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 \textit{automatically respects} $\alpha$-conversion of bound names, without anonymizing binding constructs.
Design motivated by simple denotational model in \textbf{Nom}: nomial sets inductively defined using 
\((-) \times (-), [A](-), \text{ etc.} \)

+ “\(\alpha\)-structural” recursion principle
Design motivated by simple denotational model in $\textbf{Nom}$:

nominal sets inductively defined using 

$(-) \times (-)$, $[A](-)$, etc.

+ 

“$\alpha$-structural” recursion principle

How to deal with its freshness side-conditions?
α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \text{Nom}$ and

\[
\begin{aligned}
&f_1 \in A \rightarrow_{fs} X \\
&f_2 \in X \times X \rightarrow_{fs} X \\
&f_3 \in A \times X \rightarrow_{fs} X
\end{aligned}
\]

s.t.

\[(\forall a) \ a \neq (f_1, f_2, f_3) \implies (\forall x) \ a \neq f_3(a, x) \quad \text{(FCB)}\]

\[
\exists! \hat{f} \in \Lambda \rightarrow_{fs} X
\]

s.t.

\[
\begin{aligned}
\hat{f} a &= f_1 a \\
\hat{f}(e_1 e_2) &= f_2(\hat{f} e_1, \hat{f} e_2) \\
\hat{f}(\lambda a.e) &= f_3(a, \hat{f} e) \quad \text{if} \ a \neq (f_1, f_2, f_3)
\end{aligned}
\]

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 \textbf{Nom}:

nominal sets inductively defined using \((-) \times (-), [A](-), \) etc.

+ “\(\alpha\)-structural” recursion principle

How to deal with freshness side-conditions?

\textbf{Pure}: type inference (Gabbay-P) assertion-checking (Pottier)

\textbf{Impure}: dynamically allocated global names (Shinwell-P)
\[ \hat{f} = f_1 a \]
\[ \hat{f}(e_1 e_2) = f_2(\hat{f} e_1, \hat{f} e_2) \]
\[ \hat{f}(\lambda a. e) = f_3(a, \hat{f} e) \quad \text{if } a \neq (f_1, f_2, f_2) \]

Q: how to get rid of this inconvenient proof obligation?
\[
\begin{align*}
\hat{f} &= f_1 a \\
\hat{f}(e_1 e_2) &= f_2(\hat{f} e_1, \hat{f} e_2) \\
\hat{f}(\lambda a. e) &= \nu a. f_3(a, \hat{f} e) \quad [ \ a \not\in (f_1, f_2, f_2) \ ] \\
\end{align*}
\]

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

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

A: use a local scoping construct \( \nu a. (\_) \) for names
\[ \hat{f} = f_1 a \]
\[ \hat{f}(e_1 e_2) = f_2(\hat{f} e_1, \hat{f} e_2) \]
\[ \hat{f}(\lambda a. e) = \nu a. f_3(a, \hat{f} e) \quad [a \not\in \{f_1, f_2, f_2\}] \]
\[ = \lambda a'. e' \]
\[ = \nu a'. f_3(a', \hat{f} e') \quad \text{OK!} \]

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: \(va.t\) 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 *)

```ocaml
type t;;
type var = t name;;
type term = Var of var | Lam of `var`term | App of term*term;;
```

(* semantics *)

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

(* reify : sem -> term *)

```ocaml
let rec reify d =
  match d with
  | L f -> let x = fresh in Lam(`x`)(reify(f(function () -> N(V x))))
  | N n -> reify n
and reify n =
  match n with
    | V x -> Var x
    | A(n',d') -> App(reify n', reify d');;
```

(* evals : (var * (unit -> sem))list -> term -> sem *)

```ocaml
let rec evals env t =
  match t with
    | Var x -> (match env with [] -> N(V 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 *)

```ocaml
let rec eval t = evals [] t;;
```

(* norm : lam -> lam *)

```ocaml
let norm t = reify(eval t);;
```
Dynamic allocation

- Stateful: $va.t$ 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):
  
  $\nu a. (\lambda x. t) \approx \lambda x. (\nu a. t) \quad [a \neq x]$
  
  $\nu a. (t, t') \approx (\nu a. t, \nu a. t')$

- New: a straightforward semantics using nominal sets equipped with a ‘name-restriction operation’...
A **name-restriction** operation on a nominal set $X$ is a morphism $\langle (-) \setminus (-) \rangle \in \text{Nom}(\mathcal{A} \times X, X)$ satisfying

1. $a \# a \setminus x$
2. $a \# x \Rightarrow a \setminus x = x$
3. $a \setminus (b \setminus x) = b \setminus (a \setminus x)$

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

\[
\begin{array}{ccc}
X & \xrightarrow{\kappa} & [\mathcal{A}]X \\
\downarrow{id_X} & & \rho \\
X & & [\mathcal{A}]X \\
\end{array}
\]

\[
\begin{array}{ccc}
[\mathcal{A}]X & \xrightarrow{\delta} & [\mathcal{A}]X \\
\downarrow{[\mathcal{A}]\rho} & & \downarrow{[\mathcal{A}]\rho} \\
[\mathcal{A}][\mathcal{A}]X & & [\mathcal{A}]X \\
\end{array}
\]

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

Lecture 6 11/16
Given any \( X \in \text{Nom} \) and \( \begin{cases} f_1 \in A \rightarrow_{fs} X \\ f_2 \in X \times X \rightarrow_{fs} X \\ f_3 \in A \times X \rightarrow_{fs} X \end{cases} \) s.t.

\[
(\forall a) \ a \# (f_1, f_2, f_3) \Rightarrow (\forall x) \ a \# f_3(a, x) \quad \text{(FCB)}
\]

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

If \( X \) has a name restriction operation \( (\cdot) \setminus (\cdot) \), we can trivially satisfy (FCB) by using \( a \setminus f_3(a, x) \) in place of \( f_3(a, x) \).
Given any $X \in \text{Nom}$ and
\[
\begin{cases}
  f_1 \in A \rightarrow_{fs} X \\
  f_2 \in X \times X \rightarrow_{fs} X \\
  f_3 \in A \times X \rightarrow_{fs} X
\end{cases}
\]
and a restriction operation $(\_)(\_)$ on $X$,
\[
\exists! \hat{f} \in \Lambda \rightarrow_{fs} X \quad \text{t.} \quad \begin{cases}
  \hat{f} a = f_1 a \\
  \hat{f} (e_1 e_2) = f_2 (\hat{f} e_1, \hat{f} e_2) \\
  \hat{f} (\lambda a.e) = a \backslash f_3 a, \hat{f} e)
\end{cases}
\]

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

Not much...
Examples of name-restriction

- For $\mathbb{N}$:

$$a \setminus n \triangleq n$$
Examples of name-restriction

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

- For $A' \triangleq A \cup \{\text{anon}\}$:
  \[ a \backslash a \triangleq \text{anon} \]
  \[ a \backslash a' \triangleq a' \text{ if } a' \neq a \]
  \[ a \backslash \text{anon} \triangleq \text{anon} \]
Examples of name-restriction

▶ For $\mathbb{N}$:

$$a \backslash n \triangleq n$$

▶ For $A' \triangleq A \cup \{\text{anon}\}$:

$$a \backslash t \triangleq t[\text{anon}/a]$$

▶ For $\Lambda' \triangleq \{t ::= \forall a \mid A(t, t) \mid L(a \cdot t) \mid \text{anon}\}/\alpha$:

$$a \backslash [t]_\alpha \triangleq [t[\text{anon}/a]]_\alpha$$
Examples of name-restriction

- For \( \mathbb{N} \):
  \[
  a \setminus n \triangleq n
  \]

- For \( A' \triangleq A \uplus \{\text{anon}\} \):
  \[
  a \setminus t \triangleq t[\text{anon} / a]
  \]

- For \( \Lambda' \triangleq \{ t ::= \forall a \mid A(t, t) \mid L(a \cdot t) \mid \text{anon} \}/=_{\alpha} \):
  \[
  a \setminus [t]_{\alpha} \triangleq [t[\text{anon} / a]]_{\alpha}
  \]

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


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

- type of names, Name, with terms for
  - names, \( a : \text{Name} \) \((a \in \mathcal{A})\)
  - equality test, \(_ = _ :\text{Name} \to \text{Name} \to \text{Bool}\)
  - name-swapping, \(t : T\)
    \[
    (a \land a')t : T
    \]
  - locally scoped names
    \(\nu a. t : T\)
    (binds \(a\))

with Odersky-style computation rules, e.g.

\[
\nu a. \lambda x. t = \lambda x. \nu a. t
\]
is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

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

  with computation rule that uses local scoping

$$\text{let } a . x = \alpha a . t \text{ in } t' = \nu a . (t'[t/x])$$
\textbf{\(\lambda\alpha\nu\)-Calculus}

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

\[
\begin{align*}
[\text{Bool}] & = \{\text{true, false}\} \\
[\text{Name}] & = \mathcal{A} \sqcup \{\text{anon}\} \\
[T \times T'] & = [T] \times [T'] \\
[T \rightarrow T'] & = [T] \rightarrow_{fs} [T] \\
[\text{Name}.T] & = [\mathcal{A}][T]
\end{align*}
\]

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

```haskell
names Var : Set

data Term : Set where
  V : Var -> Term --variable
  A : (Term × Term) -> Term --application term
  L : (Var . Term) -> Term --\(\lambda\)-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
```

Lecture 6
‘Nominal Agda’ (???)

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

names Var : Set

data Term : Set where
  V : Var -> Term      --variable
  A : (Term \times Term) -> Term   --application term
  L : (Var . Term) -> Term      --$\lambda$-abstraction

_/_/ : Term -> Var -> Term -> Term
(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
  Refl : t == t        --is term equality mod $\alpha$

eg : (x x′ : Var) ->
  ((V x) / x′)(L(x . V x′)) == L(x′ . V x)        --($\lambda x . x′)[x/x′] = \lambda x′ . x

eg x x′ = {! !}