Not So Fresh ML

Nicolas Pouillard and François Pottier

{Nicolas.Pouillard, Francois.Pottier}@inria.fr

CANS Seminar, 2009
Towards safer and more expressive languages for meta-programming
Program representation should stay well-typed and well-scoped
Pursuing the work on FreshML

- Inspired from FreshML
- pure FreshML for its safety
- Caml for its expressiveness
A taste of FreshML/Caml
Data type for explicitly typed lambda calculus

data Term
    = Var Atom
    | App Term Term
    | Lam < Atom \times neutral Ty \times inner Term >
    | Let < Atom \times outer Term \times inner Term >
Capture avoiding substitution

\[
\text{subst :: (Atom, Term)} \to \text{Term} \to \text{Term} \\
\text{subst (a, v) = go} \\
\text{where} \\
\text{go (Var b)} = \text{if } a \equiv b \text{ then } v \text{ else Var b} \\
\text{go (App t u)} = \text{App (go t) (go u)} \\
\text{go (Lam} < b, ty, t > \text{)} = \text{Lam} < b, ty, \text{go t} > \\
\text{go (Let} < b, t, u > \text{)} = \text{Let} < b, \text{go t, go u} >
\]
Computing the size of a term

\[
\begin{align*}
\text{size} &:: \text{Term} \rightarrow \text{Int} \\
\text{size} (\text{Var } _) & = 1 \\
\text{size} (\text{App } t \; u) & = 1 + \text{size } t + \text{size } u \\
\text{size} (\text{Lam} < _, _, t >) & = 3 + \text{size } t \\
\text{size} (\text{Let} < _, t, u >) & = 3 + \text{size } t + \text{size } u
\end{align*}
\]
FreshML considered
FreshML considered too fresh!
More efficient programs

Freshening is useless while:

- Computing the size of a term
More efficient programs

Freshening is useless while:

- Computing the size of a term
- Computing free variables
More efficient programs

Freshening is useless while:

- Computing the size of a term
- Computing free variables
- Typing some languages
More efficient programs

Freshening is useless while:

- Computing the size of a term
- Computing free variables
- Typing some languages
- Counting occurrences of some variable
More efficient programs

Freshening is useless while:

- Computing the size of a term
- Computing free variables
- Typing some languages
- Counting occurrences of some variable
- Substituting closed terms for variables
More efficient programs

Freshening is useless while:

- Computing the size of a term
- Computing free variables
- Typing some languages
- Counting occurrences of some variable
- Substituting closed terms for variables
- Deciding $\alpha$-equivalence
More efficient programs

<table>
<thead>
<tr>
<th>Freshening is useless while:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing the size of a term</td>
</tr>
<tr>
<td>Computing free variables</td>
</tr>
<tr>
<td>Typing some languages</td>
</tr>
<tr>
<td>Counting occurrences of some variable</td>
</tr>
<tr>
<td>Substituting closed terms for variables</td>
</tr>
<tr>
<td>Deciding $\alpha$-equivalence</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
To freshen or not to freshen?

- While \texttt{FRESHML} implicitly freshen
To freshen or not to freshen?

- While FRESHML implicitly freshen
- This system allows both non-freshening and freshening openings
To freshen or not to freshen?

- While FreshML implicitly freshen
- This system allows both non-freshening and freshening openings
- However we will only the non-freshening part
World-index types for atoms
World-index types for atoms

let \( x = x \) in \( x \)
Let’s classify atoms by a world they live in

The type of atoms is now indexed by a world

type Atom α
Let’s classify atoms by a world they live in

The type of atoms is now indexed by a world

type Atom α

Equality is homogeneous and prevents mixing worlds

(≡)_{Atom} :: ∀ α. Atom α → Atom α → Bool
Data type for explicitly typed lambda calculus

data Term
    = Var Atom
    | App Term Term
    | Lam < Atom × neutral Ty × inner Term >
    | Let < Atom × outer Term × inner Term >
Data type for explicitly typed lambda calculus

```haskell
data Term outer
  = Var (Atom outer)
  | App (Term outer) (Term outer)
  | Exists inner. Lam (Atom inner) Ty (Term inner)
  | Exists inner. Let (Atom inner) (Term outer) (Term inner)
```
Data type for explicitly typed lambda calculus

\[
\text{data } \text{Term } \alpha \\
\qquad = \ \text{Var} \ (\text{Atom } \alpha) \\
\mid \ \text{App} \ (\text{Term } \alpha) \ (\text{Term } \alpha) \\
\mid \ \exists \beta. \ \text{Lam} \ (\text{Atom } \beta) \ \text{Ty} \ (\text{Term } \beta) \\
\mid \ \exists \beta. \ \text{Let} \ (\text{Atom } \beta) \ (\text{Term } \alpha) \ (\text{Term } \beta)
\]
Worlds are closely related to each other

The type of (oriented) links between worlds

type $\beta \rhd \alpha$
Worlds are closely related to each other

The type of (oriented) links between worlds

\[ \text{type } \beta \triangleright \alpha \]

Links holds the set of atoms as a frontier

\[ \alpha \triangleright (\notin s) \beta \]
Worlds are closely related to each other

The type of (oriented) links between worlds

\[
\text{type } \beta \triangleright \alpha
\]

Links holds the set of atoms as a frontier

\[
\alpha (\notin S) \triangleright \beta
\]

Links are supposed to be invisible/inferred!
Data type for explicitly typed lambda calculus

\[
\text{data Term } \alpha \\
\quad = \ \text{Var} \ (\text{Atom } \alpha) \\
\quad | \ \text{App} \ (\text{Term } \alpha) \ (\text{Term } \alpha) \\
\quad | \ \exists \ \beta. \ \text{Lam} \ (\text{Atom } \beta) \ \text{Ty} \ (\text{Term } \beta) \\
\quad | \ \exists \ \beta. \ \text{Let} \ (\text{Atom } \beta) \ (\text{Term } \alpha) \ (\text{Term } \beta)
\]
Data type for explicitly typed lambda calculus

```
data Term α  
  = Var (Atom α)  
  | App (Term α) (Term α)  
  | ∃ β. Lam (β ⊢ α) (Atom β) Ty (Term β)  
  | ∃ β. Let (β ⊢ α) (Atom β) (Term α) (Term β)
```
Identity link

\[ \text{id}_{\text{Link}} :: \forall \alpha. \alpha \triangleright \alpha \]
Identity link

\[ id_{\text{Link}} :: \forall \alpha. \alpha \triangleright \alpha \]

Link composition

\[ (\circ)_{\text{Link}} :: \forall \alpha \beta \gamma. (\beta \triangleright \gamma) \rightarrow (\alpha \triangleright \beta) \rightarrow (\alpha \triangleright \gamma) \]
Link operations

Identity link

\[ \text{id}_{\text{Link}} :: \forall \alpha. \alpha \triangleright \alpha \]

Link composition

\[ (\circ)_{\text{Link}} :: \forall \alpha \beta \gamma. (\beta \triangleright \gamma) \rightarrow (\alpha \triangleright \beta) \rightarrow (\alpha \triangleright \gamma) \]

Atomic link

\[ \text{atomic}_{\text{Link}} :: \forall \alpha. \text{Atom } \alpha \rightarrow (\alpha \triangleright \alpha) \]
Casts to walk through links

Atomic casts

\[ \text{cast}_{\text{Atom}} :: \forall \alpha \beta. (\beta \triangleright \alpha) \rightarrow (\text{Atom} \beta \rightarrow \text{Atom} \alpha) \]
Casts to walk through links

Atomic casts

$$\text{cast}_{\text{Atom}} :: \forall \alpha \beta. (\beta \triangleright \alpha) \rightarrow (\text{Atom } \beta \rightarrow \text{Atom } \alpha)$$

Generalized casts

$$\text{cast}_f :: \forall \alpha \beta f. (\beta \triangleright \alpha) \rightarrow (f \beta \rightarrow f \alpha)$$
Casts to walk through links

Atomic casts

\[ \text{cast}_{\text{Atom}} :: \forall \alpha \beta. (\beta \triangleright \alpha) \rightarrow (\text{Atom } \beta \rightarrow \text{Atom } \alpha) \]

Generalized casts

\[ \text{cast}_{f} :: \forall \alpha \beta f. (\beta \triangleright \alpha) \rightarrow (f \beta \rightarrow f \alpha) \]

Cast implies proof obligations or dynamic checks!
Atom abstraction as existential quantification
Hiding the real world but keeping a link

data \( \alpha < f > = \exists \beta . \text{Abs} (\beta \triangleright \alpha) (\text{Atom} \ \beta) (f \ \beta) \)
Data type for explicitly typed lambda calculus

\[
\text{data Term } \alpha \\
\quad = \ \text{Var} (\text{Atom } \alpha) \\
\quad \mid \ \text{App} (\text{Term } \alpha) (\text{Term } \alpha) \\
\quad \mid \ \exists \beta. \ \text{Lam} (\beta \triangleright \alpha) (\text{Atom } \beta) \ \text{Ty} (\text{Term } \beta) \\
\quad \mid \ \exists \beta. \ \text{Let} (\beta \triangleright \alpha) (\text{Atom } \beta) (\text{Term } \alpha) (\text{Term } \beta)
\]
Data type for explicitly typed lambda calculus

data Term α
    = Var (Atom α)
    | App (Term α) (Term α)
    | Lam α<λβ→ (Ty, Term β)>
    | Let α<λβ→ (Term α, Term β)>
Making an abstraction

\[ \text{Abs} : \forall \alpha \beta f. \ (\beta \triangleright \alpha) \rightarrow \text{Atom} \beta \rightarrow f \beta \rightarrow \alpha<f> \]
Making an abstraction

Abs :: ∀ α β. (β ⊢ α) → Atom β → f β → α<f>

Lam :: ∀ α β. (β ⊢ α) → Atom β → Term β → Term α
Making an abstraction

\[ \text{Abs} :: \forall \alpha \beta f. (\beta \triangleright \alpha) \rightarrow \text{Atom} \beta \rightarrow f \beta \rightarrow \alpha<f> \]

\[ \text{Lam} :: \forall \alpha \beta. (\beta \triangleright \alpha) \rightarrow \text{Atom} \beta \rightarrow \text{Term} \beta \rightarrow \text{Term} \alpha \]

\[ \text{mkLam} :: \forall \alpha. \text{Atom} \alpha \rightarrow \text{Term} \alpha \rightarrow \text{Term} \alpha \]

\[ \text{mkLam} \times t = \text{Lam} (\text{atomic} \times) \times t \]
Making an abstraction

Abs :: ∀ α β f. (β ⊃ α) → Atom β → f β → α<f>

Lam :: ∀ α β. (β ⊃ α) → Atom β → Term β → Term α

mkLam :: ∀ α. Atom α → Term α → Term α
mkLam x t = Lam (atomic x) x t

mkConst x y = mkLam x (mkLam y (Var x))
Opening an abstraction does not freshen it

```
let (Abs lnk x y) = t in u
```
Opening an abstraction does not freshen it

\[
\text{let (Abs \text{ In}k \ x \ y) = t \ in \ u}
\]

\[
\Gamma \vdash t : \alpha<f> \ where \ \alpha \in \Gamma
\]
Opening an abstraction does not freshen it

```
let (Abs Ink x y) = t in u
```

\[\Gamma \vdash t : \alpha<f> \text{ where } \alpha \in \Gamma\]

\[\Gamma, \beta, \text{Ink:}\beta\triangleright\alpha, x:\text{Atom} \quad \beta, y:f \quad \beta \vdash u : \tau \text{ where } \beta \not\equiv \tau\]
Safety Properties

- Well-typed programs do not get stuck
Safety Properties

- Well-typed programs do not get stuck
- $\alpha$-equivalence is preserved by casts
Safety Properties

- Well-typed programs do not get stuck
- $\alpha$-equivalence is preserved by casts
- Casts may dynamically fail or be proven successful
Safety Properties

- Well-typed programs do not get stuck
- $\alpha$-equivalence is preserved by casts
- Casts may dynamically fail or be proven successful
- $\alpha$-equivalence is defined structurally on types
Example commuting abstraction with pairs

\[
\text{commute} :: \forall \alpha. \alpha<\lambda\beta \rightarrow (\text{Term } \beta, \text{ Term } \beta) > \\
\rightarrow (\alpha<\text{Term}>, \alpha<\text{Term}>)
\]

\[
\text{commute } t = \\
\text{let } (\text{Abs Ink } x \ (y,z)) = t \\
\text{in } (\text{Abs Ink } x \ y, \text{ Abs Ink } x \ z)
\]
Name capture does not type-check

\[
\text{wrong} :: \forall \alpha. \alpha<\text{Term}> \rightarrow \text{Term} \alpha \rightarrow \alpha<\text{Term}>
\]

\[
\text{wrong } t u =
\]
\[
\begin{align*}
\text{let} & \ (\text{Abs Ink } x y) = t \\
\text{in} & \ \text{Abs Ink } x u
\end{align*}
\]
Computing the size of a term

\[ \text{size :: } \forall \alpha. \text{ Term } \alpha \rightarrow \text{ Int} \]

\[
\begin{align*}
\text{size (Var } \_) & = 1 \\
\text{size (App t u)} & = 1 + \text{size t} + \text{size u} \\
\text{size (Lam } \_ \_ \_ \text{ t)} & = 3 + \text{size t} \\
\text{size (Let } \_ \_ \text{ t u)} & = 3 + \text{size t} + \text{size u}
\end{align*}
\]
Computing the size of a term

\[
\text{size} :: \forall \alpha. \text{Term} \\alpha \to \text{Int} \\
\text{size} (\text{Var} _) = 1 \\
\text{size} (\text{App} t u) = 1 + \text{size} t + \text{size} u \\
\text{size} (\text{Lam} _ t) = 3 + \text{size} t \\
\text{size} (\text{Let} _ t u) = 3 + \text{size} t + \text{size} u
\]

Polymorphic recursion!
Computing free variables

remove :: Atom → [Atom] → [Atom]
remove [] = []
remove a (b:bs)
  | a ≡ b = remove a bs
  | otherwise = b : remove a bs

fv :: Term → [Atom]
fv (Var a) = [a]
fv (App t u) = fv t ++ fv u
fv (Lam<a,_,t>) = remove a (fv t)
fv (Let<a,t,u>) = fv t ++ remove a (fv u)
Computing free variables

remove ::
\[ \forall \beta. (\beta \triangleright \alpha) \rightarrow \text{Atom } \beta \rightarrow [\text{Atom } \beta] \rightarrow [\text{Atom } \alpha] \]
remove \_ \_ [] = []
remove lnk a (b:bs)
| a \equiv b = remove lnk a bs
| otherwise = cast lnk b : remove lnk a bs

fv :: \forall \alpha. \text{Term } \alpha \rightarrow [\text{Atom } \alpha]
fv (Var a) = [a]
fv (App t u) = fv t ++ fv u
fv (Lam lnk a _ t) = remove lnk a (fv t)
fv (Let lnk a t u) = fv t ++ remove lnk a (fv u)
Looking up an environment

```
data Env β = Empty
           | ∃ α. Snoc (β ⊢ α) (Env α) (Atom β) Ty

lookupEnv :: ∀ α. Atom α → Env α → Ty
lookupEnv a (Snoc Ink env b ty)
  | a ≡ b    = ty
  | otherwise = lookupEnv (cast Ink a) env
lookupEnv _ Empty = error "unbound value"
```
Typing a term

typing :: ∀ α. Env α → Term α → Ty
typing env (Var v)
  = lookupEnv v env
typing env (Lam lnk a ty t)
  = ty ‘TyArrow‘ typing (Snoc lnk env a ty) t
typing env (Let lnk a t u)
  = typing (Snoc lnk env a (typing env t)) u
typing env (App t u)
  = case typing env t of
      from ‘TyArrow‘ to | from ≡ typing env u → to
      _          → error "ill typed"
Challenges and future work

- Deeper formalization and proofs
Challenges and future work

- Deeper formalization and proofs
- $\alpha$-equivalence for inside-out abstractions
Challenges and future work

- Deeper formalization and proofs
- $\alpha$-equivalence for inside-out abstractions
- Better understanding of heterogeneous comparison
Challenges and future work

- Deeper formalization and proofs
- $\alpha$-equivalence for inside-out abstractions
- Better understanding of heterogeneous comparison
- Integrating complex binding structures
Challenges and future work

- Deeper formalization and proofs
- \(\alpha\)-equivalence for inside-out abstractions
- Better understanding of heterogeneous comparison
- Integrating complex binding structures
- Properties implied by world polymorphism
Conclusion

- Explicit scopes using world indices
Conclusion

- Explicit scopes using world indices
- Non-freshening opening
Conclusion

- Explicit scopes using world indices
- Non-freshening opening
- Atom abstraction as existential quantification
Conclusion

- Explicit scopes using world indices
- Non-freshening opening
- Atom abstraction as existential quantification
- Expressiveness close to a manual model with names
Questions?
Data type for explicitly typed lambda calculus

data Term
  = Var Atom
  | App Term Term
  | Lam < Atom × neutral Ty × inner Term >
  | Let < Atom × outer Term × inner Term >
Data type for explicitly typed lambda calculus

data Term outer
  = Var (Atom outer)
  | App (Term outer) (Term outer)
  | ∃inner. Lam (Atom inner) Ty (Term inner)
  | ∃inner. Let (Atom inner) (Term outer) (Term inner)
Data type for explicitly typed lambda calculus

```
data Term α
  = Var (Atom α)
  | App (Term α) (Term α)
  | ∃ β. Lam (Atom β) Ty (Term β)
  | ∃ β. Let (Atom β) (Term α) (Term β)
```
Data type for explicitly typed lambda calculus

\begin{align*}
data \; \text{Term} \; \alpha \\
&= \text{Var} \; (\text{Atom} \; \alpha) \\
&| \; \text{App} \; (\text{Term} \; \alpha) \; (\text{Term} \; \alpha) \\
&| \; \exists \; \beta. \; \text{Lam} \; (\beta \rhd \alpha) \; (\text{Atom} \; \beta) \; \text{Ty} \; (\text{Term} \; \beta) \\
&| \; \exists \; \beta. \; \text{Let} \; (\beta \rhd \alpha) \; (\text{Atom} \; \beta) \; (\text{Term} \; \alpha) \; (\text{Term} \; \beta)
\end{align*}
Data type for explicitly typed lambda calculus

data Term α
  = Var (Atom α)
  | App (Term α) (Term α)
  | Lam α<λβ→ (Ty, Term β)>
  | Let α<λβ→ (Term α, Term β)>
Polymorphic values represent closed terms
A more direct presentation of atom sorts
Generalizing Caml data structures
Picking fresh atoms

\[\text{fresh } x \text{ in } t\]
Picking fresh atoms

\[ \text{fresh } \times \text{ in } t \]

- The atom can be used in the world you like
Picking fresh atoms

\[ \text{fresh } x \text{ in } t \quad \text{where } x \# (t \downarrow) \]

- The atom can be used in the world you like
- Same proof obligation as in pure FreshML
Picking fresh atoms (second version)

\[
fresh \ x,\lnk\text{Exp},\lnk\text{Imp} \ in \ t
\]
Picking fresh atoms (second version)

fresh \ x,\text{lnkExp},\text{lnkImp} \text{ in } t

\Gamma,\beta,\text{lnkExp}:\beta\triangleright\alpha,\text{lnkImp}:\alpha\triangleright\beta,\mathbf{x}:\text{Atom} \quad \beta \vdash t : \tau

\text{where } \alpha \in \Gamma, \beta \neq \tau
Picking fresh atoms (second version)

\[ \text{fresh } x, \text{lnkExp}, \text{lnkImp in } t \]

\[
\Gamma, \beta, \text{lnkExp} : \beta \triangleright \alpha, \text{lnkImp} : \alpha \triangleright \beta, x : \text{Atom} \quad \beta \vdash t : \tau \\
\text{where } \alpha \in \Gamma, \ \beta \neq \tau
\]

- The fresh atom is in an existential world
Picking fresh atoms (second version)

\[\text{fresh } x, \text{lnkExp}, \text{lnkImp in } t\]

\[\Gamma, \beta, \text{lnkExp: } \beta \triangleright \alpha, \text{lnkImp: } \alpha \triangleright \beta, x: \text{Atom } \beta \vdash t : \tau\]

where \(\alpha \in \Gamma, \beta \neq \tau\)

- The fresh atom is in an existential world
- Links are provided to import and export things
Picking fresh atoms (second version)

\[ \text{fresh } x, \text{lnkExp}, \text{lnkImp} \text{ in } t \]

\[ \Gamma, \beta, \text{lnkExp} : \beta \triangleright \alpha, \text{lnkImp} : \alpha \triangleright \beta, x : \text{Atom} \quad \beta \vdash t : \tau \]

where \( \alpha \in \Gamma, \beta \# \tau \)

- The fresh atom is in an existential world
- Links are provided to import and export things
- Proof obligations relied to casts
Freshening is still available

\[
\text{let } (\text{Abs } \_\text{lnk } (\text{fresh } x) y) = t \text{ in } u
\]
Freshening is still available

\[
\text{let } (\text{Abs } \_\text{lnk} (\text{fresh } x) y) = t \text{ in } u
\]

Freshening allows to use the same world

\[
\Gamma \vdash t : \alpha<f> \text{ where } \alpha \in \Gamma \\
\Gamma, \_\text{lnk} : \alpha \triangleright \alpha, x: \text{Atom } \alpha, y : f \alpha \vdash u : \tau
\]
Precise control over scopes

Having explicit world subsume:

- Caml inner/outer/neutral annotations
Precise control over scopes

Having explicit world subsume:
- **CAML** inner/outer/neutral annotations
- **CAML** pattern types/expressions types distinction
Precise control over scopes

- Cαml inner/outer/neutral annotations
- Cαml pattern types/expression types distinction
- FreshML/Cαml atom sorts
Safe heterogeneous comparison!
Safe heterogeneous comparison!

\[
\text{atmEqH} :: \forall \alpha \, \beta. \text{Atom} \, \alpha \rightarrow \text{Atom} \, \beta \rightarrow (\beta \triangleright \alpha) \rightarrow \text{Bool}
\]
Safe heterogeneous comparison!

$$\text{atmEqH} :: \forall \alpha \beta. \text{Atom } \alpha \to \text{Atom } \beta \to (\beta \triangleright \alpha) \to \text{Bool}$$

$$\text{atmEqH } a \ b \ \text{Ink} \mid b \notin \text{Ink} = a \equiv \text{cast Ink } b$$
$$\mid \text{otherwise } = \text{False}$$
Substituting closed terms for variables

\[
\text{substClosed} :: \\
\forall \alpha. \text{Atm } \alpha \to (\forall \beta. \text{Term } \beta) \to \text{Term } \alpha \to \text{Term } \alpha
\]

\[
\text{substClosed } a \, v = \text{go \, id}
\]

where

\[
\text{go} :: \forall \delta. (\delta \triangleright \alpha) \to \text{Term } \delta \to \text{Term } \delta
\]

\[
\text{go \, Ink } (\text{Var } b) \mid \text{atmEqH } a \, b \, \text{Ink} = v \\
\mid \text{otherwise} \quad = \text{Var } b
\]

\[
\text{go \, Ink } (\text{App } t \, u) \\
= \text{App } (\text{go \, Ink } t) \, (\text{go \, Ink } u)
\]

\[
\text{go \, Ink } (\text{Lam Ink'} b \, ty \, t) \\
= \text{Lam Ink'} b \, ty \, (\text{go } (\text{Ink } \circ \text{Ink'}) \, t)
\]

\[
\text{go \, Ink } (\text{Let Ink'} b \, t \, u) \\
= \text{Let Ink'} b \, (\text{go \, Ink } t) \, (\text{go } (\text{Ink } \circ \text{Ink'}) \, u)
\]