Names are (mostly) Useless:

Encoding Nominal Logic Programming Techniques with Use-counting and Dependent Types

Jason Reed

September 20, 2008
Workshop on Mechanizing Metatheory

Binding and Names

- There are various familiar ways of handling binding
- HOAS, Nominal Logic, deBruijn indices, etc.
- **Nominal logic** supposed to allow particularly easy reasoning about **disequality**, **apartness**: primitive apartness relation $a\#b$

Example: α-inequality of λ-terms (in Nominal Logic Programming)

[taken from Cheney, Urban '06]

$\mathit{var} : \mathit{name} \rightarrow \mathit{term}$

$\mathit{lam} : \langle \mathit{name} \rangle \mathit{term} \rightarrow \mathit{term}$

$\mathit{aneq}\ (\mathit{lam}\ \langle x \rangle E)\ (\mathit{lam}\ \langle x \rangle E')\ :\!-\ \mathit{aneq}\ E\ E'$

$\mathit{aneq}\ (\mathit{var}\ X)\ (\mathit{var}\ Y)\ :\!-\ X \# Y$

$\ldots$

Example: α-inequality of λ-terms (in HOAS)

$var : name \rightarrow term$

$lam : (name \rightarrow term) \rightarrow term$

$aneq\ (lam\ E)\ (lam\ E') \mathrel{:}\!-\ \Pi x{:}name.aneq\ (E\ x)\ (E'\ x)$

$aneq\ (var\ X)\ (var\ Y) \mathrel{:}\!-$?

Problem: last clause (apparently) can't help but match even when X and Y are equal.

Even worse with usual HOAS encoding of terms where variables are not specially distinguished!

Alternate HOAS Encoding

- Actually could tediously keep track of and pass around a list of names discovered so far each time a new name is introduced
- Effectively implement apartness manually by walking through this list
- Not terrifically satisfying

Another Idea

- Use concepts from **resource-sensitive** substructural logics (e.g. linear logic) to get simple **encoding** of apartness relation
  - without introducing it as primitive as in nominal logic
  - without explicit list-passing or -crawling as in HOAS above

Sketch

- **Declare** $X\#Y$ as a relation, with kind something like $name \to name \to \mathsf{type}$.
- **Define** $X\#Y$ with one clause something like $\Pi X{:}name.\Pi Y{:}name.X\#Y$.
- But we don't want **any** X and Y in this relation, just **different** ones
- So **consume** each argument linearly to enforce disjointness: think '$name \multimap name \multimap \cdots$'
- Want some kind of **linear Pi**, so we can say something like $\Pi X \hat{:} name.\Pi Y \hat{:} name.X\#Y$.
- **Key Idea 1**: *Use disjointness of linear resources to model apartness of names*

Problem with Linear Dependent Types

Naïvely combining linearity with dependency can lead to serious problems.

Suppose we tried to typecheck

$\lambda x.\lambda y.(y \mathbin{\hat{}} x) : \Pi x \mathbin{\hat{:}} o.\Pi y{:}(o \multimap fam \mathbin{\hat{}} x).fam \mathbin{\hat{}} x$

in the signature

$o : \mathsf{type}. \qquad fam : o \multimap \mathsf{type}.$

Then we'd get:

$$\frac{x \mathbin{\hat{:}} o \vdash x : o \qquad y : o \multimap fam \mathbin{\hat{}} x \vdash y : o \multimap fam \mathbin{\hat{}} x}{x \mathbin{\hat{:}} o,\ y : o \multimap fam \mathbin{\hat{}} x \vdash y \mathbin{\hat{}} x : fam \mathbin{\hat{}} x}$$

Context splitting strands y away from x!

Solution

- Can't seem to have relations (type families) themselves actually **use** (consume) resources linearly
- But we still need to **mention** linear resources, e.g. in the clause: $\Pi X \mathrel{\hat{:}} name.\Pi Y \mathrel{\hat{:}} name.X\#Y$.
- Introduce '**Useless**' function type $A \not\multimap B$, useless function kind $A \not\multimap \mathsf{type}$ to allow **mention** without **use**
- Will have $\# : name \not\multimap name \not\multimap \mathsf{type}$
- **Key Idea 2**: *Use useless functions to reconcile linearity with the dependency of the type family # on names that are resources*

Plan

- Sketch appropriate **logic** for encoding
- Show how **apartness** is encoded
- Examples of **use** of apartness relation

n-Linear Logic

- Useless functions and linear Pi are both instances of a more general **n-linear dependent function type** $\Pi x{:}^{n} A.B$
- Function uses its argument **exactly** n times
- Useless: $n = 0$ ($A \not\multimap B = \Pi x{:}^{0} A.B$)
- Linear: $n = 1$ ($\Pi x \mathbin{\hat{:}} A.B = \Pi x{:}^{1} A.B$)
- Note that if $\lambda x.M : \Pi x{:}^{n} A.B$, then x is used n times in M, not in B!
- In fact x will be required to be **used** zero times in B, but may still get **mentioned** in B (B might contain as a subterm e.g. $c \,\hat{}\, x$ for $c : A \not\multimap A'$)

Judgmental Setup

$(x :^{n} A)$ means: x gets used exactly n times

$$\Delta ::= x_1 :^{n_1} A_1, \ldots, x_K :^{n_K} A_K$$

$$\Gamma ::= x_1 : B_1, \ldots, x_K : B_K$$

Typing judgment:

$$\Delta; \Gamma \vdash M : C$$

n-Linear dependent function types

$$\frac{\Gamma; \Delta, \boxed{x :^n A} \vdash M : B}{\Gamma; \Delta \vdash \hat{\lambda} x.M : \Pi x{:}^n A.B}$$

$$\frac{\Gamma; \Delta_1 \vdash M : \Pi x{:}^n A.B \qquad \Gamma; \Delta_2 \vdash N : A}{\Gamma; \boxed{\Delta_1 + n \cdot \Delta_2} \vdash M \,\hat{}\, N : [N/x]B}$$

$$(x :^n A) + (x :^m A) = (x :^{n+m} A)$$

$$n \cdot (x :^m A) = (x :^{nm} A)$$

Ordinary dependent function types

$$\frac{\Gamma,\ x : A\ ; \Delta \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : \Pi x{:}A.B}$$

$$\frac{\Gamma; \Delta \vdash M : \Pi x{:}A.B \qquad \Gamma;\ 0 \cdot \Delta\ \vdash N : A}{\Gamma; \Delta \vdash M\, N : [N/x]B}$$

Use of Variables

$$\frac{x : A \in \Gamma}{\Gamma;\ 0 \cdot \Delta \vdash x : A} \qquad \frac{}{\Gamma;\ (x :^1 A) + 0 \cdot \Delta \vdash x : A}$$

Additives

$$\frac{}{\Gamma;\Delta \vdash \langle\rangle : \top} \qquad \frac{\Gamma;\Delta \vdash M : A \qquad \Gamma;\Delta \vdash N}{\Gamma;\Delta \vdash \langle M, N\rangle : A\&B}$$

$$\frac{\Gamma;\Delta \vdash M : A\&B}{\Gamma;\Delta \vdash \pi_1 M : A} \qquad \frac{\Gamma;\Delta \vdash M : A\&B}{\Gamma;\Delta \vdash \pi_2 M : B}$$

Well-Formedness of Dependent Types

$$\frac{\Gamma; \Delta, \boxed{x :^0 A} \vdash B : \mathsf{type}}{\Gamma; \Delta \vdash \Pi x{:}^n A.B : \mathsf{type}} \qquad \frac{\Gamma, \boxed{x : A}; \Delta \vdash B : \mathsf{type}}{\Gamma; \Delta \vdash \Pi x{:}A.B : \mathsf{type}}$$

- Argument of a (n-)linear Π is required to "be used **zero** times" in the body of the type.
- Safe generalization of usual requirement that it is not **mentioned** to occur (i.e. the nondependent function type $\multimap$)
- Strict generalization because other constants used in B may have types like $C \not\multimap D$, which promise that they use their substructural argument zero times.

Encoding Apartness

$name$: type .

$\# : name \not\multimap name \not\multimap \mathsf{type}$

$irrefl : \Pi X{:}^1 name. \Pi Y{:}^1 name. \; (X \# Y \;\circ\!- \top)$

That's it!

Encoding Apartness

$name : \mathsf{type}$.

$\# : name \multimap\!\!\!\!/\ name \multimap\!\!\!\!/\ \mathsf{type}$

$irrefl : \Pi X{:}^1 name.\Pi Y{:}^1 name.\ (X \# Y \ \circ\!\!-\ \top\ )$

Note that:

- $X \# Y$ short for $\#\,\hat{}\,X\,\hat{}\,Y$
- $\circ\!\!-\ \top$ because other names besides X and Y may be present
- Resources hypotheses of names consumed in **derivation** of apartness and not in **formation** of the apartness relation

Encoding α-inequality

$var : name \not\multimap term$

$lam : (name \not\multimap term) \to term$

$\_ : aneq\ (lam\ E)\ (lam\ E') \multimapinv (\Pi x{:}^1 name.aneq\ (E \mathbin{\hat{}} x)\ (E' \mathbin{\hat{}} x))$

$\_ : aneq\ (var\ X)\ (var\ Y) \multimapinv X \# Y$

$\cdots$ (more cases, just as in nominal logic program)

Encoding α-inequality

$var : name \not\multimap term$

$lam : (name \not\multimap term) \rightarrow term$

$\_ : aneq\ (lam\ E)\ (lam\ E') \multimapinv (\Pi x{:}^{1} name.aneq\ (E \mathbin{\hat{}} x)\ (E' \mathbin{\hat{}} x))$

$\_ : aneq\ (var\ X)\ (var\ Y) \multimapinv X \# Y$

- Functions over names are 0-linear dependent functions. (“*Names are Useless*”)

Encoding α-inequality

$var : name \not\multimap term$

$lam : (name \not\multimap term) \to term$

$\_ : aneq\ (lam\ E)\ (lam\ E') \multimapinv (\Pi x{:}^1 name.aneq\ (E \mathbin{\hat{}} x)\ (E' \mathbin{\hat{}} x))$

$\_ : aneq\ (var\ X)\ (var\ Y) \multimapinv X \# Y$

- Functions over names are 0-linear dependent functions.
- Linear functions automatically propagate the set of names.

Encoding α-inequality

$var : name \not\multimap term$

$lam : (name \not\multimap term) \rightarrow term$

$\_ : aneq\ (lam\ E)\ (lam\ E') \multimapinv (\ \Pi x{:}^1 name\ .aneq\ (E \mathbin{\hat{}} x)\ (E' \mathbin{\hat{}} x))$

$\_ : aneq\ (var\ X)\ (var\ Y) \multimapinv X \# Y$

- Functions over names are 0-linear dependent functions.
- Linear functions automatically propagate the set of names.
- 1-linear dependent function abstracts over new name.

The Encoding In Action

(abbreviate *name* as n)

$$\dfrac{\dfrac{x_1 :^1 n, x_3 :^1 n \vdash \top \qquad x_2 :^1 n \vdash x_2 : n \qquad x_4 :^1 n \vdash x_4 : n}{x_1 :^1 n, x_2 :^1 n, x_3 :^1 n, x_4 :^1 n \vdash x_4 \# x_2}}{x_1 :^1 n, x_2 :^1 n, x_3 :^1 n, x_4 :^1 n \vdash aneq\ (var\ x_4)\ (var\ x_2)}$$

Recall: $irrefl : \Pi X{:}^1 name. \Pi Y{:}^1 name.\ (X \# Y \multimapinv \top)$

$$\dfrac{\dfrac{x_1 :^1 n, x_3 :^1 n \vdash \top \qquad x_2 :^X n \vdash x_2 : n \qquad x_2 :^X n \vdash x_2 : n}{x_1 :^1 n, x_2 :^1 n, x_3 :^1 n, x_4 :^1 n \vdash x_2 \# x_2}}{x_1 :^1 n, x_2 :^1 n, x_3 :^1 n, x_4 :^1 n \vdash aneq\ (var\ x_2)\ (var\ x_2)}$$

Problem: no $X \in \mathbb{N}$ s.t. $X + X = 1$

Encoding a Programming Language with Store

$eval : store \rightarrow exp \rightarrow result \rightarrow \mathsf{type}$

$letref : val \rightarrow (val \rightarrow exp) \rightarrow exp$ % $\mathbf{let}\, x = \mathbf{ref}\, v\, \mathbf{in}\, e$

$let! : val \rightarrow (val \rightarrow exp) \rightarrow exp$ % $\mathbf{let}\, x = (!v)\, \mathbf{in}\, e$

$loc : name \nrightarrow val$

$((\_, \_) :: \_)\ : name \nrightarrow val \rightarrow store \rightarrow store$

Consider a small CPS language with updatable store represented as a list of name/value pairs.

Encoding a Programming Language with Store

$eval : store \to exp \to result \to \mathsf{type}$

$letref : val \to (val \to exp) \to exp$ % $\mathbf{let}\ x = \mathbf{ref}\ v\ \mathbf{in}\ e$

$let! : val \to (val \to exp) \to exp$ % $\mathbf{let}\ x = (!v)\ \mathbf{in}\ e$

$loc : name \multimap val$

$((\_, \_) :: \_) : name \multimap val \to store \to store$

$\_ : eval\ S\ (letref\ V\ E)\ R \multimapinv \Pi\ell{:}^1 n.\ eval\ ((\ell, V) :: S)\ (E\ (loc\,\hat{}\,\ell))\ R$

$\_ : eval\ S\ (let!\ (loc\,\hat{}\,L)\ E)\ R \multimapinv (lookup\ S\,\hat{}\,L\ V\ \&\ eval\ S\ (E\ V)\ R)$

$lookup : store \to name \multimap val \to \mathsf{type}$

$\_ : lookup\ ((N, V) :: S)\,\hat{}\ N\ V \multimapinv \top$

$\_ : lookup\ ((N', \_) :: S)\,\hat{}\ N\ V \multimapinv (N\#N'\ \&\ lookup\ S\,\hat{}\ N\ V)$

Reasoning in a Programming Language with Store

$\mathit{wfstore} : \mathit{store} \rightarrow \mathsf{type}$

$\mathit{notin} : \mathit{name} \not\multimap \mathit{store} \rightarrow \mathsf{type}$

$\_ : \mathit{wfstore}\ \mathit{nil} \circ\!- \top$

$\_ : \mathit{wfstore}\ ((\ N\ , \_) :: S) \circ\!- (\mathit{notin}\,\hat{}\ N\ \ S\ \&\ \mathit{wfstore}\ S)$

$\_ : \mathit{notin}\,\hat{}\ N\ \ \mathit{nil} \circ\!- \top$

$\_ : \mathit{notin}\,\hat{}\ N\ \ ((\ N'\ , \_) :: S) \circ\!- (\mathit{notin}\,\hat{}\ N\ \ S\ \&\ N \,\#\, N'\ )$

Or: could use substructural features directly, for shorter or more expressive encoding

$\mathit{wfstore}' : \mathit{store} \rightarrow \mathsf{type}$

$\_ : \mathit{wfstore}'\ \mathit{nil} \circ\!- \top$ (or just $\_ : \mathit{wfstore}'\ \mathit{nil}$)

$\_ : \ \Pi x{:}^{1}\mathit{name}\ .(\mathit{wfstore}'\ S \multimap \mathit{wfstore}'\ ((\ x\ , \_) :: S))$

Related Work

- n-use functions [Wright, Momigliano]
- Other 0-use ("irrelevant") functions [Pfenning, Ley-Wild]
- RLF [Ishtiaq, Pym]
- HLF
  - Designed for statement of metatheorems for Linear LF.
  - Does n-linear Πs above, and more (e.g. some of BI)
  - Prototype implementation

Conclusion

- **Key Idea 1**: *Use disjointness of linear resources to model apartness of names*
- **Key Idea 2**: *Use useless functions to reconcile linearity with the dependency of the type family # on names that are resources*
- Substructural dependent types can imitate nominal logic programming techniques
- Practical?
- In what ways does it do even better?

Thanks