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
- Cαml for its expressiveness

A taste of FreshML/Cαml

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 >
```

Capture avoiding substitution

```
subst :: (Atom, Term) → Term → Term
subst (a, v) = go
  where
    go (Var b)          = if a ≡ b then v else Var b
    go (App t u)        = App (go t) (go u)
    go (Lam<b,ty,t>) = Lam<b, ty, go t>
    go (Let<b,t,u>)  = Let<b, go t, go u>
```

Computing the size of a term

```
size :: Term → Int
size (Var _)       = 1
size (App t u)     = 1 + size t + size u
size (Lam<_,_,t>)  = 3 + size t
size (Let<_,t,u>)  = 3 + size t + size u
```

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 α-equivalence

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 α-equivalence
- ...

To freshen or not to freshen?

- While 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

$(\equiv)_{\texttt{Atom}}$ `::` $\forall\ \alpha.$ `Atom` $\alpha \rightarrow$ `Atom` $\alpha \rightarrow$ `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

```
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)
```

Worlds are closely related to each other

The type of (oriented) links between worlds

`type` $\beta \triangleright \alpha$

Worlds are closely related to each other

The type of (oriented) links between worlds

`type` $\beta \triangleright \alpha$

Links holds the set of atoms as a frontier

$\alpha \overset{(\notin S)}{\triangleright} \beta$

Worlds are closely related to each other

The type of (oriented) links between worlds

$\texttt{type}\ \beta \triangleright \alpha$

Links holds the set of atoms as a frontier

$\alpha \overset{(\notin S)}{\triangleright} \beta$

Links are supposed to be invisible/inferred!

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

$$
\begin{array}{l}
\texttt{data Term } \alpha \\
\quad = \texttt{Var (Atom } \alpha\texttt{)} \\
\quad \mid \texttt{App (Term } \alpha\texttt{) (Term } \alpha\texttt{)} \\
\quad \mid \exists\ \beta.\ \texttt{Lam (}\beta \rhd \alpha\texttt{) (Atom } \beta\texttt{) Ty (Term } \beta\texttt{)} \\
\quad \mid \exists\ \beta.\ \texttt{Let (}\beta \rhd \alpha\texttt{) (Atom } \beta\texttt{) (Term } \alpha\texttt{) (Term } \beta\texttt{)}
\end{array}
$$

Link operations

Identity link

$\mathtt{id}_{\mathtt{Link}}$ :: $\forall\ \alpha.\ \alpha \rhd \alpha$

Link operations

Identity link

$\mathtt{id}_{\mathtt{Link}}$ :: $\forall\ \alpha.\ \alpha \rhd \alpha$

Link composition

$(\circ)_{\mathtt{Link}}$ :: $\forall\ \alpha\ \beta\ \gamma.\ (\beta \rhd \gamma) \rightarrow (\alpha \rhd \beta) \rightarrow (\alpha \rhd \gamma)$

Link operations

Identity link

$\mathtt{id}_{\mathtt{Link}}$ :: $\forall\ \alpha.\ \alpha \rhd \alpha$

Link composition

$(\circ)_{\mathtt{Link}}$ :: $\forall\ \alpha\ \beta\ \gamma.\ (\beta \rhd \gamma) \to (\alpha \rhd \beta) \to (\alpha \rhd \gamma)$

Atomic link

$\mathtt{atomic}_{\mathtt{Link}}$ :: $\forall\ \alpha.\ \mathtt{Atom}\ \alpha \to (\alpha \rhd \alpha)$

Casts to walk through links

Atomic casts

$\texttt{cast}_{\texttt{Atom}}$ :: $\forall\ \alpha\ \beta.\ (\beta \triangleright \alpha) \longrightarrow (\texttt{Atom}\ \beta \longrightarrow \texttt{Atom}\ \alpha)$

Casts to walk through links

Atomic casts

$\texttt{cast}_{\texttt{Atom}}$ :: $\forall\ \alpha\ \beta.\ (\beta \rhd \alpha) \rightarrow (\texttt{Atom}\ \beta \rightarrow \texttt{Atom}\ \alpha)$

Generalized casts

$\texttt{cast}_{\texttt{f}}$ :: $\forall\ \alpha\ \beta\ \texttt{f}.\ (\beta \rhd \alpha) \rightarrow (\texttt{f}\ \beta \rightarrow \texttt{f}\ \alpha)$

Casts to walk through links

Atomic casts

$\texttt{cast}_{\texttt{Atom}} :: \forall\ \alpha\ \beta.\ (\beta \rhd \alpha) \rightarrow (\texttt{Atom}\ \beta \rightarrow \texttt{Atom}\ \alpha)$

Generalized casts

$\texttt{cast}_{\texttt{f}} :: \forall\ \alpha\ \beta\ \texttt{f}.\ (\beta \rhd \alpha) \rightarrow (\texttt{f}\ \beta \rightarrow \texttt{f}\ \alpha)$

Cast implies proof obligations or dynamic checks!

Atom abstraction as existential quantification

Hiding the real world but keeping a link

```
data α<f> = ∃ β. Abs (β ▷ α) (Atom β) (f β)
```

Data type for explicitly typed lambda calculus

$$
\begin{array}{l}
\texttt{data Term } \alpha \\
\quad = \texttt{Var (Atom } \alpha\texttt{)} \\
\quad \mid \texttt{App (Term } \alpha\texttt{) (Term } \alpha\texttt{)} \\
\quad \mid \exists\ \beta.\ \texttt{Lam (}\beta \triangleright \alpha\texttt{) (Atom } \beta\texttt{) Ty (Term } \beta\texttt{)} \\
\quad \mid \exists\ \beta.\ \texttt{Let (}\beta \triangleright \alpha\texttt{) (Atom } \beta\texttt{) (Term } \alpha\texttt{) (Term } \beta\texttt{)}
\end{array}
$$

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

$$\texttt{Abs :: } \forall\ \alpha\ \beta\ \texttt{f. } (\beta \rhd \alpha) \to \texttt{Atom } \beta \to \texttt{f } \beta \to \alpha\texttt{<f>}$$

Making an abstraction

$$\mathtt{Abs} :: \forall\ \alpha\ \beta\ \mathtt{f}.\ (\beta \triangleright \alpha) \longrightarrow \mathtt{Atom}\ \beta \longrightarrow \mathtt{f}\ \beta \longrightarrow \alpha\mathtt{<f>}$$

$$\mathtt{Lam} :: \forall\ \alpha\ \beta.\ (\beta \triangleright \alpha) \longrightarrow \mathtt{Atom}\ \beta \longrightarrow \mathtt{Term}\ \beta \longrightarrow \mathtt{Term}\ \alpha$$

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
```

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

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

$\Gamma \vdash$ `t` : α`<f>` where $\alpha \in \Gamma$

Opening an abstraction does not freshen it

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

$\Gamma \vdash$ `t : ` α`<f>` where $\alpha \in \Gamma$

Γ,β,`lnk:`$\beta \triangleright \alpha$,`x:Atom` β,`y:f` $\beta \vdash$ `u :` τ where β `#` τ

Safety Properties

- Well-typed programs do not get stuck

Safety Properties

- Well-typed programs do not get stuck
- α-equivalence is preserved by casts

Safety Properties

- Well-typed programs do not get stuck
- α-equivalence is preserved by casts
- Casts may dynamically fail or be proven successful

Safety Properties

- Well-typed programs do not get stuck
- α-equivalence is preserved by casts
- Casts may dynamically fail or be proven successful
- α-equivalence is defined structurally on types

Example commuting abstraction with pairs

```
commute :: ∀ α. α<λβ→ (Term β, Term β)>
                    → (α<Term>, α<Term>)
commute t =
  let (Abs lnk x (y,z)) = t
  in (Abs lnk x y, Abs lnk x z)
```

Name capture does not type-check

```
wrong :: ∀ α. α<Term> → Term α → α<Term>
wrong t u =
  let (Abs lnk x y) = t
  in Abs lnk x u
```

Computing the size of a term

```
size :: ∀ α. Term α → Int
size (Var _)         = 1
size (App t u)       = 1 + size t + size u
size (Lam _ _ _ t)   = 3 + size t
size (Let _ _ t u)   = 3 + size t + size u
```

Computing the size of a term

```
size :: ∀ α. Term α → Int
size (Var _)         = 1
size (App t u)       = 1 + size t + size u
size (Lam _ _ _ t)   = 3 + size t
size (Let _ _ t u)   = 3 + size t + 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 ::
   ∀ β α. (β ▷ α) → Atom β → [Atom β] → [Atom α]
remove _   _ [] = []
remove lnk a (b:bs)
  | a ≡ b = remove lnk a bs
  | otherwise = cast lnk b : remove lnk a bs
```

```
fv :: ∀ α. Term α →  [Atom α]
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 lnk env b ty)
  | a ≡ b     = ty
  | otherwise = lookupEnv (cast lnk 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
- α-equivalence for inside-out abstractions

Challenges and future work

- Deeper formalization and proofs
- α-equivalence for inside-out abstractions
- Better understanding of heterogeneous comparison

Challenges and future work

- Deeper formalization and proofs
- α-equivalence for inside-out abstractions
- Better understanding of heterogeneous comparison
- Integrating complex binding structures

Challenges and future work

- Deeper formalization and proofs
- α-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
- Expressivness 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

```
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)
```

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 Cαml data structures

Picking fresh atoms

```
fresh x in t
```

Picking fresh atoms

```
fresh x in t
```

- The atom can be used in the world you like

Picking fresh atoms

`fresh x in t` where $\mathtt{x}\#(\mathtt{t}\Downarrow)$

- The atom can be used in the world you like
- Same proof obligation as in pure FreshML

Picking fresh atoms (second version)

```
fresh x,lnkExp,lnkImp in t
```

Picking fresh atoms (second version)

```
fresh x,lnkExp,lnkImp in t
```

Γ,β,lnkExp:$\beta \triangleright \alpha$,lnkImp:$\alpha \triangleright \beta$,x:Atom $\beta \vdash$ t : τ
where $\alpha \in \Gamma$, β # τ

Picking fresh atoms (second version)

```
fresh x,lnkExp,lnkImp in t
```

Γ,β,lnkExp:$\beta \triangleright \alpha$,lnkImp:$\alpha \triangleright \beta$,x:Atom $\beta \vdash$ t : τ
where $\alpha \in \Gamma$, β # τ

- The fresh atom is in an existential world

Picking fresh atoms (second version)

```
fresh x,lnkExp,lnkImp in t
```

Γ,β,lnkExp:$\beta \triangleright \alpha$,lnkImp:$\alpha \triangleright \beta$,x:Atom $\beta \vdash$ t : τ
    where $\alpha \in \Gamma$, β # τ

- The fresh atom is in an existential world
- Links are provided to import and export things

Picking fresh atoms (second version)

```
fresh x,lnkExp,lnkImp in t
```

Γ,β,`lnkExp`:$\beta \triangleright \alpha$,`lnkImp`:$\alpha \triangleright \beta$,`x:Atom` $\beta \vdash$ `t` : τ
  where $\alpha \in \Gamma$, β # τ

- 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

```
let (Abs _lnk (fresh x) y) = t in u
```

Freshening is still available

```
let (Abs _lnk (fresh x) y) = t in u
```

Freshening allows to use the same world

$\Gamma \vdash$ `t : `α`<f> where `$\alpha \in \Gamma$

Γ`,_lnk:`$\alpha \triangleright \alpha$`,x:Atom `$\alpha$`,y:f `$\alpha \vdash$` u : `$\tau$

Precise control over scopes

Having explicit world subsume:

- Cαml inner/outer/neutral annotations

Precise control over scopes

Having explicit world subsume:

- Cαml inner/outer/neutral annotations
- Cαml pattern types/expression types distinction

Precise control over scopes

Having explicit world subsume:

- Cαml inner/outer/neutral annotations
- Cαml pattern types/expression types distinction
- FreshML/Cαml atom sorts

Safe heterogeneous comparison!

Safe heterogeneous comparison!

```
atmEqH :: ∀ α β. Atom α → Atom β → (β ▷ α) → Bool
```

Safe heterogeneous comparison!

```
atmEqH :: ∀ α β. Atom α → Atom β → (β ▷ α) → Bool
```

```
atmEqH a b lnk | b ∉ lnk   = a ≡ cast lnk b
               | otherwise = False
```

Substituting closed terms for variables

```
substClosed ::
    ∀ α. Atm α → (∀ β. Term β) → Term α → Term α
substClosed a v = go id
  where
    go :: ∀ δ. (δ ▷ α) → Term δ → Term δ
    go lnk (Var b) | atmEqH a b lnk = v
                   | otherwise      = Var b
    go lnk (App t u)
      = App (go lnk t) (go lnk u)
    go lnk (Lam lnk' b ty t)
      = Lam lnk' b ty (go (lnk ◦ lnk') t)
    go lnk (Let lnk' b t u)
      = Let lnk' b (go lnk t) (go (lnk ◦ lnk') u)
```