A Meta Language for Programming with Bound Names Modulo Renaming

Andrew M. Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

An account of work in progress
in collaboration with Jamie Gabbay

Copy of these slides at
$\langle$`http://www.cl.cam.ac.uk/users/amp12/talks/`$\rangle$

Abstract

This talk describes work in progress (with Jamie Gabbay) on the design of an ML-style metalanguage, FML, for programming recursively defined functions on user-defined data types whose constructors may involve binding. Object language terms modulo alpha-conversion can be faithfully represented up to operational equivalence as values of suitable FML data types. The design of FML is 'semantically driven', in as much as it arises out of contemplating the model of variable-binding in terms of Fraenkel-Mostowski sets given in [Gabbay and Pitts, Proc. LICS '99, pp 214–224]. Compared with ML, the language has a novel type constructor for abstractions; facilities for declaring fresh names; and recursive definitions can use a form of pattern-matching on bound names of abstractions. The crucial point is that the FML type system ensures that these features can only be used in well-typed programs in ways which are insensitive to renaming of bound names.

Recent work on metamathematics of abstract syntax involving static binders by Gabbay-AMP, Fiore-Plotkin-Turi, Hofmann (see Proc. LICS'99) stems from observation that

can use initial algebra semantics for

$$\left(\begin{array}{c}\text{inductive set of} \\ \text{abstract syntax trees}\end{array} \right) \Big/ \ \alpha\text{-conversion}$$

provided we work with *variable* sets,

i.e. in various categories of (pre)sheaves.

Here: consequences of this observation for programming language design (specifically, user-declared data types).

FML — an ML-style language for meta-programming

- Nameful, rather than nameless, representation of object languages as user-declared data types whose constructors can involve (statically scoped) name-binding via use of abstraction types. Can still reason about data values using a form of structural induction.

- Local declaration of fresh names, and pattern-matching in definitions using abstraction patterns. (The two notions are connected.)

- Type system ensures that well-typed programs respect renaming of bound names.

- Language design is 'semantically-driven' — by Gabbay-Pitts model of name binding (LICS'99): names denote atoms in FM set theory, etc.

Naïve denotational semantics

ML types and their values	$\mapsto$	sets and their elements
FML types and their values	$\mapsto$	FM-sets and their elements

To be less naïve requires use of domain theory (in categories of FM-sets).

For relevant info on FM-sets see Gabbay & Pitts, Proc. LICS'99, pp 214–224.

Here: will describe FML by example, glimpsing its type system and operational semantics.

To motivate the novelties of FML, we look at the paradigmatic example: encoding capture-free substitution for terms of the untyped lambda calculus.

Abstract syntax trees in ML—a paradigmatic example

```
datatype lam = Var of string
             | App of lam*lam
             | Lam of string*lam;

fun sub t x (Var y) = if y=x then t
                             else Var y
  | sub t x (App(u,v)) = App(sub t x u,
                             sub t x v)
  | sub t x (Lam(y,u)) = Lam(y, sub t x u);
```

Values of type `lam` give a *many-one* representation of λ-terms modulo α-conversion.

As an implementation of *capture-free substitution* of `t` for `Var x` in `u`, `sub t x u` is only partially correct.

As an implementation of *capture-free substitution* of `t` for `Var x` in `u`, `sub 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 FML language computes freshness information and ensures that it's version of `sub` respects the above convention. Here is the FML version of the code on Slide 5...

Abstract syntax trees modulo α-conversion in FML

```
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`, modulo contextual equivalence, are in bijection with (open) λ-terms modulo α-conversion; and

`sub:lam -> atm -> lam -> lam`

implements capture-free substitution.

Example of an untypable function:

list of binding variables in a λ-term

```
fun  bv (Var a) = nil
  |  bv (App(t,u)) = append(bv t)(bv u)
  |  bv (Lam[a]t) = a::(bv t);
```

Does not type check (good!).

Compare with typable FML function `cb:lam -> int` to count binders in a λ-term:

```
fun  cb (Var a) = 0
  |  cb (App(t,u)) = (cb t)+(cb u)
  |  cb (Lam[a]t) = 1+(cb t);
```

[Can prove `cb` is total by structural induction for `lam`, using theory from (Gabbay & Pitts, 1999).]

The FML code on Slides 6 and 7 contains examples of the following novelties compared with ML:

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

To explain these, need to say something about the FML type system. . .

The FML type system

Uses judgements of the form

$\Gamma \vdash E : \tau$	‘given Γ, expression E has type τ’
$\boxed{\Gamma \vdash E \mathbin{\texttt{#}} a}$	‘given Γ, atom identifier a is fresh for E’

(in the second case, expect Γ to contain assumption $a : \texttt{atm}$).

Hypotheses Γ may contain

$x : \tau$	typing assumptions
$x \mathbin{\texttt{#}} a$	freshness assumptions

about value identifiers.

The denotational semantics in FM-sets of the freshness judgement uses the notion of *finite support* which is part of the theory of FM-sets: denotationally, an atom is fresh for E if it is not in the support of the denotation of E.

Atom abstraction

Typing

$$\frac{\Gamma \vdash a : \mathtt{atm} \qquad \Gamma \vdash E : \tau}{\Gamma \vdash \mathtt{[}a\mathtt{]}\,E : \mathtt{[atm]}\,\tau}$$

Freshness

$$\Gamma, a : \mathtt{atm} \vdash \mathtt{[}a\mathtt{]}\,E \;\mathtt{\#}\; a \qquad\qquad \frac{\Gamma \vdash E \;\mathtt{\#}\; a'}{\Gamma \vdash \mathtt{[}a\mathtt{]}\,E \;\mathtt{\#}\; a'}$$

N.B. $\mathtt{[}a\mathtt{]}\,E$ *is not a binding form*!

In the first rule, $\Gamma \vdash a : \mathtt{atm}$ will hold if $(a : \mathtt{atm}) \in \Gamma$.

Matching abstraction patterns

is done via concretion at a locally fresh atom

For example

```
fun  ···
 |   sub t x (Lam[a]u) = Lam[a](sub t x u)
```

elaborates (in the static semantics of FML) to

```
fun  ···
 |   sub t x (Lam y) =
           new a in  Lam [a](sub t x ( y @ a ))
```

To be explained:

- *concretion* of an abstraction;
- declaration of a *locally fresh atoms*.

Concretion at an atom

Typing

$$\frac{\Gamma \vdash E : \texttt{[atm]}\tau \quad \Gamma \vdash E \;\texttt{\#}\; a}{\Gamma \vdash E \;\texttt{@}\; a : \tau}$$

Freshness

$$\frac{\Gamma \vdash E \;\texttt{\#}\; a' \quad \Gamma \vdash a \;\texttt{\#}\; a'}{\Gamma \vdash E \;\texttt{@}\; a \;\texttt{\#}\; a'}$$

N.B. because of the first rule, *typing assertions can depend upon freshness assertions*.

Local fresh atoms

Typing [requires *hypothetical* freshness]

$$\frac{\boxed{\Gamma \mathbin{\texttt{\#}} a} \vdash E : \tau \qquad \boxed{\Gamma \mathbin{\texttt{\#}} a} \vdash E \mathbin{\texttt{\#}} a}{\Gamma \vdash \texttt{new}\, a\, \texttt{in}\, E : \tau} \quad a \notin \mathrm{dom}(\Gamma)$$

where, for example,

if $\Gamma = [x : \tau, y : \texttt{atm}, x \mathbin{\texttt{\#}} y]$

then $\Gamma \mathbin{\texttt{\#}} a = [x : \tau, y : \texttt{atm}, x \mathbin{\texttt{\#}} y, a : \texttt{atm}, x \mathbin{\texttt{\#}} a, y \mathbin{\texttt{\#}} a]$.

Freshness . . . [omitted] . . .

N.B. $\texttt{new}$ is a binder: free occurrences of a in E are bound in $\texttt{new}\, a\, \texttt{in}\, E$.

An example of the use of the rule on Slide 13:

$$\frac{a : \mathtt{atm}, b : \mathtt{atm}, a \mathrel{\texttt{\#}} b \vdash (\mathtt{Var}\, a) : \mathtt{lam} \qquad a : \mathtt{atm}, b : \mathtt{atm}, a \mathrel{\texttt{\#}} b \vdash (\mathtt{Var}\, a) \mathrel{\texttt{\#}} b}{a : \mathtt{atm} \vdash \mathtt{new}\, b\, \mathtt{in}\, (\mathtt{Var}\, a) : \mathtt{lam}}$$

and both hypotheses are provable.

FML's notion of name restriction, $\texttt{new}\ a\ \texttt{in}\ (-)$, builds in an important simplification that always seems to be present in uses of 'fresh names' in informal syntax-manipulating algorithms, viz. *expressions in the scope of a fresh name only use it in ways which are insensitive to renaming*

For example

$$\vdash \texttt{new}\ a\ \texttt{in}\ \texttt{Lam}\ \texttt{[}a\texttt{]}(\texttt{Var}\ a) : \texttt{lam}$$

because $\vdash\ \texttt{[}a\texttt{]}(\texttt{Var}\ a)\ \texttt{\#}\ a$, but

$$\nvdash \texttt{new}\ a\ \texttt{in}\ (\texttt{Var}\ a) : \texttt{lam}$$

because $\nvdash\ \texttt{Var}\ a\ \texttt{\#}\ a$.

Abstraction-concretion

$$\frac{E \Downarrow V}{(\texttt{[}a\texttt{]}\,E)\ \texttt{@}\ b \Downarrow (a\,b)\cdot V}$$

where $(a\,b)\cdot V$ is the result of swapping all occurrences of a and b in V.

Local fresh atoms

$$\frac{[b/a]E \Downarrow V}{\texttt{new}\ a\ \texttt{in}\ E \Downarrow V} \quad b \notin \mathrm{fv}(\texttt{new}\ a\ \texttt{in}\ E)$$

where $[b/a]E$ is the result of capture-avoiding substitution of b for all free occurrences of a in E.

N.B. in the rule for evaluating concretions of abstractions, it would not be correct to use $[b/a]V$ instead of $(a\,b) \cdot V$. This is because of the fact that atom abstraction, $\mathtt{[}b\mathtt{]}(-)$, is not a variable binding operation and hence we are allowed to replace '$-$' by expressions with free occurrences of b during a substitution $[b/a]$. For example

$$(\mathtt{[}a\mathtt{]}(\mathtt{[}b\mathtt{]}a))\,\mathtt{@}\,b \Downarrow (a\,b) \cdot (\mathtt{[}b\mathtt{]}a) = \mathtt{[}a\mathtt{]}b$$

whereas $[b/a](\mathtt{[}b\mathtt{]}a) = \mathtt{[}b\mathtt{]}b$, which is denotationally distinct from $\mathtt{[}a\mathtt{]}b$ and certainly not equal to the denotation of $(\mathtt{[}a\mathtt{]}(\mathtt{[}b\mathtt{]}a))\,\mathtt{@}\,b$.

N.B. the rule for $\mathtt{new}$ is simpler than that for dynamically created names (of references or exceptions) in ML: the name b we use to instantiate the bound variable a in $\mathtt{new}\ a\ \mathtt{in}\ E$ can be chosen *locally fresh*, i.e. just fresh with respect to E—we do not have to pass around a context of currently used names. This is because the rules on Slide 15 are only applied to well-typed terms: it is the FML typing rule for $\mathtt{new}$ that ensures that this simple 'local freshness' is semantically correct (i.e. fits the denotational semantics in FM-sets).

Contextual equivalence ($=_{\mathrm{ctx}}$) of abstractions

FML contextual equivalence: observe results of evaluation in contexts of concrete type $\mathtt{int}$, $\mathtt{char}$,. . .

(N.B. like τ `->` τ, atom-abstraction types $\mathtt{[atm]}\tau$ are *not* concrete.)

Key lemma:

$$\Gamma, a : \mathtt{atm}, a' : \mathtt{atm} \vdash \mathtt{[}a\mathtt{]}V =_{\mathrm{ctx}} \mathtt{[}a'\mathtt{]}V' : \mathtt{[atm]}\tau$$

if and only if

$$(\Gamma, a : \mathtt{atm}, a' : \mathtt{atm}) \mathbin{\#} b \vdash (a\, b) \cdot V =_{\mathrm{ctx}} (a'\, b) \cdot V' : \tau.$$

Hence,

values (modulo $=_{\mathrm{ctx}}$) of FML data types like $\mathtt{lam}$ are in bijection with the corresponding abstract syntax trees *modulo* α*-conversion*.

Maybe the talk so far has given the impression that the design of FML is quite unproblemmatic. However, freshness is in general undecidable in the FM-sets model; and (so) there are difficulties with getting a sufficiently expressive, yet decidable freshness-inference system in the presence of recursive function declarations (which are an essential feature). Slide 17 gives the current (quite weak) rule we use for typing function values. Slide 18 illustrates the fact that it does not allow as much freshness information to be deduced as one would like; and Slide 19 gives some evidence that it is possible to work around this.

Function abstraction and application

Typing . . .[usual rules]. . .

Freshness

$$\frac{(y \mathbin{\texttt{\#}} a) \in \Gamma, \text{ for all } y \in \mathrm{fv}(E) \smallsetminus \{f, x\}}{\Gamma \vdash (\texttt{fun}\ \ f \;\texttt{=}\; \{x \;\texttt{=>}\; E\}) \mathbin{\texttt{\#}} a}$$

$$\frac{\Gamma \vdash E_1 \mathbin{\texttt{\#}} a \quad \Gamma \vdash E_1 \mathbin{\texttt{\#}} a}{\Gamma \vdash E_1\, E_2 \mathbin{\texttt{\#}} a}$$

N.B. freshness of an atom for the *denotation* of a function value is undecidable in general. The above rule gives a simple (too simple?) decidable approximation.

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*!—cannot prove `(rem a (fv t)) # a`.

[To do that, seem to need to use types like

`(atm list) -> (atm list # a)`

i.e. *types dependent upon atoms*.]

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
  | [a](b::bs) where b=a => rem2 [a]bs
  | [a](b::bs) where b # a => b::(rem2 [a]bs)};
```

`fv2` does type check: $\vdash$ `fv2 : lam -> atm list`
(and does code the intended function).

The code for `rem2` illustrates FML's treatment of *atom-equality testing*. It is non-trivial to give this in a form which allows good freshness-inference in the type system. We use *equality and inequality guards* in matches—briefly alluded to at the the end of (Gabbay & Pitts, 1999).

To do

- Establish the main theoretical properties of FML:
  type soundness, correctness of the representation of syntax mod α (cf. Slide 16), . . .

- Investigate type/freshness-inference algorithms.

- Implement FML.

- Do the novel features of FML provide a good idiom for syntax-manipulating algorithms *and proofs about them*?

  Is the current type/freshness inference system sufficiently expressive?

- Logical frameworks based on these ideas.
  (*Primitive recursion* does generalise to this setting.)