathtt{x}]$ = capture-free substitution of $\mathtt{y}$ for $\mathtt{x}$ in $\mathtt{V}$).

Theorem. $=_\alpha$ coincides with relation $\sim$ inductively generated by

$$\mathtt{(Vr\ x)} \sim \mathtt{(Vr\ x)} \qquad \frac{\mathtt{U} \sim \mathtt{U}' \quad \mathtt{V} \sim \mathtt{V}'}{\mathtt{Ap(U,V)} \sim \mathtt{Ap(U',V')}}$$

$$\frac{(\mathtt{z}\ \mathtt{x}) \cdot \mathtt{U} \sim (\mathtt{z}\ \mathtt{y}) \cdot \mathtt{V}}{\mathtt{Lm(x,U)} \sim \mathtt{Lm(y,V)}} \text{ if } \mathtt{z} \text{ fresh for } \mathtt{x}, \mathtt{U}, \mathtt{y}, \mathtt{V}$$

$(\mathtt{y}\ \mathtt{x}) \cdot \mathtt{V}$ = result of swapping *all* occurrences of $\mathtt{y}$ and $\mathtt{x}$ in $\mathtt{V}$.
N.B. don't need to know which constructors are binders to define this.

The Theorem suggests how to make a purely inductive definition of lambda-terms mod alpha-equivalence, by working with sets equipped with an abstract notion of swapping atoms . . .

FM-sets

are sets X equipped with an abstract notion of swapping atoms, $x \in X \mapsto (a\ b) \cdot x \in X$ (where a, b range over a fixed infinite set of 'atoms'), satisfying

$$(a\ a) \cdot x = x = (a\ b) \cdot ((b\ a) \cdot x)$$

plus preservation properties, like

$$(a\ b) \cdot (f\ x) = ((a\ b) \cdot f)\,((a\ b) \cdot x)$$

plus an important *finite support* property . . .

[FM = Fraenkel-Mostowski
cf. their permutation model of ZFA from 1930s.]

An abstract notion of freshness

Define

$$a \mathbin{\#} x \qquad \text{('}a\text{ is fresh for }x\text{')}$$

to mean that $(a\ b) \cdot x = x$ holds for all but finitely many atoms b.

Finite support axiom. All elements x of an FM-set must satisfy that $a \mathbin{\#} x$ holds for all but finitely many atoms a.

[Check: for lambda-terms t using atoms as variables,
$a \mathbin{\#} t$ holds iff a does not occur in syntax tree of t
and this is true for all but finitely many a.]

Atoms are not constants

because they have different ‘swapping’ characteristics:

atoms a satisfy $(a_1\ a_2) \cdot a =$
$\mathtt{if}\ a = a_1\ \mathtt{then}\ a_2\ \mathtt{elseif}\ a = a_2\ \mathtt{then}\ a_1\ \mathtt{else}\ a$

whereas

constants k (numbers, characters, strings, etc) satisfy
$(a_1\ a_2) \cdot k = k$.

In `FreshML`, we use the type `atm` of atoms to represent object-level variables and types like `string` to represent object-level constants.

Variable-binding is represented via the following notion of atom-abstraction . . .

FM-set $\mathtt{[atm]X}$ of atom-abstractions

Introduction: $\boxed{a\,.\,x} \in \mathtt{[atm]X}$, if $a \in \mathtt{atm}$ and $x \in \mathtt{X}$.

Equality: $a\,.\,x = b\,.\,y \in \mathtt{[atm]X}$ iff $(c\ a) \cdot x = (c\ b) \cdot y \in \mathtt{X}$, for some/any $c \in \mathtt{atm}$ with $c \,\#\, (a, x, b, y)$.

Swapping: $(a_1\ a_2) \cdot (a\,.\,x) = ((a_1\ a_2) \cdot a)\,.\,((a_1\ a_2) \cdot x)$.

(Hence $a' \,\#\, a\,.\,x$ iff $a' = a$ or $a' \,\#\, x$.)

Atom-abstraction, cont.

Elimination: given $f \in \texttt{(atm * X -> Y)}$ satisfying

$$\forall a \in \texttt{atm}, x \in \texttt{X}\,(a \;\texttt{\#}\; f \Rightarrow a \;\texttt{\#}\; f\,\texttt{(}a, x\texttt{)})$$

then for any $e \in \texttt{[atm]X}$ there is a unique element

$$\boxed{\texttt{case}\; e \;\texttt{of}\; \{a\,\texttt{.}\,x \;\texttt{=>}\; f\,\texttt{(}a, x\texttt{)}\}} \in \texttt{Y}$$

satisfying

Reduction: for all $b \in \texttt{atm}$, $y \in \texttt{X}$,
$\texttt{case}\; b\,\texttt{.}\,y \;\texttt{of}\; \{a\,\texttt{.}\,x \;\texttt{=>}\; f\,\texttt{(}a, x\texttt{)}\} = f\,\texttt{(}c, (c\;b) \cdot y\texttt{)}$,
for some/any $c \;\texttt{\#}\; (f, b\,\texttt{.}\,y)$.

Putting this altogether we get an inductive FM-set of lambda-terms mod alpha-equivalence . . .

An inductive FM-set of lambda-terms mod α-equivalence

```
datatype lam = Var of atm
             | App of lam*lam
             | Lam of [atm]lam
```

Recall characterisation of $=_\alpha$ for `ltree` as $\sim$.
Analogous definition for `lam`

$$\mathtt{(Var\ a)} \sim \mathtt{(Var\ a)} \qquad \frac{\mathtt{U} \sim \mathtt{U}' \quad \mathtt{V} \sim \mathtt{V}'}{\mathtt{App}(\mathtt{U},\mathtt{V}) \sim \mathtt{App}(\mathtt{U}',\mathtt{V}')}$$

$$\frac{(\mathtt{c}\ \mathtt{a}) \cdot \mathtt{U} \sim (\mathtt{c}\ \mathtt{b}) \cdot \mathtt{V}}{\mathtt{Lam}\ \mathtt{a.U} \sim \mathtt{Lam}\ \mathtt{b.V}} \text{ if } \mathtt{c}\ \#\ (\mathtt{a},\mathtt{U},\mathtt{b},\mathtt{V})$$

just yields $=$ (because of the definition of $\mathtt{a.U} = \mathtt{b.V}$).

Can prove this, and many other things about `lam`, using a form of structural induction . . .

Structural induction for `lam`

Given $\phi \in$ `(lam -> bool)`, to prove $\phi(\texttt{V})$ holds for all $\texttt{V} \in \texttt{lam}$, suffices to show

- $\phi(\texttt{Var a})$, for all $\texttt{a} \in \texttt{atm}$
- $\phi(\texttt{U})$ & $\phi(\texttt{V}) \Rightarrow \phi(\texttt{App(U,V)})$, for all $\texttt{U}, \texttt{V} \in \texttt{lam}$
- $\phi(\texttt{U}) \Rightarrow \phi(\texttt{Lam a.U})$, for some/any $\texttt{a \# } \phi$ and all $\texttt{U} \in \texttt{lam}$.

This 'for some/any fresh $a \ldots$' keeps on cropping up! It is the $\mathsf{N}a\,(\ldots)$ quantifier of (Gabbay-Pitts, LICS'99) and has very pleasant logical properties.

The $\texttt{FreshML}$ type system combines conventional typing with freshness inference.

E.g. to deduce $\texttt{case}\ e\ \texttt{of}\ \{a\,\texttt{.}\,x\ \texttt{=>}\ f\,\texttt{(}a, x\texttt{)}\} \in \texttt{Y}$, requires not only $e \in \texttt{[atm]X}$ and $f \in \texttt{(atm * X -> Y)}$, but also $\forall a \in \texttt{atm}, x \in \texttt{X}\,(a\ \texttt{\#}\ f \Rightarrow a\ \texttt{\#}\ f\,\texttt{(}a, x\texttt{)})$.

Difficulty: semantic freshness '$a\ \texttt{\#}\ f$' is in general undecidable (because . . .). $\texttt{FreshML}$ typing is decidable (good!), so only gives an approximation to $\texttt{\#}$, so does not type all semantically meaningful expressions. E.g. . . .

List of free variables, I

```
fun  fv (Var a) = a::nil
 |   fv (App(t,u)) = append(fv t)(fv u)
 |   fv (Lam a.t) = rem a (fv t);
```

where `rem : atm -> (atm list) -> atm list`

is . . . [obvious definition] . . .

`fv` *does not type check*! — current FreshML type system does not prove `a # (rem a (fv t))`, even though it's true in the FM-sets model.

In this case there's a work-around involving changing `rem` . . .

List of free variables, II

```
fun  fv2(Var a) = a::nil
 |   fv2(App(t,u)) = append(fv2 t)(fv2 u)
 |   fv2(Lam a.t) = rem2 a.(fv2 t);
```

where `rem2 : [atm](atm list) -> atm list` is

```
fun  rem2 a.nil = nil
 |   rem2 a.(b::bs) = ifeq(a,b)
                        then (rem2 a.bs)
                        else b::(rem2 a.bs)
```

In this case we do get `fv2 : lam -> atm list`
(and it does have the intended semantics).

The code for `rem2` illustrates `FreshML`'s treatment of *atom-equality testing*. It is non-trivial to give this in a form that allows good freshness-inference in the type system. The `ifeq(a,b) ...` conditional has a typing rule with better freshness properties than the normal combination of conditional and equality test, `if a = b ...` (see rule (15) in the paper in the conference proceedings). `FreshML` also uses *equality and inequality guards* in matches to get the same strength of freshness inference in, arguably, a more convenient form.

Summary

inductively defined FM-sets using atom-abstraction
provide a model of
abstract syntax modulo bound-variable renaming
that strikes a good balance between the inconveniently concrete
machinery of ordinary algebraic data types and the
inconveniently abstract machinery of HOAS.

Witness the fact that the FM-sets model gives rise to a language like `FreshML` that retains the flavour of the conventional, concrete approach of `ML` while providing automatic support for renaming, freshness, etc.

To do

- Practical experience with `FreshML`: do its novel features provide a good idiom for syntax-manipulating algorithms (and proofs about them)?

  Probably need generalisations of $\mathtt{[atm]X}$: e.g. $\mathtt{[A]X}$ for types other than $\mathtt{A} = \mathtt{atm}$

- More expressive type systems that better approximate $\mathtt{\#}$.

- Can `FreshML` be implemented *efficiently*?

- Logical frameworks based on FM-sets model (e.g. using dependently typed generalisation of atom-abstraction, $\mathtt{[a} \in \mathtt{atm]X(a)}$).