Generative Names and Dependent Types

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory
Generative Names and Dependent Types: from FreshML to ‘FreshAgda’

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory
What did FreshML give the world?

Shinwell+AMP+Gabbay
ICFP 2003
What did FreshML give the world?

HOPE
What did FreshML give the world?

HOPE

higher-order functional programming

+ generative names that are permutable

\[ \nu a.e \quad \text{swap } a, b \text{ in } e \]
What did FreshML give the world?

LOVE
What did FreshML give the world?

LOVE

lots of very elegant binder-manipulating algorithms expressed in a familiar ‘nameful’ way
Inductive types with $\alpha$-abstraction

names Var -- a type of permutable, generative names

data Term where
   -- inductive type of $\lambda$-terms mod $\alpha$
   V : Var -> Term -- variable
   A : (Term \times Term) -> Term -- application term
   L : (Var . Term) -> Term -- $\lambda$-abstraction term

_/\_ : Term -> Var -> Term -> Term -- capture-avoiding substitution
(t / x)(V x1) = if x = x1 then t else V x1
(t / x)(A(t1 , t2)) = A((t / x )t1 , (t / x )t2)
(t / x)(L(x1 . t1)) = L(x1 . (t / x)t1)

Can freely mix _ . _ and _ -> _ to get more subtle examples (e.g. for NbE).
Inductive types with $\alpha$-abstraction

Underlying calculus:

introduction: $\alpha a.\ e$ ($\alpha$-abstraction)

elimination: $e @ e'$ (concretion)

reduction: $(\alpha a.\ e) @ e' \rightarrow \nu a.\ (\text{swap } a, e' \text{ in } e)$ $a \not\equiv e'$
What did FreshML give the world?

**HOPE**

higher-order functional programming

+ generative names that are permutable

\( v a . e \)

& no future in re-engineering general-purpose HOFLs (Shinwell’s Fresh OCaml is no longer supported)—be domain-specific instead

ML & Haskell already have this, but not (??) this

\( \text{swap } a , b \text{ in } e \)
Aim

Constructive Type Theory + generative, permutable names: combine (total) FreshML with Agda/Coq.

Application domain: formal proofs about operational semantics.
Aim

Constructive Type Theory + generative, permutable names: combine (total) FreshML with Agda/Coq.

Application domain: formal proofs about operational semantics.

Design criteria:

- [ease-of-use] no monad syntax

‘do’ notation is unnecessarily sequential for generative names
Aim

Constructive Type Theory + generative, permutable names: combine (total) FreshML with Agda/Coq.

Application domain: formal proofs about operational semantics.

Design criteria:

- [ease-of-use] no monad syntax
- [ease-of-use] no bunched contexts, just $\nu$

cf. previous work on nominal type theory by Schöpp-Stark and Cheney
Aim

Constructive Type Theory + generative, permutable names: combine (total) FreshML with Agda/Coq.

Application domain: formal proofs about operational semantics.

Design criteria:

- [ease-of-use] no monad syntax
- [ease-of-use] no bunched contexts, just $\nu$
- [technical] Curry-Howard for nominal logic’s freshness quantifier: proofs of $\alpha$-structural induction

$= \alpha$-structurally recursive programs

Dependently typed $\alpha$-abstraction

\[
\Gamma, a : \text{Name} \vdash A : \text{Set} \\
\hline
\Gamma \vdash \forall a. A : \text{Set}
\]

- For simplicity, assume just one type \text{Name} of names and write $\forall a. A$ instead of $\forall a: \text{Name}. A$.
- When $a$ does not occur in $A$, then $\forall a. A$ should be like FreshML’s type \text{Name}. $A$ of $\alpha$-abstractions (cf. $(x:A) \rightarrow B$ versus $A \rightarrow B$).
Dependently typed $\alpha$-abstraction

\[ \Gamma, a : \text{Name} \vdash A : \text{Set} \]
\[ \Gamma \vdash \forall a. A : \text{Set} \]

\[ \Gamma, a : \text{Name} \vdash e : A \]
\[ \Gamma \vdash \alpha a. e : \forall a. A \]

\[ \Gamma \vdash e : \forall a. A \]
\[ \Gamma \vdash e' : \text{Name} \]
\[ \Gamma \vdash e @ e' : \nu a. (\text{swap } a, e' \text{ in } A) \]
Dependently typed $\alpha$-abstraction

$I$-formation: \[
\frac{\Gamma, a : \text{Name} \vdash A : \text{Set}}{\Gamma \vdash \forall a. A : \text{Set}}
\]

$I$-introduction: \[
\frac{\Gamma, a : \text{Name} \vdash e : A}{\Gamma \vdash \alpha a. e : \forall a. A}
\]

$I$-elimination: \[
\frac{\Gamma \vdash e : \forall a. A \quad \Gamma \vdash e' : \text{Name}}{\Gamma \vdash e @ e' : \forall a. (\text{swap } a, e' \text{ in } A)}
\]

permutative, not substitutive, dependency types on names
Dependently typed $\alpha$-abstraction

$I$-formation:  \[
\begin{array}{c}
\Gamma, a : \text{Name} \vdash A : \text{Set} \\
\hline
\Gamma \vdash \forall a. A : \text{Set}
\end{array}
\]

$I$-introduction:  \[
\begin{array}{c}
\Gamma, a : \text{Name} \vdash e : A \\
\hline
\Gamma \vdash \forall a. e : \forall a. A
\end{array}
\]

$I$-elimination:  \[
\begin{array}{c}
\Gamma \vdash e : \forall a. A \quad \Gamma \vdash e' : \text{Name} \\
\hline
\Gamma \vdash e @ e' : \nu a. (\text{swap } a, e' \text{ in } A)
\end{array}
\]

Generative names in types
Dependently typed $\alpha$-abstraction

$\forall$-formation: \[
\Gamma, a : \text{Name} \vdash A : \text{Set} \quad \frac{}{\Gamma \vdash \forall a. A : \text{Set}}
\]

$\forall$-introduction: \[
\Gamma, a : \text{Name} \vdash e : A \quad \frac{}{\Gamma \vdash \forall a. e : \forall a. A}
\]

$\forall$-elimination: \[
\Gamma \vdash e : \forall a. A \quad \Gamma \vdash e' : \text{Name} \quad \frac{}{\Gamma \vdash e@e' : \forall a. (\text{swap } a, e' \text{ in } A)}
\]

$\forall$-equality: \[
\Gamma, a : \text{Name} \vdash e : A \quad \Gamma \vdash e' : \text{Name} \quad \frac{}{\Gamma \vdash (\forall a. e)@e' \equiv \forall a. (\text{swap } a, e' \text{ in } e) : \forall a. (\text{swap } a, e' \text{ in } A)}
\]

what does ‘$\equiv$’ mean for expressions with generative names?
Decidable equality for generative expressions

\[
\begin{align*}
\nu a . e &= e & (a \neq e) \\
\nu a . \nu b . e &= \nu b . \nu a . e \\
E[\nu a . e] &= \nu a . E[e] & (a \neq E) \\
(\lambda x \rightarrow e) \nu &= e[\nu/x] & \text{[Plotkin’s } \beta \nu \text{]}
\end{align*}
\]

evaluation contexts: \( E ::= \bullet | E e | \nu E | \nu a . E | \cdots \)
expressions: \( e ::= x | a | \lambda x \rightarrow e | e e | \nu a . e | \cdots \)
canonical forms: \( \nu ::= a | \lambda x \rightarrow e | u | \cdots \)
neutral forms: \( u ::= x | u v | \cdots \)
Decidable equality for generative expressions

\[ \nu a. e = e \quad (a \neq e) \]
\[ \nu a. \nu b. e = \nu b. \nu a. e \]
\[ E[\nu a. e] = \nu a. E[e] \quad (a \neq E) \]
\[ (\lambda x \rightarrow e) \nu = e[\nu/x] \quad \text{[Plotkin's } \beta \nu] \]

evaluation contexts: \[ E ::= \bullet | E e | \nu E | \nu a. E | \cdots \]
expressions: \[ e ::= x | a | \lambda x \rightarrow e | e e | \nu a. e | \cdots \]
canonical forms: \[ \nu ::= a | \lambda x \rightarrow e | u | \cdots \]
neutral forms: \[ u ::= x | u \nu | \cdots \]

References? (N.B. open expressions; and definition of \( E/\nu/u \) in presence of \( \Pi- \), \( \Sigma- \) & \( \Upeta- \) types is subtle.)
Generative names creep into the pure CTT fragment

Conventional **Π**-elimination:

\[
\begin{align*}
\Gamma & \vdash e_1 : (x : A) \rightarrow B \\
\Gamma & \vdash e_2 : A \\
\hline
\Gamma & \vdash e_1 e_2 : B[e_1/x]
\end{align*}
\]

**Nu** **Π**-elimination:

\[
\begin{align*}
\Gamma & \vdash e_1 : (x : A) \rightarrow B \\
\Gamma & \vdash e_2 = \nu \vec{a}. \nu : A \\
\hline
\Gamma & \vdash e_1 e_2 : \nu \vec{a}. B[\nu/x]
\end{align*}
\]
Done:

- Declarative type system with $\Sigma/\Pi/\text{Set} + \nu/\text{swap}/\mathcal{N}$.

Semi-done:

- Model using **nominal sets** (specifically, a version of Moggi’s dynamic allocation monad on the universe of ‘FM-sets’ of Gabbay+AMP).

Not done:

- Decidability of type-checking (via algorithmic type system equivalent to the declarative one).
- Inductive types + dependently typed pattern-matching.
- Implementation.