A Universe of Binding and Computation

Dan Licata and Robert Harper
Carnegie Mellon University

Goal

Functional programming with binding and scope

Applications:

- Domain-specific logics for reasoning about code
- Mechanized metatheory

Goal

Functional programming with binding and scope

Applications:

- Domain-specific logics for reasoning about code
- Mechanized metatheory

Two important ingredients...

Binding

Represent bound variables:

lam(x.e) , $\forall x{:}\tau.A$, hypothetical judgements

Binding

Represent bound variables:

lam(x.e) , $\forall x{:}\tau.A$, hypothetical judgements

E.g. type exp representing syntax of λ-terms:

$$\text{app} : \text{exp} \Rightarrow \text{exp} \Rightarrow \text{exp}$$

$$\text{lam} : (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp}$$

Binding

Represent bound variables:

lam(x.e) , ∀x:τ.A , hypothetical judgements

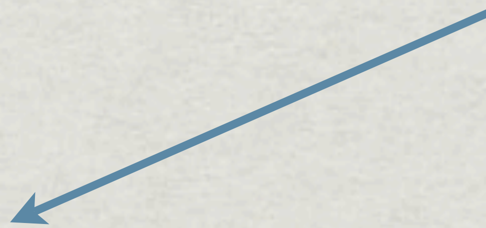
E.g. type exp representing syntax of λ-terms:

$$\text{app} : \text{exp} \Rightarrow \text{exp} \Rightarrow \text{exp}$$

$$\text{lam} : (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp}$$

Computation

pattern-matching
recursive function



normalize : exp ⊃ exp
normalize (lam x.e) = ...
normalize (app e1 e2) = …

Our Approach

Our Approach

1. Makes an *a priori* type distinction between $\Rightarrow$ (binding) and $\supset$ (computation)

   [unlike Parametric & Weak HOAS / Hybrid]

Our Approach

1. Makes an *a priori* type distinction between $\Rightarrow$ (binding) and $\supset$ (computation)

   [unlike Parametric & Weak HOAS / Hybrid]

2. As two types in the same language [unlike Twelf/Delphin/Beluga]

Our Approach

1. Makes an *a priori* type distinction between $\Rightarrow$ (binding) and $\supset$ (computation)

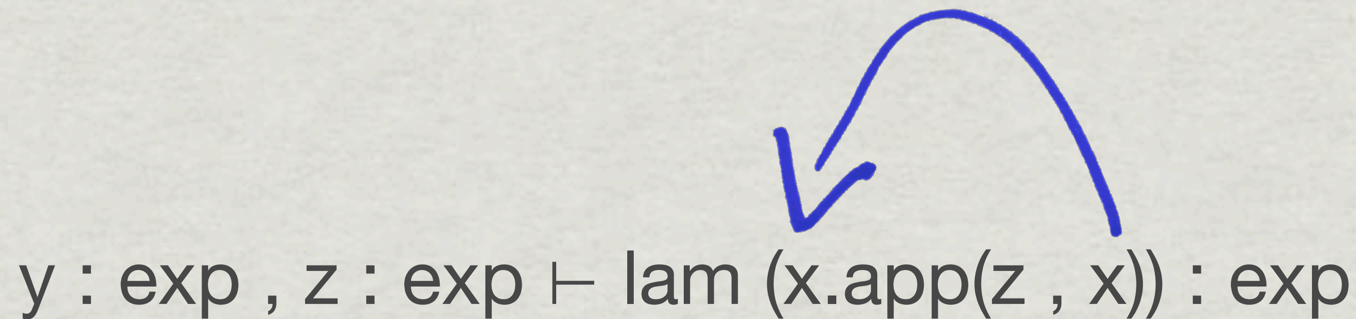
   [unlike Parametric & Weak HOAS / Hybrid]

2. As two types in the same language
   [unlike Twelf/Delphin/Beluga]

3. Treats variables *pronominally*
   [unlike nominal logic / FreshML / cαml]

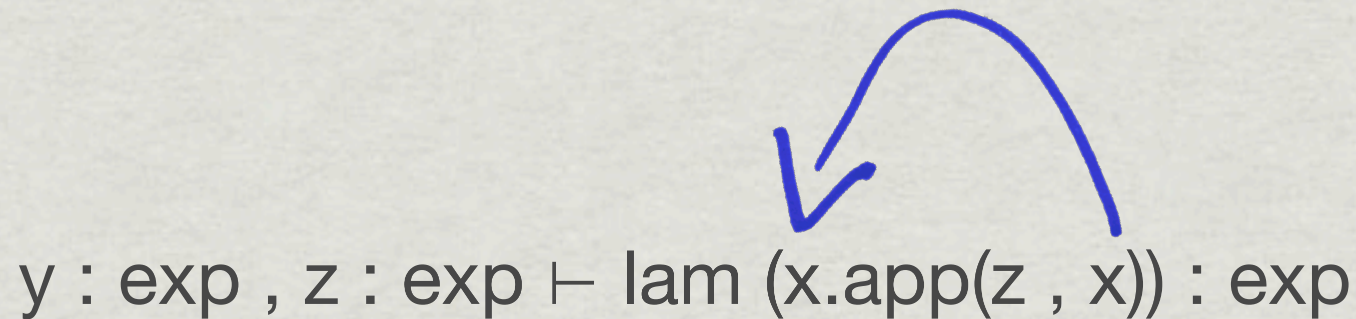
Pronominal Variables

Treat variables as **pronouns**, not nouns: reference to a designated binding site



Pronominal Variables

Treat variables as **pronouns**, not nouns:
reference to a designated binding site



Can test
equality of variables
as pointers,
not as names

Contextual Types

Contextual types <Ψ> A track scoping of variables:

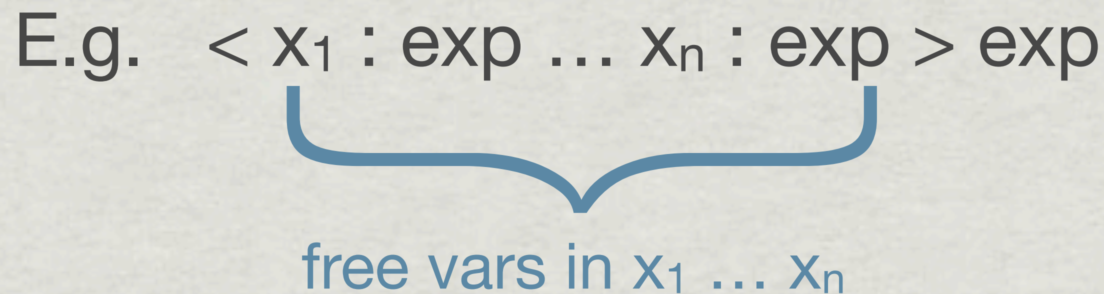
E.g. $< x_1 : exp \ldots x_n : exp > exp$

free vars in $x_1 \ldots x_n$

Contextual Types

Contextual types <Ψ> A track scoping of variables:

E.g. $< x_1 : exp \ldots x_n : exp >$ exp



Permit precise types for computations:

normalize : $<\cdot>$exp $\supset$ $<\cdot>$exp

normalize/open : $\forall$ Ψ. <Ψ>exp $\supset$ <Ψ>exp

7

This Paper

- Implement a framework as a *universe* in Agda
- Examples and comparisons with Twelf/Delphin/Beluga/FreshML
- Datatype-generic implementations of the structural properties

This Paper

- **Implement framework as a *universe* in Agda**
- Examples and comparisons with Twelf/Delphin/Beluga/FreshML
- Datatype-generic implementations of the structural properties

Universe

Ingredients:

- Datatype of codes naming a user-defined collection of types

  data Code where …

- Interpretation function maps codes to Agda Sets:

  Elements : Code → Set

Universe

Ingredients:

- Datatype of codes naming a user-defined collection of types

  data Code where …

- Interpretation function maps codes to Agda Sets:

  Elements : Code → Set

This work: universe of **contextual** types

Contextual Universe

- Datatype of codes for contextual types:

  data Ctx $\Psi ::= \cdot \mid \Psi , D$

  data Code $A ::= A\ \mathsf{list} \mid A \supset B \mid D \mid \Psi \Rightarrow A \mid \forall \Psi.\ A$

- Interpretation $\langle\Psi\rangle A$:

  $\langle_\rangle_ : \mathrm{Ctx} \to \mathrm{Code} \to \mathrm{Set}$

11

Interpretation

Context

Code for a Contextual Type

Agda Set

$$<\Psi>\ (A\ \mathsf{list}) = \mathsf{List}\ <\Psi>A$$

$$<\Psi>\ (A \supset B) = <\Psi>A \rightarrow <\Psi>B$$

$$<\Psi>\ (\Psi' \Rightarrow A) = <\Psi,\Psi'>A$$

$$<\Psi>\ (\forall\ \Psi'.\ A) = (\Psi' : \mathsf{Ctx}) \rightarrow <\Psi>(A\ \Psi')$$

$$<\Psi>\ D = \ldots$$

Interpretation

```
app : (exp * exp) ⇒ exp

lam : (exp ⇒ exp) ⇒ exp


<Ψ> exp     = Expr Ψ where

  data Expr : Ctx → Set where
      lam : <Ψ>(exp ⇒ exp) → Expr Ψ

      app : <Ψ>(exp  *  exp) → Expr Ψ
      var   : (exp ∊ Ψ) → Expr Ψ
```


13

Pronominal Variables

var : (exp ∈ Ψ) → Expr Ψ

```
data _∈_ : Datatype → Ctx → Set where
  i0 : D ∈ (Ψ , D)
  iS : (D ∈ Ψ) → D ∈ (Ψ , D')
```

This Paper

- Implement framework as a *universe* in Agda
- **Examples and comparisons with Twelf/Delphin/Beluga/FreshML**
- Datatype-generic implementations of the structural properties

Scope-correct NBE

Normalize syntactic λ-terms by interpreting them as computational functions ⊃ in the metalanguage

Scope-correct NBE

Normalize syntactic λ-terms by interpreting them as computational functions ⊃ in the metalanguage

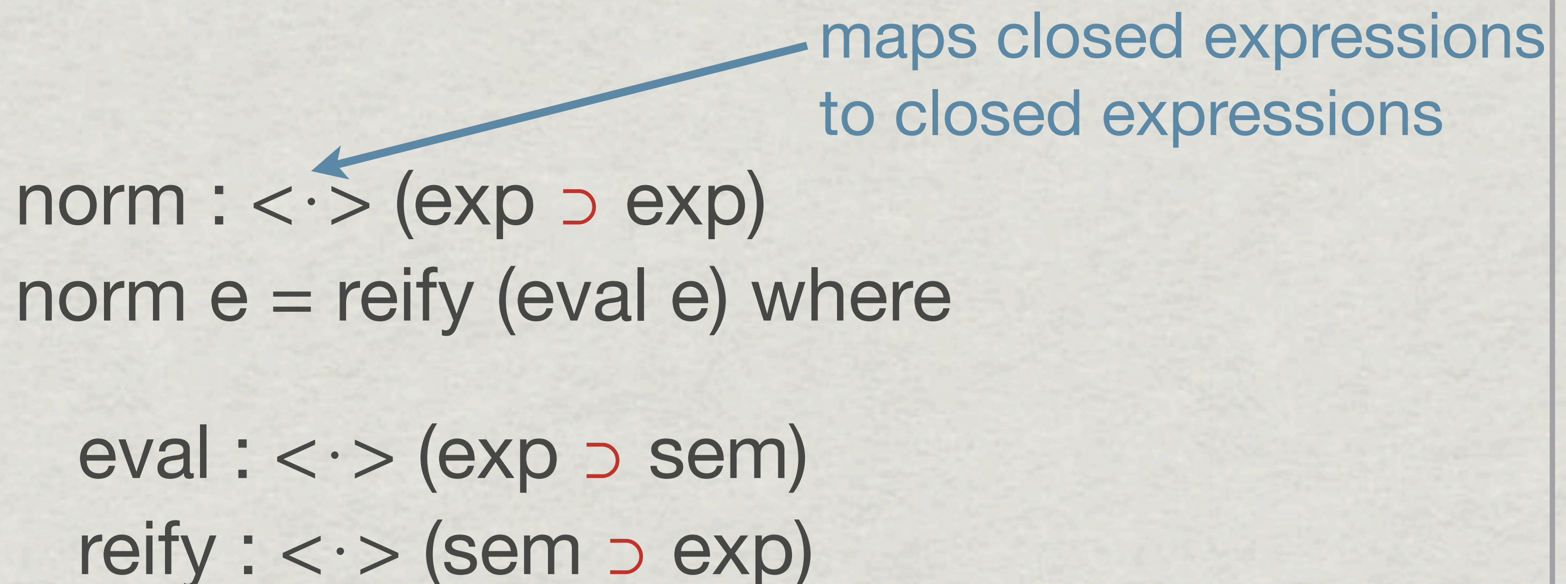
norm : <·> (exp ⊃ exp)
norm e = reify (eval e) where

eval : <·> (exp ⊃ sem)
reify : <·> (sem ⊃ exp)

16

Scope-correct NBE

Normalize syntactic λ-terms by interpreting them as computational functions ⊃ in the metalanguage



Semantics

Semantics

- First cut:

$$\text{sem} = \mu s.\ s \supset s$$

eval : <·> exp ⊃ sem

eval (app e1 e2) = (unroll (eval e1)) (eval e2)

Semantics

- First cut:

  sem = μs. s ⊃ s

  eval : <·> exp ⊃ sem
  eval (app e1 e2) = (unroll (eval e1)) (eval e2)

- But how do you write reify : <·> sem ⊃ exp ?

17

Semantics

- First cut:

  sem = μs. s ⊃ s

  eval : <·> exp ⊃ sem
  eval (app e1 e2) = (unroll (eval e1)) (eval e2)

- But how do you write reify : <·> sem ⊃ exp ?

  Requires a slightly different target type...

17

Semantics

Semantic S ::= slam φ | neut(R)
Neutral R ::= x | napp(R,S)

napp : neu ⇒ sem ⇒ neu

neut : neu ⇒ sem

slam : (sem ⊃ sem) ⇒ sem

Semantics

sem ⊃ sem

Semantic S ::= slam φ | neut(R)

Neutral R ::= x | napp(R,S)

napp : neu ⇒ sem ⇒ neu

neut : neu ⇒ sem

slam : (sem ⊃ sem) ⇒ sem

however, it's not enough that φ works in the current context Ψ

Semantics

Semantic S ::= slam φ | neut(R)
Neutral R ::= x | napp(R,S)

napp : neu ⇒ sem ⇒ neu

neut : neu ⇒ sem

slam : (∀ Ψ. Ψ ⇒ (sem ⊃ sem)) ⇒ sem

semantic function that anticipates
extensions of the context

Semantics

sem ⊃ sem

Semantic S ::= slam φ | neut(R)

Neutral R ::= x | napp(R,S)

napp : neu ⇒ sem ⇒ neu

neut : neu ⇒ sem

slam : (∀ Ψ. Ψ ⇒ (sem ⊃ sem)) ⇒ sem

semantic function that anticipates extensions of the context

eval: $\langle\cdot\rangle$ (exp $\supset$ sem)

20

$$\text{eval}: \langle\cdot\rangle \; \forall\, \Psi_e, \Psi_s. \;\; \text{env}\; \Psi_e\; \Psi_s \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$$

$\langle \Psi \rangle ([\Psi'] A) = \langle \Psi' \rangle A$

$$\text{eval} : \langle \cdot \rangle \; \forall \Psi_e, \Psi_s. \; \text{env}\ \Psi_e\ \Psi_s \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$$

$\langle\Psi\rangle\ ([\Psi']\ A) = \langle\Psi'\rangle\ A$

eval: $\langle\cdot\rangle\ \forall\ \Psi_e, \Psi_s.$

$\quad\quad \text{env}\ \Psi_e\ \Psi_s$

$\quad\quad \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

Environment type:

$$\text{env}\ \Psi_e\ \Psi_s = [\Psi_e](\text{exp}\#) \supset [\Psi_s]\text{sem}$$

$\langle\Psi\rangle$ $([\Psi'] A) = \langle\Psi'\rangle A$

eval: $\langle\cdot\rangle$ $\forall$ $\Psi_e, \Psi_s.$

$\text{env } \Psi_e \; \Psi_s$

$\supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

Environment type:

$$\text{env } \Psi_e \; \Psi_s = [\Psi_e](\text{exp\#}) \supset [\Psi_s]\text{sem}$$

$\langle\Psi\rangle$ $(D\#) = D \in \Psi$

```
eval: <·> ∀ Ψe, Ψs.
          env Ψe Ψs
        ⊃ [Ψe]exp ⊃ [Ψs]sem

eval σ (var x) = σ x
eval σ (app e1 e2) = appsem (eval σ e1) (eval σ e2)
eval σ (lam e) = ?
```

$\text{eval}: \langle\cdot\rangle\ \forall\ \Psi_e, \Psi_s.$

$\text{env}\ \Psi_e\ \Psi_s$

$\supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

$\text{eval}\{\Psi_e\}\{\Psi_s\}\ \sigma\ (\text{lam e}) = \text{slam}\ \varphi$

$\text{eval}: \langle\cdot\rangle\ \forall\ \Psi_e, \Psi_s.$

$\quad \text{env}\ \Psi_e\ \Psi_s$

$\quad \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

$\text{eval}\{\Psi_e\}\{\Psi_s\}\ \sigma\ (\text{lam}\ e) = \text{slam}\ \varphi$ where

$\varphi : \langle\Psi_s\rangle\ \forall\ \Psi_s'.\ \Psi_s' \Rightarrow (\text{sem} \supset \text{sem})$

$\varphi = ?$

eval: $\langle \cdot \rangle\ \forall\ \Psi_e, \Psi_s.$

$\quad\quad\quad\quad \text{env}\ \Psi_e\ \Psi_s$

$\quad\quad\quad \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

$\text{eval}\{\Psi_e\}\{\Psi_s\}\ \sigma\ (\text{lam e}) = \text{slam}\ \varphi$ where

$\varphi : \langle \Psi_s \rangle\ \forall\ \Psi_s'.\ \Psi_s' \Rightarrow (\text{sem} \supset \text{sem})$

$\varphi\ \Psi_s'\ s' = \text{eval}\{\Psi_e, \text{exp}\}\{\Psi_s, \Psi_s'\}\ \sigma'\ e$

25

$\text{eval}: \langle\cdot\rangle\ \forall\ \Psi_e, \Psi_s.$
$\quad \text{env}\ \Psi_e\ \Psi_s$
$\quad \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

$\text{eval}\{\Psi_e\}\{\Psi_s\}\ \sigma\ (\text{lam}\ e) = \text{slam}\ \varphi\ \text{where}$

$\varphi : \langle\Psi_s\rangle\ \forall\ \Psi_s'.\ \Psi_s' \Rightarrow (\text{sem} \supset \text{sem})$

$\varphi\ \Psi_s'\ s' = \text{eval}\{\Psi_e, \text{exp}\}\{\Psi_s, \Psi_s'\}\ \sigma'\ e\ \text{where}$

$\sigma' : ([\Psi_e, \text{exp}]\text{exp\#} \supset [\Psi_s, \Psi_s']\text{sem}$
$\sigma' = \text{extend}\ \sigma\ \text{with}\ s'$

$\text{eval}: \langle\cdot\rangle\ \forall\ \Psi_e, \Psi_s.$
$\quad\quad \text{env}\ \Psi_e\ \Psi_s$
$\quad\quad \supset [\Psi_e]\text{exp} \supset [\Psi_s]\text{sem}$

$\text{eval}\{\Psi_e\}\{\Psi_s\}\ \sigma\ (\text{lam e}) = \text{slam}\ \varphi\ \text{where}$

$\varphi : \langle\Psi_s\rangle\ \forall\ \Psi_s'.\ \Psi_s' \Rightarrow (\text{sem} \supset \text{sem})$

$\varphi\ \Psi_s'\ s' = \text{eval}\{\Psi_e, \text{exp}\}\{\Psi_s, \Psi_s'\}\ \sigma'\ e\ \text{where}$

$\sigma' : ([\Psi_e, \text{exp}]\text{exp\#} \supset [\Psi_s, \Psi_s']\text{sem}$
$\sigma'\ \text{i0} = s'$
$\sigma'\ (\text{iS}\ x) = \text{weaken}\ (\sigma\ x)\ \text{with}\ \Psi_s'$

σ' (iS x) = weaken (σ x) with Ψ_S'

has type $<\Psi_s>$sem

σ' (iS x) = weaken (σ x) with Ψ_s'

28

has type $<\Psi_s>sem$

$\sigma'\ (iS\ x) = weaken\ (\sigma\ x)\ with\ \Psi_s'$

$weaken : <\Psi_s>\ sem \supset (\Psi_s' \Rightarrow sem)$

has type $<\Psi_s>$sem

σ' (iS x) = weaken (σ x) with Ψ_s'

weaken : $<\Psi_s>$ sem $\supset$ ($\Psi_s' \Rightarrow$ sem)

In raw Agda, would need to implement for each object language…

This Paper

- Implement framework as a *universe* in Agda
- Examples and comparisons with Twelf/Delphin/Beluga/FreshML
- **Datatype-generic implementations of the structural properties**

Structural Properties

- Weakening: $\forall \Psi.\ \Psi \Rightarrow (A \supset (D \Rightarrow A))$
- Substitution: $\forall \Psi.\ \Psi \Rightarrow (D \Rightarrow A) \supset (D \supset A)$
- Exchange: $\forall \Psi.\ \Psi \Rightarrow (D_1 \Rightarrow D_2 \Rightarrow A) \supset (D_2 \Rightarrow D_1 \Rightarrow A)$
- Contraction: $\forall \Psi.\ \Psi \Rightarrow (D \Rightarrow D \Rightarrow A) \supset (D \Rightarrow A)$
- Strengthening: $\forall \Psi.\ \Psi \Rightarrow (D \Rightarrow A) \supset A$

Structural Properties

In *mixed*, *pronominal* setting,
structural properties do not hold at all types:

Computational functions express
side conditions on the current context

$$\frac{(\Psi \vdash J) \supset 0}{\Psi \vdash J'}$$

Structural Properties

In *mixed*, *pronominal* setting,
structural properties do not hold at all types:

Computational functions express
side conditions on the current context

$$\frac{(\Psi \vdash J) \supset 0}{\Psi \vdash J'}$$

"it is not the case that $\Psi \vdash J$"

Proof **cannot** be weakened
with J because $\Psi, J \vdash J$

Structural Properties

In *mixed*, *pronominal* setting,
structural properties do not hold at all types:

Computational functions express
side conditions on the current context

$$\frac{(\Psi \vdash J) \supset 0}{\Psi \vdash J'}$$

"it is not the case that $\Psi \vdash J$"

Proof **cannot** be weakened
with J because $\Psi, J \vdash J$

But there are sufficient conditions under which they do hold…

Structural Properties

Instances of weakening:

Structural Properties

Instances of weakening:

- exp $\supset$ (exp $\Rightarrow$ exp)

Structural Properties

Instances of weakening:

- exp $\supset$ (exp $\Rightarrow$ exp) **OK**

32

Structural Properties

Instances of weakening:

- $\text{exp} \supset (\text{exp} \Rightarrow \text{exp})$ **OK**
- $(\text{exp} \Rightarrow \text{exp}) \supset (\text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}))$

Structural Properties

Instances of weakening:

- $\mathrm{exp} \supset (\mathrm{exp} \Rightarrow \mathrm{exp})$ **OK**
- $(\mathrm{exp} \Rightarrow \mathrm{exp}) \supset (\mathrm{exp} \Rightarrow (\mathrm{exp} \Rightarrow \mathrm{exp}))$ **OK**

Structural Properties

Instances of weakening:

- exp $\supset$ (exp $\Rightarrow$ exp) **OK**
- (exp $\Rightarrow$ exp) $\supset$ (exp $\Rightarrow$ (exp $\Rightarrow$ exp)) **OK**
- (exp $\supset$ exp) $\supset$ (exp $\Rightarrow$ (exp $\supset$ exp))

Structural Properties

Instances of weakening:

- $\text{exp} \supset (\text{exp} \Rightarrow \text{exp})$ **OK**
- $(\text{exp} \Rightarrow \text{exp}) \supset (\text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}))$ **OK**
- $(\text{exp} \supset \text{exp}) \supset (\text{exp} \Rightarrow (\text{exp} \supset \text{exp}))$ **?**

Structural Properties

Weakening: (exp ⊃ exp) ⊃ (exp ⇒ (exp ⊃ exp))

Given, for example,

normalize : <·> (exp ⊃ exp)
normalize (lam x.e) = …
normalize (app e1 e2) = …
normalize (var x) = impossible

33

Structural Properties

Weakening: (exp ⊃ exp) ⊃ (exp ⇒ (exp ⊃ exp))

Must create

normalize’ : <·> (exp ⇒ (exp ⊃ exp))

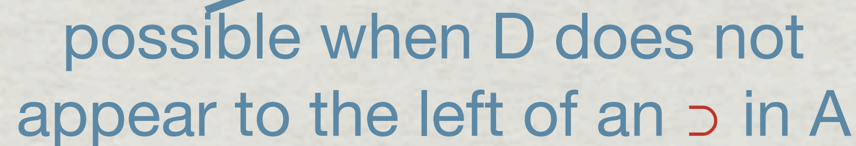
normalize’ y.(lam x.e) = …
normalize’ y.(app e1 e2) = …
normalize’ y.(var y) = ???

34

Structural Properties

- Structural properties are not built-in (unlike LF): OK that they only hold at certain types
- Implemented datatype-generically but conditionally: “free” when conditions hold
- Conditions discharged automatically

weaken: $A \supset (D \Rightarrow A)$ if canWkn(D,A)



Conclusion

- Implemented a framework for mixing binding and computation in a pronominal setting
- Datatype-generic implementations of the structural properties by recursion over universe
- See paper for more examples and comparisons with Twelf/Delphin/Beluga/FreshML

Future Work

- Positivity check for signature + termination
- Equational behavior of structural properties
- Named syntax for variables;
  More-convenient syntax for structural properties
- Derived induction principles for structural props
- Dependent types!

Thanks for listening!