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) , ∀x:τ.A , hypothetical judgements

3

Binding

Represent bound variables:

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

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

app : exp ⇒ exp ⇒ exp

lam : (exp ⇒ exp) ⇒ exp

Binding

Represent bound variables:

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

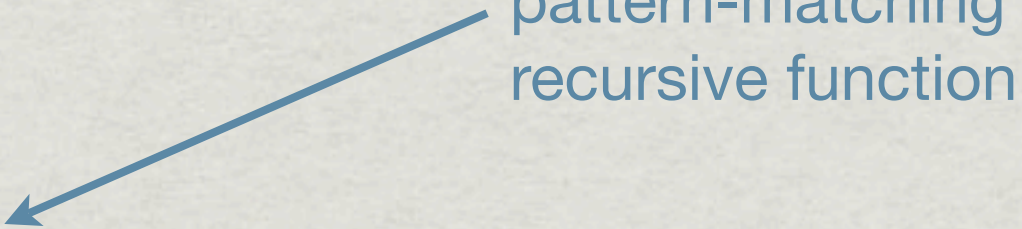
E.g. type exp representing syntax of λ-terms:

app : exp ⇒ exp ⇒ exp

lam : (exp ⇒ exp) ⇒ exp

weak function space representing binding:
means “an exp in the presence of a new exp”

Computation




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

Our Approach

Our Approach

1. Makes an *a priori* type distinction between
   ⇒ (binding) and ⊃ (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]

5

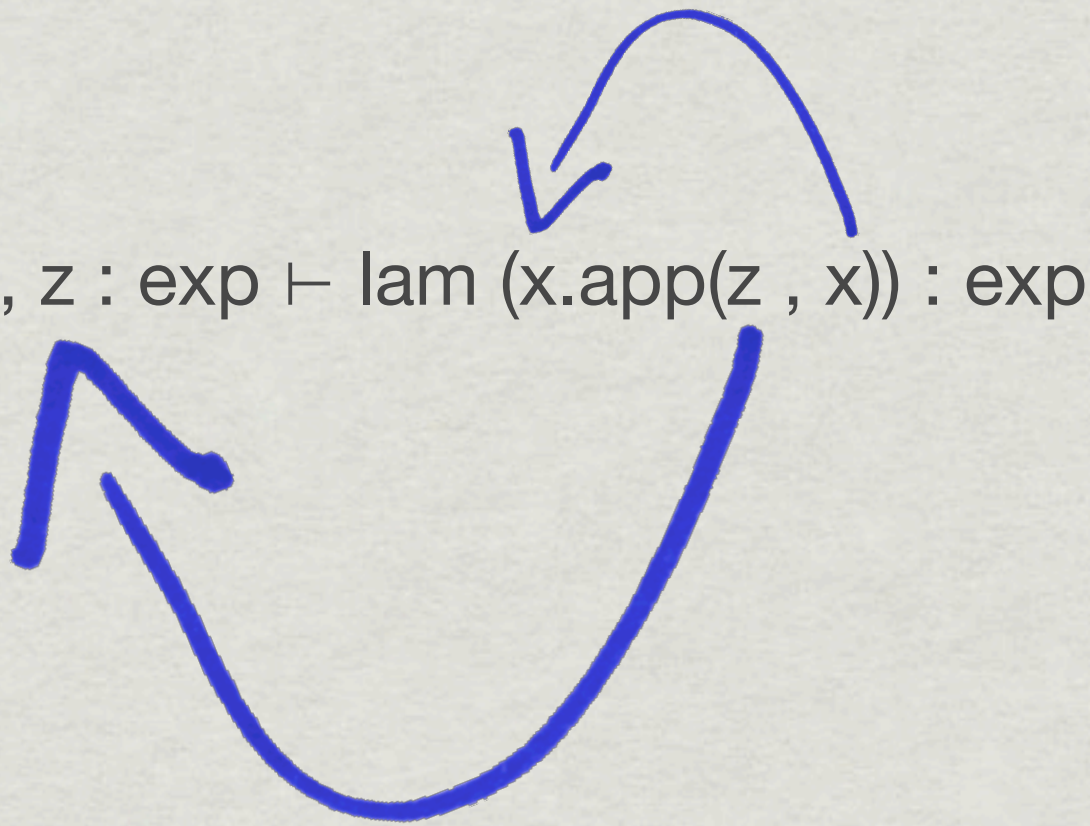
Our Approach

1. Makes an *a priori* type distinction between
   ⇒ (binding) and ⊃ (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]

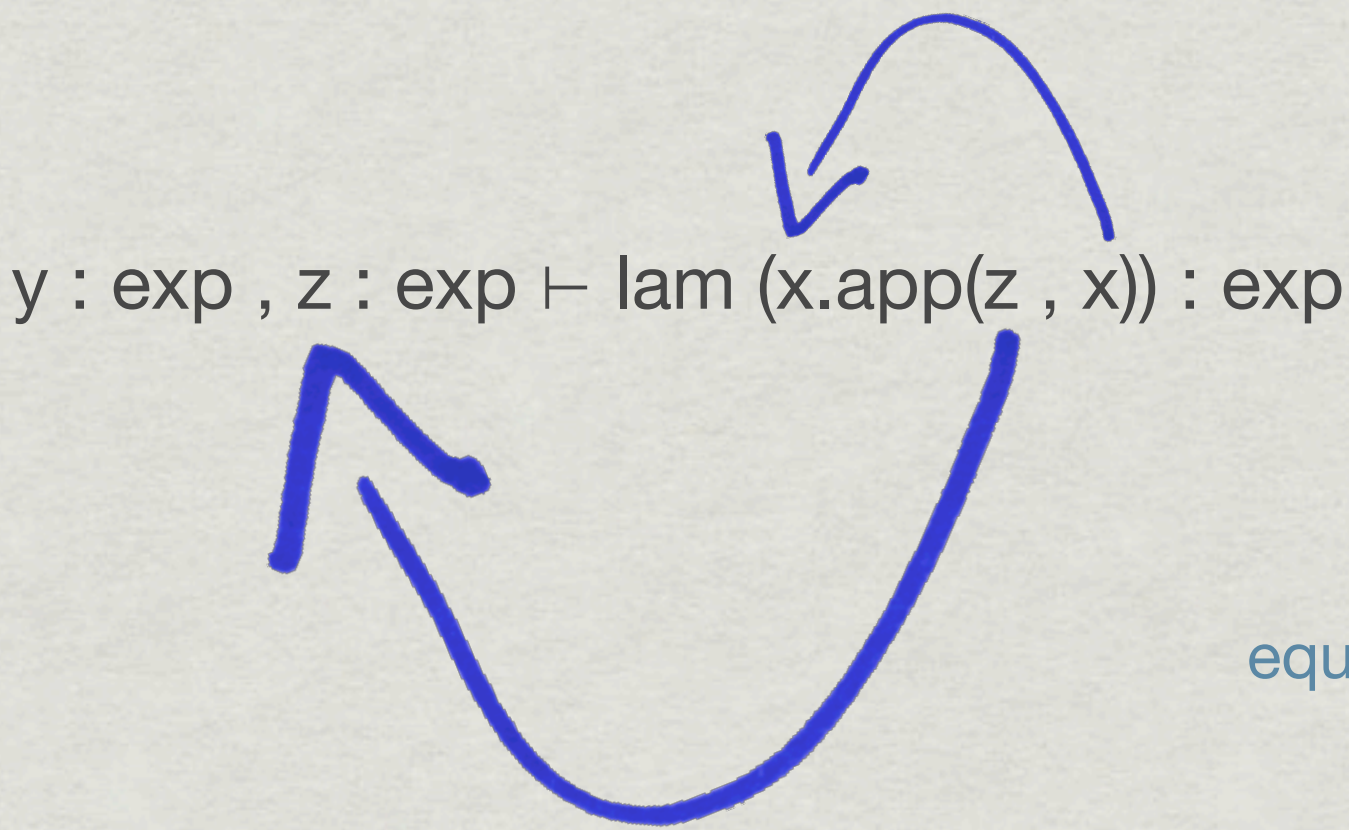
Pronominal Variables

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. $< x_1 : \text{exp} \ \ldots \ x_n : \text{exp} > \text{exp}$

free vars in $x_1 \ \ldots \ x_n$

Permit precise types for computations:

normalize : $<\cdot>\text{exp} \supset <\cdot>\text{exp}$

normalize/open : $\forall \Psi.\ <\Psi>\text{exp} \supset <\Psi>\text{exp}$

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

8

This Paper

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

9

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

10

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

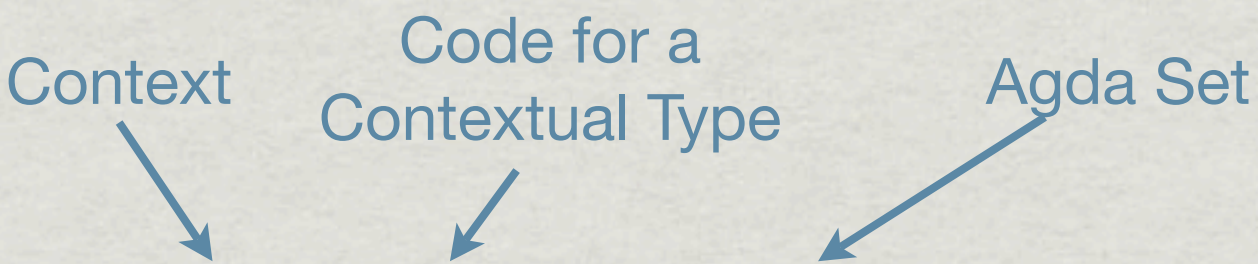
  data Code $A ::= A\ \mathsf{list} \mid A \supset B \mid D \mid \Psi \Rightarrow A \mid \forall\, \Psi.\, A$

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

  $\langle \_ \rangle \_ : \mathsf{Ctx} \to \mathsf{Code} \to \mathsf{Set}$

11

Interpretation



$$\langle \Psi \rangle\ (A\ \mathit{list}) = \mathsf{List}\ \langle \Psi \rangle A$$

$$\langle \Psi \rangle\ (A \supset B) = \langle \Psi \rangle A \to \langle \Psi \rangle B$$

$$\langle \Psi \rangle\ (\Psi' \Rightarrow A) = \langle \Psi, \Psi' \rangle A$$

$$\langle \Psi \rangle\ (\forall\ \Psi'.\ A) = (\Psi' : \mathsf{Ctx}) \to \langle \Psi \rangle (A\ \Psi')$$

$$\langle \Psi \rangle\ D = \ldots$$

12

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')
```


14

This Paper

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

15

Scope-correct NBE

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

16

Scope-correct NBE

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

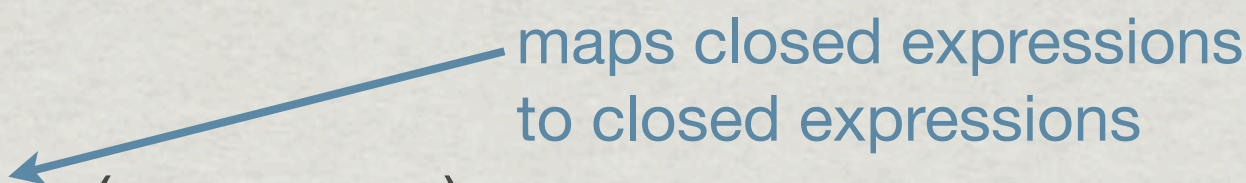
$$\mathrm{norm} : \langle\cdot\rangle\ (\mathrm{exp} \supset \mathrm{exp})$$
$$\mathrm{norm}\ e = \mathrm{reify}\ (\mathrm{eval}\ e)\ \text{where}$$

$$\mathrm{eval} : \langle\cdot\rangle\ (\mathrm{exp} \supset \mathrm{sem})$$
$$\mathrm{reify} : \langle\cdot\rangle\ (\mathrm{sem} \supset \mathrm{exp})$$

16

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

Semantics

Semantics

- First cut:

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

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

17

Semantics

- First cut:

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

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

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

- But how do you write reify : $\langle \cdot \rangle$ sem $\supset$ exp ?

17

Semantics

- First cut:

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

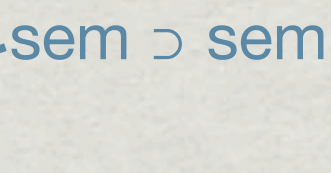
eval : <·> exp ⊃ sem
eval (app e1 e2) = (unroll (eval e1)) (eval e2)

- 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

18

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

19

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

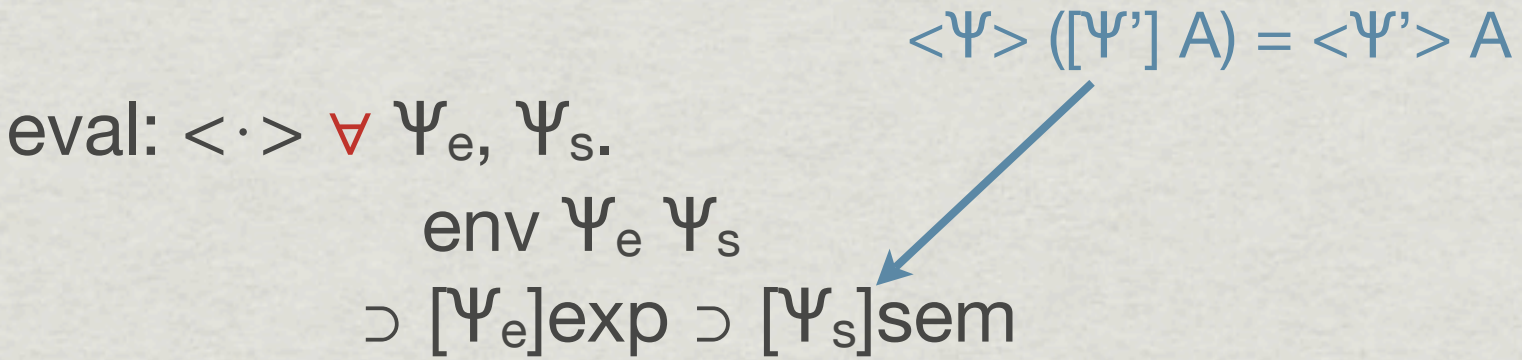
semantic function that anticipates extensions of the context

19

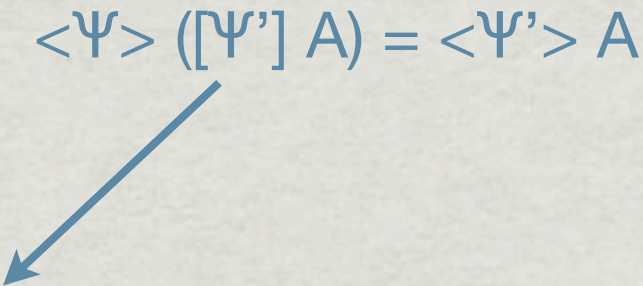
eval: $\langle \cdot \rangle$ (exp $\supset$ sem)

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

21



21



eval: <·> ∀ Ψ_e, Ψ_s.

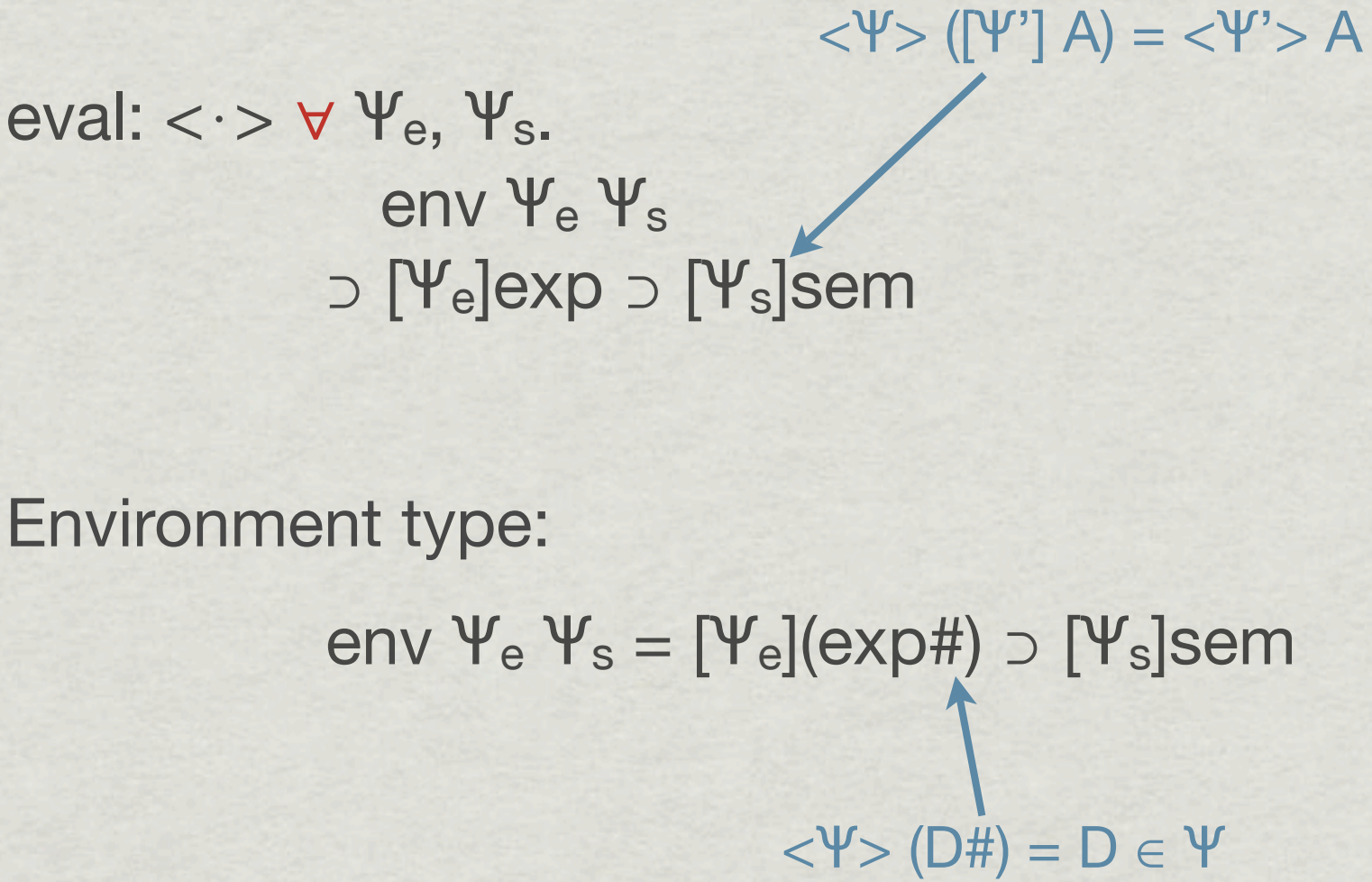
env Ψ_e Ψ_s

⊃ [Ψ_e]exp ⊃ [Ψ_s]sem

Environment type:

env Ψ_e Ψ_s = [Ψ_e](exp#) ⊃ [Ψ_s]sem

21



Environment type:

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

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

21

eval: $\langle\cdot\rangle$ $\forall$ Ψ_e, Ψ_s.

env Ψ_e Ψ_s

$\supset$ [Ψ_e]exp $\supset$ [Ψ_s]sem

eval σ (var x) = σ x

eval σ (app e1 e2) = appsem (eval σ e1) (eval σ e2)

eval σ (lam e) = ?

22

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

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

eval: $<\cdot>$ $\forall$ Ψ_e, Ψ_s.

env Ψ_e Ψ_s

$\supset$ [Ψ_e]exp $\supset$ [Ψ_s]sem

eval{Ψ_e}{Ψ_s} σ (lam e) = slam φ where

φ : <Ψ_s> $\forall$ Ψ_s'. Ψ_s' $\Rightarrow$ (sem $\supset$ sem)

φ = ?

24

eval: <·> ∀ Ψ_e, Ψ_s.

env Ψ_e Ψ_s

⊃ [Ψ_e]exp ⊃ [Ψ_s]sem

eval{Ψ_e}{Ψ_s} σ (lam e) = slam φ where

φ : <Ψ_s> ∀ Ψ_s'. Ψ_s' ⇒ (sem ⊃ sem)

φ Ψ_s' s' = eval{Ψ_e, exp}{Ψ_s,Ψ_s'} σ' e

25

eval: $<\cdot>$ $\forall$ Ψ_e, Ψ_s.

env Ψ_e Ψ_s

$\supset$ [Ψ_e]exp $\supset$ [Ψ_s]sem

eval{Ψ_e}{Ψ_s} σ (lam e) = slam φ where

φ : <Ψ_s> $\forall$ Ψ_s'. Ψ_s' $\Rightarrow$ (sem $\supset$ sem)

φ Ψ_s' s' = eval{Ψ_e, exp}{Ψ_s,Ψ_s'} σ' e where

σ' : ([Ψ_e, exp]exp# $\supset$ [Ψ_s,Ψ_s']sem

σ' = extend σ with s'

26

eval: $\langle\cdot\rangle$ $\forall$ Ψ_e, Ψ_s.

env Ψ_e Ψ_s

$\supset$ [Ψ_e]exp $\supset$ [Ψ_s]sem

eval{Ψ_e}{Ψ_s} σ (lam e) = slam φ where

φ : <Ψ_s> $\forall$ Ψ_s'. Ψ_s' $\Rightarrow$ (sem $\supset$ sem)

φ Ψ_s' s' = eval{Ψ_e, exp}{Ψ_s,Ψ_s'} σ' e where

σ' : ([Ψ_e, exp]exp# $\supset$ [Ψ_s,Ψ_s']sem

σ' i0 = s'

σ' (iS x) = weaken (σ x) with Ψ_s'

27

σ’ (iS x) = weaken (σ x) with Ψ_s’

has type $\langle \Psi_s \rangle_{sem}$

σ' (iS x) = weaken (σ x) with Ψ_s'

28

has type $\langle \Psi_s \rangle$sem

σ' (iS x) = weaken (σ x) with Ψ_s'

weaken : $\langle \Psi_s \rangle$ sem $\supset$ ($\Psi_s' \Rightarrow$ sem)

28

has type $\langle \Psi_s \rangle$sem

σ' (iS x) = weaken (σ x) with Ψ_s'

weaken : $\langle \Psi_s \rangle$ sem $\supset$ ($\Psi_s' \Rightarrow$ sem)

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

28

This Paper

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

29

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: $\forall \Psi.\ \Psi \Rightarrow (D \Rightarrow A) \supset A$

30

Structural Properties

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

Computational functions express
side conditions on the current context

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

31

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 $\Psi, J \vdash J$

31

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…

31

Structural Properties

Instances of weakening:

32

Structural Properties

Instances of weakening:

- exp ⊃ (exp ⇒ exp)

32

Structural Properties

Instances of weakening:

- exp ⊃ (exp ⇒ exp) **OK**

32

Structural Properties

Instances of weakening:

- exp ⊃ (exp ⇒ exp) **OK**
- (exp ⇒ exp) ⊃ (exp ⇒ (exp ⇒ exp))

32

Structural Properties

Instances of weakening:

- exp ⊃ (exp ⇒ exp) **OK**
- (exp ⇒ exp) ⊃ (exp ⇒ (exp ⇒ exp)) **OK**

32

Structural Properties

Instances of weakening:

- exp ⊃ (exp ⇒ exp) **OK**
- (exp ⇒ exp) ⊃ (exp ⇒ (exp ⇒ exp)) **OK**
- (exp ⊃ exp) ⊃ (exp ⇒ (exp ⊃ exp))

32

Structural Properties

Instances of weakening:

- exp ⊃ (exp ⇒ exp) **OK**
- (exp ⇒ exp) ⊃ (exp ⇒ (exp ⇒ exp)) **OK**
- (exp ⊃ exp) ⊃ (exp ⇒ (exp ⊃ exp)) **?**

32

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))

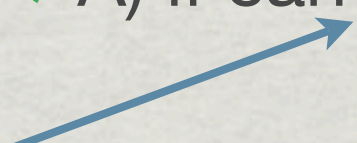
normalize' y.(lam x.e) = …
normalize' y.(app e1 e2) = …
normalize' y.(var y) = ???

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)



possible when D does not appear to the left of an $\supset$ in A

35

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

36

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!

37

Thanks for listening!