Lecture 6: functional programming

Semantics: what's it for?

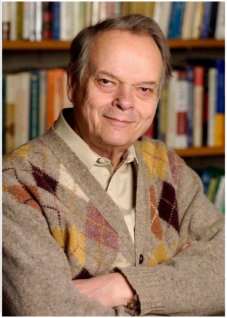
- Program verification.
- Implementation of existing programming languages.
- Design of new programming languages.

"Why is it so hard to design a good programming language? Naively, one might expect that a straightforward extension of the conventional notation of science and mathematics should provide a completely adequate programming language. But the history of language design has destroyed this illusion.

"The truth of the matter is that putting languages together is a very tricky business. When one attempts to combine language concepts, unexpected and counterintuitive interactions arise. At this point, even the most experienced designer's intuition must be buttressed by a rigorous definition of what the language means.

"Of course, this is what programming language semantics is all about."

John Reynolds, 1990



FreshML

It aimed to provide, within an ML-style functional programming language, higher-order structural recursion that automatically respects α-conversion of bound names, without anonymizing binding constructs.

FreshML

Design motivated by simple denotational model in **Nom**:

nominal sets inductively defined using
$(-) \times (-)$, $[\mathbb{A}](-)$, etc.
$+$
"α-structural" recursion principle

FreshML

Design motivated by simple denotational model in $\mathbf{Nom}$:

nominal sets inductively defined using
$(-) \times (-)$, $[\mathbb{A}](-)$, etc.
$+$
"α-structural" recursion principle

How to deal with its freshness side-conditions?

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 \in \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 \in X \times X \to_{\mathrm{fs}} X \\ f_3 \in \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \mathrel{\#} (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \mathrel{\#} f_3(a, x) \qquad \text{(FCB)}$$

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(a, \hat{f}\, e) \quad \text{if } a \mathrel{\#} (f_1, f_2, f_3) \end{cases}$

Can we avoid explicit reasoning about finite support, # and (FCB) when computing 'mod α'?

Want definition/computation to be separate from proving.

FreshML

Design motivated by simple denotational model in $\mathbf{Nom}$:

nominal sets inductively defined using
$(-) \times (-)$, $[\mathbb{A}](-)$, etc.
$+$
"α-structural" recursion principle

How to deal with freshness side-conditions?

Pure: type inference (Gabbay-P)
assertion-checking (Pottier)

Impure: dynamically allocated global names
(Shinwell-P)

$$
\begin{aligned}
\hat{f}\, a &= f_1\, a \\
\hat{f}(e_1\, e_2) &= f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\
\hat{f}(\lambda a.\, e) &= f_3(a, \hat{f}\, e) \qquad \text{if } a \mathrel{\#} (f_1, f_2, f_2)
\end{aligned}
$$

$$= \lambda a'.\, e' \qquad\qquad = f_3(a', \hat{f}\, e')$$

Q: how to get rid of this inconvenient proof obligation?

$$\begin{array}{rcl} \hat{f}\,a & = & f_1\,a \\ \hat{f}(e_1\,e_2) & = & f_2(\hat{f}\,e_1, \hat{f}\,e_2) \\ \hat{f}(\lambda a.\,e) & = & \nu a.\,f_3(a, \hat{f}\,e) \quad [\ a \,\#\, (f_1, f_2, f_2)\ ] \end{array}$$

$= \lambda a'.\,e'$ $\qquad$ $= \nu a'.\,f_3(a', \hat{f}\,e')$ **OK!**

Q: how to get rid of this inconvenient proof obligation?

A: use a local scoping construct $\nu a.\,(-)$ for names

$$
\begin{aligned}
\hat{f} &= f_1\, a \\
\hat{f}(e_1\, e_2) &= f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\
\hat{f}(\lambda a.\, e) &= \nu a.\, f_3(a, \hat{f}\, e) \quad [\, a \;\#\; (f_1, f_2, f_2)\,]
\end{aligned}
$$

$= \lambda a'.\, e'$ $\quad$ $= \nu a'.\, f_3(a', \hat{f}\, e')$ **OK!**

Q: how to get rid of this inconvenient proof obligation?

A: use a local scoping construct $\nu a.\, (-)$ for names

which one?!

Dynamic allocation

- Stateful: $\nu a.\,t$ means "add a fresh name a' to the current state and return $t[a'/a]$".
- Used in Shinwell's Fresh OCaml = OCaml +
  - name types and name-abstraction type former
  - name-abstraction patterns
    —matching involves dynamic allocation of fresh names

  [`www.fresh-ocaml.org`].

Sample Fresh OCaml code

```
(* syntax *)
type t;;
type var = t name;;
type term = Var of var | Lam of «var»term | App of term*term;;

 (* semantics *)
type sem = L of ((unit -> sem) -> sem) | N of neu
and  neu = V of var | A of neu*sem;;

 (* reify : sem -> term *)
let rec reify d =
  match d with L f -> let x = fresh in Lam(«x»(reify(f(function () -> N(V x)))))
             | N n -> reifyn n
and reifyn n =
  match n with V x -> Var x
             | A(n',d') -> App(reifyn n', reify d');;

(* evals : (var * (unit -> sem))list -> term -> sem *)
let rec evals env t  =
  match t with Var x -> (match env with [] -> N(V x)
                                   | (x',v)::env -> if x=x' then v() else evals env (Var x))
             | Lam(«x»t) -> L(function v -> evals ((x,v)::env) t)
             | App(t1,t2) -> (match evals env t1 with L f -> f(function () -> evals env t2)
                                                  | N n -> N(A(n,evals env t2)));;

(* eval : term -> sem *)
let rec eval t = evals [] t;;

(* norm : lam -> lam *)
let norm t = reify(eval t);;
```

Dynamic allocation

- Stateful: $\nu a.t$ means "add a fresh name a' to the current state and return $t[a'/a]$".
- Used in Shinwell's Fresh OCaml = OCaml +
  - name types and name-abstraction type former
  - name-abstraction patterns
    —matching involves dynamic allocation of fresh names

[www.fresh-ocaml.org].

Dynamic allocation

- Stateful: $\nu a.\,t$ means "add a fresh name a' to the current state and return $t[a'/a]$".

Statefulness disrupts familiar mathematical properties of pure datatypes. So we will try to reject it in favour of...

Odersky's $\nu a.(-)$

[M. Odersky, *A Functional Theory of Local Names*, POPL'94]

- Unfamiliar—apparently not used in practice (so far).
- Pure equational calculus, in which local scopes 'intrude' rather than extrude (as per dynamic allocation):

$$
\begin{array}{rcll}
\nu a.(\lambda x.t) & \approx & \lambda x.(\nu a.t) & [a \neq x] \\
\nu a.(t\,,t') & \approx & (\nu a.t\,,\nu a.t') &
\end{array}
$$

- New: a straightforward semantics using nominal sets equipped with a 'name-restriction operation'...

Name-restriction

A name-restriction operation on a nominal set X is a morphism $(-)\backslash(-) \in \mathbf{Nom}(\mathbb{A} \times X, X)$ satisfying

- $a \mathbin{\#} a \backslash x$
- $a \mathbin{\#} x \;\Rightarrow\; a \backslash x = x$
- $a \backslash (b \backslash x) = b \backslash (a \backslash x)$

Equivalently, a morphism $\rho : [\mathbb{A}]X \to X$ making

$$
\begin{tikzcd}
X \arrow[r, "\kappa"] \arrow[dr, "id_X"'] & {[\mathbb{A}]X} \arrow[d, "\rho"] \\
& X
\end{tikzcd}
\qquad
\begin{tikzcd}
{[\mathbb{A}][\mathbb{A}]X} \arrow[rr, "\delta"] \arrow[d, "{[\mathbb{A}]\rho}"'] & & {[\mathbb{A}][\mathbb{A}]X} \arrow[d, "{[\mathbb{A}]\rho}"] \\
{[\mathbb{A}]X} \arrow[r, "\rho"'] & X & {[\mathbb{A}]X} \arrow[l, "\rho"]
\end{tikzcd}
$$

commute, where $\kappa\, x = \langle a \rangle x$ for some (or indeed any) $a \mathbin{\#} x$; and where $\delta(\langle a \rangle \langle a' \rangle x) = \langle a' \rangle \langle a \rangle x$.

Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \mathrel{\#} (f_1, f_2, f_3) \ \Rightarrow\ (\forall x)\ a \mathrel{\#} f_3(a, x) \qquad \text{(FCB)}$$

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ .t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(a, \hat{f}\, e) \quad \text{if } a \mathrel{\#} (f_1, f_2, f_3) \end{cases}$

If X has a name restriction operation $(-)\backslash(-)$, we can trivially satisfy (FCB) by using $a \backslash f_3(a, x)$ in place of $f_3(a, x)$.

Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 \in \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 \in X \times X \to_{\mathrm{fs}} X \\ f_3 \in \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$

and a restriction operation $(-)\backslash(-)$ on X,

$$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X \text{ .t. } \begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = a \backslash f_3(a, \hat{f}\, e) \end{cases}$$

Is requiring X to carry a name-restriction operation much of a hindrance for applications?

Not much. . .

Examples of name-restriction

- For $\mathbb{N}$: $$a \backslash n \triangleq n$$

Examples of name-restriction

- For $\mathbb{N}$: $$a \backslash n \triangleq n$$
- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\mathbf{anon}\}$:

$$\begin{aligned} a \backslash a &\triangleq \mathbf{anon} \\ a \backslash a' &\triangleq a' \quad \text{if } a' \neq a \\ a \backslash \mathbf{anon} &\triangleq \mathbf{anon} \end{aligned}$$

Examples of name-restriction

- For $\mathbb{N}$:
$$a \backslash n \triangleq n$$

- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\mathbf{anon}\}$:
$$a \backslash t \triangleq t[\mathbf{anon}/a]$$

- For $\Lambda' \triangleq \{t ::= \mathtt{V}\, a \mid \mathtt{A}(t\,,t) \mid \mathtt{L}(a\,.\,t) \mid \mathbf{anon}\}/{=_\alpha}$:
$$a \backslash [t]_\alpha \triangleq [t[\mathbf{anon}/a]]_\alpha$$

Examples of name-restriction

- For $\mathbb{N}$:
$$a \backslash n \triangleq n$$

- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\mathbf{anon}\}$:
$$a \backslash t \triangleq t[\mathbf{anon}/a]$$

- For $\Lambda' \triangleq \{t ::= \mathtt{V}\, a \mid \mathtt{A}(t\,,t) \mid \mathtt{L}(a\,.\,t) \mid \mathbf{anon}\}/{=_\alpha}$:
$$a \backslash [t]_\alpha \triangleq [t[\mathbf{anon}/a]]_\alpha$$

- Nominal sets with name-restriction are closed under products, coproducts, name-abstraction and exponentiation by a nominal set.

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, JFP 21(2011)235–286]

is standard simply-typed λ-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$, with terms for
  - names, $a : \mathtt{Name}$ $(a \in \mathbb{A})$
  - equality test, $\_ = \_ : \mathtt{Name} \to \mathtt{Name} \to \mathtt{Bool}$
  - name-swapping, $\dfrac{t : T}{(a \wr a')t : T}$
  - locally scoped names $\dfrac{t : T}{\nu a.\, t : T}$ (binds a)

  with Odersky-style computation rules, e.g.

$$\nu a.\, \lambda x.\, t = \lambda x.\, \nu a.\, t$$

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, JFP 21(2011)235–286]

is standard simply-typed λ-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$
- name-abstraction types, $\mathtt{Name}\,.\,T$, with terms for
  - name-abstraction, $\dfrac{t : T}{\alpha a.\, t : \mathtt{Name}\,.\,T}$ (binds a)
  - unbinding, $\dfrac{t : \mathtt{Name}\,.\,T \qquad t' : T'}{\mathtt{let}\; a\,.\,x = t\;\mathtt{in}\; t' : T'}$ (binds a & x in t')

    with computation rule that uses local scoping

$$\boxed{\mathtt{let}\; a\,.\,x = \alpha a.\, t\;\mathtt{in}\; t' = \nu a.\,(t'[t/x])}$$

$\lambda\alpha\nu$-Calculus

Denotational semantics. $\lambda\alpha\nu$-calculus has a straightforward interpretation in **Nom** that is sound for the computation rules—types denote nominal sets equipped with a name-restriction operation:

$$
\begin{array}{rcll}
\llbracket \texttt{Bool} \rrbracket & = & \{\mathbf{true}, \mathbf{false}\} & \\
\llbracket \texttt{Name} \rrbracket & = & \mathbb{A} \uplus \{\mathbf{anon}\} & \\
\llbracket T \times T' \rrbracket & = & \llbracket T \rrbracket \times \llbracket T' \rrbracket & \llbracket \nu a.\, a \rrbracket \\
\llbracket T \to T' \rrbracket & = & \llbracket T \rrbracket \to_{\mathbf{fs}} \llbracket T \rrbracket & \\
\llbracket \texttt{Name}\,.\, T \rrbracket & = & [\mathbb{A}]\llbracket T \rrbracket &
\end{array}
$$

See [NSB, Section 9.4].

$\lambda\alpha\nu$-calculus as a FP language

To do: revisit FreshML using Odersky-style local names rather than dynamic allocation (cf. [Lösch+AMP, POPL 2013]

‘Nominal Agda’ (???)

Can the $\lambda\alpha\nu$-calculus be extended from simple to dependent types?

```
names Var : Set

data Term : Set where                     --(possibly open) λ-terms mod α
  V : Var -> Term                         --variable
  A : (Term × Term)-> Term                --application term
  L : (Var . Term) -> Term                --λ-abstraction

_/_ : Term -> Var -> Term -> Term         --capture-avoiding substitution
(t / x)(V x′) = if x = x′ then t else V x′
(t / x)(A(t′ , t″)) = A((t / x )t′ , (t / x )t″)
(t / x)(L(x′ . t′)) = L(x′ . (t / x)t′)

data _==_ (t : Term) : Term -> Set where  --intensional equality
  Refl : t == t
```

'Nominal Agda' (???)

Can the $\lambda\alpha\nu$-calculus be extended from simple to dependent types?

```
names Var : Set

data Term : Set where                    --(possibly open) λ-terms mod α
  V : Var -> Term                        --variable
  A : (Term × Term)-> Term               --application term
  L : (Var . Term) -> Term               --λ-abstraction

_/_ : Term -> Var -> Term -> Term        --capture-avoiding substitution
(t / x)(V x′) = if x = x′ then t else V x′
(t / x)(A(t′ , t″)) = A((t / x )t′ , (t / x )t″)
(t / x)(L(x′ . t′)) = L(x′ . (t / x)t′)

data _==_ (t : Term) : Term -> Set where --intensional equality
  Refl : t == t                          --is term equality mod α

eg : (x x′ : Var) ->
  ((V x) / x′)(L(x . V x′)) == L(x′ . V x)   --(λx.x′)[x/x′] = λx′.x
eg x x′ = {! !}
```