

Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions

Kathrin Stark, Steven Schäfer, Jonas Kaiser



CPP 2019
January 15

Our Motivation: The Formalisation of Metatheory

- Of programming languages and logical systems with binders,
 - ▶ e.g. Call-By-Push-Value (CBPV):

$$\begin{array}{ll} V \in vl ::= x \mid () \mid (V_1, V_2) \mid \text{inj}_i V \mid \{M\} & \text{values} \\ M \in tm ::= M V \mid \lambda x. M & \\ & \mid \text{let } x \leftarrow M_1 \text{ in } M_2 \\ & \mid \text{split}(V, x_1.x_2.M) \\ & \mid \text{case}(V, x_1.M_1, x_2.M_2) \mid V! \dots & \text{computations} \end{array}$$

- Formalising proofs as
 - ▶ weak and strong normalisation
 - ▶ CBV and CBN can be simulated in CBPV

Our Motivation: The Formalisation of Metatheory

- Of programming languages and logical systems **with binders**,
 - ▶ e.g. Call-By-Push-Value (CBPV):

$$\begin{aligned} V \in vl &::= x \mid () \mid (V_1, V_2) \mid \text{inj}_i V \mid \{M\} && \text{values} \\ M \in tm &::= M V \mid \lambda x. M \\ &\mid \text{let } x \leftarrow M_1 \text{ in } M_2 \\ &\mid \text{split}(V, x_1.x_2.M) \\ &\mid \text{case}(V, x_1.M_1, x_2.M_2) \mid V! \dots && \text{computations} \end{aligned}$$

- Formalising proofs as
 - ▶ weak and strong normalisation
 - ▶ CBV and CBN can be simulated in CBPV

Our Motivation: The Formalisation of Metatheory

- Of programming languages and logical systems with binders,
 - ▶ e.g. Call-By-Push-Value (CBPV):

$$\begin{aligned} V \in vl &::= x \mid () \mid (V_1, V_2) \mid \text{inj}_i V \mid \{M\} && \text{values} \\ M \in tm &::= M V \mid \lambda x. M \\ &\mid \text{let } x \leftarrow M_1 \text{ in } M_2 \\ &\mid \text{split}(V, x_1.x_2.M) \\ &\mid \text{case}(V, x_1.M_1, x_2.M_2) \mid V! \dots && \text{computations} \end{aligned}$$

- Formalising proofs as
 - ▶ weak and strong normalisation
 - ▶ CBV and CBN can be simulated in CBPV

Binders Come With Boilerplate – Strong Normalization for CBPV

(values) $V, W ::= x \mid () \mid (V_1, V_2) \mid \text{inj}_i V \mid \{M\}$
(computations) $M, N ::= \text{split}(V, x_1.x_2.M) \mid \text{case}_0(V) \mid \langle \rangle$
 $\mid \text{case}(V, x_1.M_1, x_2.M_2) \mid V!$
 $\mid \text{return } V \mid \text{let } x \leftarrow M \text{ in } N$
 $\mid \lambda x.M \mid M V$
 $\mid \langle M_1, M_2 \rangle \mid \text{prj}_i M$



Expressions

Figure 1. CBPV syntax

Binders Come With Boilerplate – Strong Normalization for CBPV



Expressions

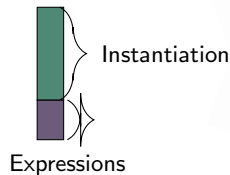
Value typing

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$
$$\frac{}{\Gamma \vdash () : 1}$$
$$\frac{\Gamma \vdash V : A_i}{\Gamma \vdash \text{inj}_i V : A_1 + A_2}$$
$$\frac{\Gamma \vdash V_1 : A_1 \quad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

Computation typing

$$\frac{\Gamma \vdash V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash M : C}{\Gamma \vdash \text{split}(V, x_1.x_2.M) : C}$$
$$\frac{\Gamma \vdash V : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash M_1 : C}{\Gamma \vdash \text{case}(V, x_1.M_1) : C}$$
$$\frac{\Gamma \vdash M : C}{\Gamma \vdash \{M\} : UC}$$
$$\frac{\Gamma \vdash M : C}{\Gamma \vdash \text{inj}_i V \mid \{M\} : \text{eo}(V) \mid \langle \rangle}$$
$$\frac{\Gamma \vdash V : 0}{\Gamma \vdash \text{case}(V, _)$$

Binders Come With Boilerplate – Strong Normalization for CBPV



$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

Value typing

$M > M'$

Primitive reduction

$\text{split}((V_1, V_2), x_1.x_2.M) > M[V_1/x_1, V_2/x_2]$

$\text{case}(\text{inj}_i V, x_1.M_1, x_2.M_2) > M_i[V/x_i]$

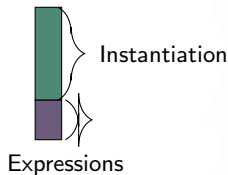
$\{M\}! > M$

$\text{let } x \leftarrow \text{return } V \text{ in } M > M[V/x]$

$(\lambda x.M) V > M[V/x]$

$\langle M_1, M_2 \rangle > M_i$

Binders Come With Boilerplate – Strong Normalization for CBPV

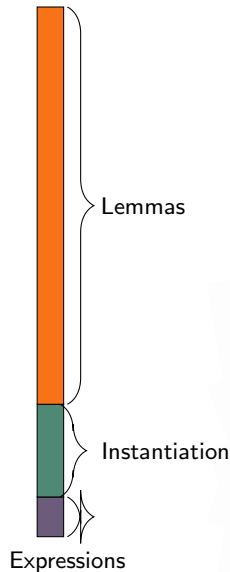


$$C[A \rightarrow C] := \{\lambda x.M \mid \forall V \in \mathcal{V}[A]. M[V/x] \in \mathcal{E}[C]\}$$
$$C[C_1 \& C_2] := \{(M_1, M_2) \mid M_1 \in \mathcal{E}[C_1], M_2 \in \mathcal{E}[C_2]\}$$

Semantic Typing

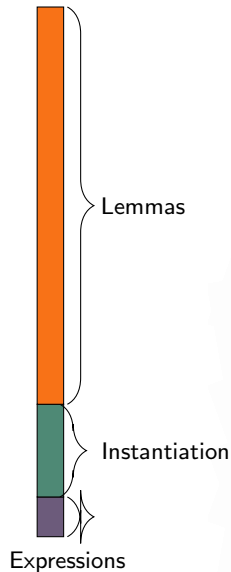
$$\mathcal{E}[C] := \{M \mid \exists N. M \Downarrow N \wedge N \in C[C]\}$$
$$\mathcal{G}[\Gamma] := \{\gamma \mid \forall (x : A) \in \Gamma, \gamma x \in \mathcal{V}[A]\}$$
$$\Gamma \models V : A := \forall \gamma \in \mathcal{G}[\Gamma]. V[\gamma] \in \mathcal{V}[A]$$
$$\Gamma \models M : C := \forall \gamma \in \mathcal{G}[\Gamma]. M[\gamma] \in \mathcal{E}[C]$$

Binders Come With Boilerplate – Strong Normalization for CBPV



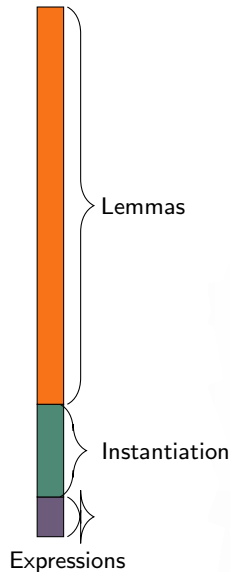
$C[A \rightarrow C] := \{\lambda x.M \mid \forall V \in \mathcal{V}[A]. M[V/x] \in \mathcal{E}[C]\}$
 $C[C_1 \& C_2] := \{(M_1, M_2) \mid M_1 \in \mathcal{E}[C_1], M_2 \in \mathcal{E}[C_2]\}$
Semantic
 $\mathcal{E}[C] := \{M \mid \dots\}$
Lemma 4.1. The following hold:
 1. $C[C] \subseteq \mathcal{E}[C]$.
 2. $C[C]$ only contains normal forms w.r.t. $\sim$.
 If $M \sim^* N$ and $N \in \mathcal{E}[C]$, then $M \in \mathcal{E}[C]$.
 If $M \sim^* N$ and $N \in \mathcal{V}[A]$, then $M[V/x] \in \mathcal{E}[C]$.

Binders Come With Boilerplate – Strong Normalization for CBPV



$C[A \rightarrow C] := \{\lambda x.M \mid \forall V \in \mathcal{V}[A]. M[V/x] \in \mathcal{E}[C]\}$
 $C[C_1 \& C_2] := \{(M_1, M_2) \mid M_1 \in \mathcal{E}[C_1], M_2 \in \mathcal{E}[C_2]\}$
Semantic
 if $\Gamma \vdash V : A$, then $\Gamma \vDash M : C$, then $\Gamma \vDash M : C$ and $M \in \mathcal{E}[C]$.
Theorem 4.2. If $\Gamma \vdash M : C$, then $\Gamma \vDash M : C$ holds:
Lemma 4.1.
 1. $C[C] \subseteq \mathcal{E}[C]$
 2. $C[C]$ only contains terms w.r.t. $\sim$.
 If $M \sim^* N$ and $N \in \mathcal{E}[C]$, then $M \in \mathcal{E}[C]$.

Binders Come With Boilerplate – Strong Normalization for CBPV



$$C[A \rightarrow C] := \{\lambda x.M \mid \forall V \in \mathcal{V}[A]. M[V] \in \mathcal{E}[C]\}$$

$$C[C_1 \& C_2] := \{(M_1, M_2) \mid M_1 \in \mathcal{E}[C_1], M_2 \in \mathcal{E}[C_2]\}$$

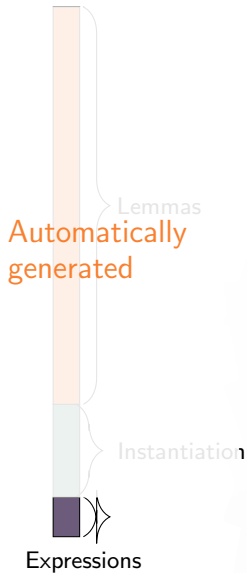
Semantic

$$c[\text{var } 0, \gamma \circ \langle \uparrow \rangle][v, \text{ids}] = c[v, \gamma]$$

Theorem 4.2 (Strong Normalization):
if $\Gamma \vdash V : A$, then $\Gamma \models M : C$, then $\Gamma \models M$ holds:
forms w.r.t. $\sim$.
 $M \in \mathcal{E}[C]$.

Lemma 4.1.
1. $C[C] \subseteq \mathcal{E}[C]$
2. $C[C]$ only contains terms that are strongly normalizing.
If $M \sim^* N$ and $N \in \mathcal{E}[C]$, then $M \in \mathcal{E}[C]$.

Binders Come With Boilerplate – Strong Normalization for CBPV


$$C[A \rightarrow C] := \{\lambda x.M \mid \forall V \in \mathcal{V}[A]. M[V] \in \mathcal{E}[C]\}$$

$c[\text{var } 0, \gamma \circ \langle \uparrow \rangle][v, \text{ids}] = c[v, \gamma]?$

Related Work: Various Ways to Represent Binders

- parallel de Bruijn [de Bruijn '72]
- single-point de Bruijn
- nominal logic [Pitts '01]
- HOAS [Pfenning et al. '88]
- locally nameless [Aydemir et al. '08]
- contextual modalTT [Nanevski et al. '08]
- . . .

Related Work: Various Ways to Represent Binders

- parallel de Bruijn [de Bruijn '72]
- single-point de Bruijn
- nominal logic [Pitts '01]
- HOAS [Pfenning et al. '88]
- locally nameless [Aydemir et al. '08]
- contextual modalTT [Nanevski et al. '08]
- . . .

Our requirements:

- Works in a general-purpose proof assistant/Coq
- Little set-up
- Little overhead in the proofs

Related Work: Various Ways to Represent Binders

- parallel de Bruijn [de Bruijn '72]
- single-point de Bruijn
- nominal logic [Pitts '01]
- HOAS [Pfenning et al. '88]
- locally nameless [Aydemir et al. '08]
- contextual modalTT [Nanevski et al. '08]
- ...

Our requirements:

- Works in a general-purpose proof assistant/Coq
- Little set-up
- Little overhead in the proofs

Extension of Autosubst 1 [Schäfer/Tebbi/Smolka '15]

The Autosubst Workflow

Signature file, e.g.

```
app : term → term → term  
lam : (term → term) → term
```

⇓ Autosubst 2 compiler

Output file

Instantiation `[-]`
+ standard operations
+ automation

The Autosubst Workflow

Signature file, e.g.

```
app : term → term → term  
lam : (term → term) → term
```



Second-Order HOAS signature,
potentially multivariate and mutual inductive

⇓ Autosubst 2 compiler

Output file

Instantiation `[-]`

+ standard operations

+ automation

The Autosubst Workflow

Signature file, e.g.

```
app : term → term → term  
lam : (term → term) → term
```

⇓ Autosubst 2 compiler

} Second-Order HOAS signature,
potentially multivariate and mutual inductive

} Multi-layer compiler

Output file

Instantiation `[-]`
+ standard operations
+ automation

The Autosubst Workflow

Signature file, e.g.

`app : term → term → term`
`lam : (term → term) → term`



Autosubst 2 compiler

Output file

Instantiation `[-]`
+ standard operations
+ automation

} Second-Order HOAS signature,
potentially multivariate and mutual inductive

} Multi-layer compiler

parallel de Bruijn [de Bruijn '72]

- renamings + substitutions
- unscoped or well-scoped

+ σ -calculus [Abadi et al. '91]:
solve equations of the form $s = t$

The Autosubst Workflow

Signature file, e.g.

`app : term → term → term`
`lam : (term → term) → term`

⇓ Autosubst 2 compiler

Output file

Instantiation `[-]`
+ standard operations
+ automation

} Second-Order HOAS signature,
potentially multivariate and mutual inductive

} Multi-layer compiler

parallel de Bruijn [de Bruijn '72]

- renamings + substitutions
- unscoped or well-scoped

+ σ -calculus [Abadi et al. '91]:
solve equations of the form $s = t$

Tool / case studies : www.ps.uni-saarland.de/extras/autosubst2

The Autosubst Workflow – Autosubst 1 vs. Autosubst 2

Signature file, e.g.

```
app : term → term → term  
lam : (term → term) → term
```

⇓ Autosubst 2 compiler

Output file

Instantiation $[-]$
+ standard operations
+ automation

} Second-Order HOAS signature,

} potentially multivariate and mutual inductive

} Multi-layer compiler

} parallel de Bruijn [de Bruijn '72]

■ renamings + substitutions

■ unscoped or well-scoped

+ σ -calculus [Abadi et al. '91]:

solve equations of the form $s = t$

Tool / case studies : www.ps.uni-saarland.de/extras/autosubst2

Organisation of the Talk

Signature file, e.g.

```
app : term → term → term  
lam : (term → term) → term
```

⇓ Autosubst 2 compiler

Output file

Instantiation $[-]$
+ standard operations
+ automation

Second-Order HOAS signature,

potentially multivariate and mutual inductive ④

Multi-layer compiler ②

parallel de Bruijn [de Bruijn '72]

■ renamings + substitutions

■ unscoped or well-scoped ③

+ σ -calculus [Abadi et al. '91]:

solve equations of the form $s = t$

Tool ①/ case studies ⑤: www.ps.uni-saarland.de/extras/autosubst2

Organisation of the Talk

Signature file, e.g.

$\text{app} : \text{term} \rightarrow \text{term} \rightarrow \text{term}$
 $\text{lam} : (\text{term} \rightarrow \text{term}) \rightarrow \text{term}$

$\Downarrow$ Autosubst 2 compiler

Output file

Instantiation $[-]$
+ standard operations
+ automation

} Second-Order HOAS signature,
potentially multivariate and mutual inductive ④

} Multi-layer compiler ②

parallel de Bruijn [de Bruijn '72]

- renamings + substitutions
- unscoped or well-scoped ③

+ σ -calculus [Abadi et al. '91]:
solve equations of the form $s = t$

Tool ① / case studies ⑤: www.ps.uni-saarland.de/extras/autosubst2

Revisiting Strong Normalization for CBPV

Revisiting Strong Normalization for CBPV



```
Open ▾  cbpv.sig      
~/Documents/Git Repos Redmine/autosubst-2-ref...t-2/as2-with-functors/examples/coq/scoped-cbpv  
  
valtype : Type  
comptype : Type  
value : Type  
comp : Type  
bool : Type  
  
zero: valtype  
one: valtype  
U: comptype -> valtype  
Sigma: valtype -> valtype -> valtype  
cross: valtype -> valtype -> valtype  
  
cone: comptype  
F: valtype -> comptype  
Pi: comptype -> comptype -> comptype  
arrow: valtype -> comptype -> comptype  
  
u: value  
pair: value -> value -> value  
inj: bool -> value -> value
```

Revisiting Strong Normalization for CBPV

```
emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command ProofTree Interrupt Restart Help

(eq_trans) ((eq_trans) (eq_refl) ((ap) ( $\lambda x \rightarrow \text{caseP (nvalue) } x \text{ } (\_)$ ) H1)) ((ap) ( $\lambda x \rightarrow \text{caseP (nvalue) } (\_) x$ ) H2).

Definition upRen_value_value { m : N } { n : N } ( $\xi$  : (fn) (m)  $\rightarrow$  (fn) (n)) : _ :=
  (up_ren)  $\xi$ .

Fixpoint ren_value { nvalue : N } { nvalue : N } (xvalue : (fn) (nvalue)  $\rightarrow$  (fn) (nvalue)) (s : value (nvalue)) : _ :=
  match s with
  | var_value ( $\_$ ) s  $\rightarrow$  (var_value (nvalue)) (xvalue s)
  | u ( $\_$ )  $\rightarrow$  u (nvalue)
  | pair ( $\_$ ) s0 s1  $\rightarrow$  pair (nvalue) ((ren_value xvalue) s0) ((ren_value xvalue) s1)
  | inj ( $\_$ ) s0 s1  $\rightarrow$  inj (nvalue) (( $\lambda x \rightarrow x$ ) s0) ((ren_value xvalue) s1)
  | thunk ( $\_$ ) s0  $\rightarrow$  thunk (nvalue) ((ren_comp xvalue) s0)
  end
with ren_comp { nvalue : N } { nvalue : N } (xvalue : (fn) (nvalue)  $\rightarrow$  (fn) (nvalue)) (s : comp (nvalue)) : _ :=
  match s with
  | cu ( $\_$ )  $\rightarrow$  cu (nvalue)
  | force ( $\_$ ) s0  $\rightarrow$  force (nvalue) ((ren_value xvalue) s0)
  |  $\lambda$  ( $\_$ ) s0  $\rightarrow$   $\lambda$  (nvalue) ((ren_comp (upRen_value_value xvalue)) s0)
  | app ( $\_$ ) s0 s1  $\rightarrow$  app (nvalue) ((ren_comp xvalue) s0) ((ren_value xvalue) s1)
  | tuple ( $\_$ ) s0 s1  $\rightarrow$  tuple (nvalue) ((ren_comp xvalue) s0) ((ren_comp xvalue) s1)
  | ret ( $\_$ ) s0  $\rightarrow$  ret (nvalue) ((ren_value xvalue) s0)
  | letin ( $\_$ ) s0 s1  $\rightarrow$  letin (nvalue) ((ren_comp xvalue) s0) ((ren_comp (upRen_value_value xvalue)) s1)
  | proj ( $\_$ ) s0 s1  $\rightarrow$  proj (nvalue) (( $\lambda x \rightarrow x$ ) s0) ((ren_comp xvalue) s1)
  | caseZ ( $\_$ ) s0  $\rightarrow$  caseZ (nvalue) ((ren_value xvalue) s0)
  | caseS ( $\_$ ) s0 s1 s2  $\rightarrow$  caseS (nvalue) ((ren_value xvalue) s0) ((ren_comp (upRen_value_value xvalue)) s1) ((ren_comp (upRen_value_value xvalue)) s2)
  | caseP ( $\_$ ) s0 s1  $\rightarrow$  caseP (nvalue) ((ren_value xvalue) s0) ((ren_comp (upRen_value_value (upRen_value_value xvalue)) s1))
  end.

Definition up_value_value { m : N } { nvalue : N } ( $\emptyset$  : (fn) (m)  $\rightarrow$  value (nvalue)) : _ :=
  (scons) ((var_value ((S) nvalue)) (var_zero)) ((funcomp) (ren_value (shift))  $\emptyset$ ).

Fixpoint subst_value { nvalue : N } { nvalue : N } (signavalue : (fn) (nvalue)  $\rightarrow$  value (nvalue)) (s : value (nvalue)) : _ :=
  match s with
  | var_value ( $\_$ ) s  $\rightarrow$  signavalue s
  | u ( $\_$ )  $\rightarrow$  u (nvalue)
  | pair ( $\_$ ) s0 s1  $\rightarrow$  pair (nvalue) ((subst_value signavalue) s0) ((subst_value signavalue) s1)
  | inj ( $\_$ ) s0 s1  $\rightarrow$  inj (nvalue) (( $\lambda x \rightarrow x$ ) s0) ((subst_value signavalue) s1)
  | thunk ( $\_$ ) s0  $\rightarrow$  thunk (nvalue) ((subst_comp signavalue) s0)
  end
with subst_comp { nvalue : N } { nvalue : N } (signavalue : (fn) (nvalue)  $\rightarrow$  value (nvalue)) (s : comp (nvalue)) : _ :=
  match s with
  | cu ( $\_$ )  $\rightarrow$  cu (nvalue)
  | force ( $\_$ ) s0  $\rightarrow$  force (nvalue) ((subst_value signavalue) s0)
  |  $\lambda$  ( $\_$ ) s0  $\rightarrow$   $\lambda$  (nvalue) ((subst_comp (signavalue (upRen_value_value xvalue))) s0)
  | app ( $\_$ ) s0 s1  $\rightarrow$  app (nvalue) ((subst_comp signavalue) s0) ((subst_value signavalue) s1)
  | tuple ( $\_$ ) s0 s1  $\rightarrow$  tuple (nvalue) ((subst_comp signavalue) s0) ((subst_comp signavalue) s1)
  | ret ( $\_$ ) s0  $\rightarrow$  ret (nvalue) ((subst_value signavalue) s0)
  | letin ( $\_$ ) s0 s1  $\rightarrow$  letin (nvalue) ((subst_comp signavalue) s0) ((subst_comp (signavalue (upRen_value_value xvalue))) s1)
  | proj ( $\_$ ) s0 s1  $\rightarrow$  proj (nvalue) ((subst_value signavalue) s0) ((subst_comp signavalue) s1)
  | caseZ ( $\_$ ) s0  $\rightarrow$  caseZ (nvalue) ((subst_value signavalue) s0)
  | caseS ( $\_$ ) s0 s1 s2  $\rightarrow$  caseS (nvalue) ((subst_value signavalue) s0) ((subst_comp signavalue) s1) ((subst_comp signavalue) s2)
  | caseP ( $\_$ ) s0 s1  $\rightarrow$  caseP (nvalue) ((subst_value signavalue) s0) ((subst_comp signavalue) s1)
  end.
```

Revisiting Strong Normalization for CBPV

```
emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command ProofTree Interrupt Restart Help

(**
  Master Thesis, Page 10
  This file contains operational semantics, context semantics and bigstep semantics as well as the definition of normal forms, normality and
  evaluation
*)

Set Implicit Arguments.
Require Import Ω Logic List Classes.Morphisms.
Import List Notations.

Require Export CBPV.Terms CBPV.Base CBPV.AbstractReductionSystems.
Import CommaNotation.

(** * Semantics *)

Reserved Notation "A '≥' B" (at level 80).
Reserved Notation "A '↪' B" (at level 80).

(** ** Primitive Reduction *)
Inductive pstep {n: ℕ}: comp n → comp n → P :=
| pstepForce (c: comp n):
  <{c}>! ≥ c
| pstepApp (c: comp (S n)) (c': comp n) v:
  [v..] = c' → (λ c) v ≥ c'
| pstepProj (b: B) (c c₁ c₂: comp n) :
  (c = if b then c₁ else c₂) → proj b (tuple c₁ c₂) ≥ c
| pstepLetIn (c: comp (S n)) c' v:
  c[v..] = c' → $ ← (ret v); c ≥ c'
| pstepCaseS v (b: B) c (c₁ c₂: comp (S n)):
  (if b then c₁ else c₂)[v..] = c → caseS (inj b v) c₁ c₂ ≥ c
| pstepCaseP v₁ v₂ (c: comp (S (S n))) c':
  c[v₂, v₁..] = c' → caseP (pair v₁ v₂) c ≥ c'
where "A '≥' B" := (pstep A B).

(** ** Operational Semantics *)
Inductive step {n: ℕ}: comp n → comp n → P :=
| stepPrimitive (c c': comp n) : c ≥ c' → c > c'
| stepApp (c c': comp n) v: c > c' → c v > c' v
```

Revisiting Strong Normalization for CBPV

```
emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command ProofTree Interrupt Restart Help

intros H1 m n f c H2. inv H2. constructor. constructor. now apply H1.
Qed.

(** ** Semantic Types *)

Fixpoint V {n: ℕ} (A: valtype) (v: value n) :=
  match A with
  | zero =>
    ⊥
  | one =>
    match v with u => ⊤ | _ => ⊥ end
  | U B =>
    match v with <{ c }> => close (C B) c | _ => ⊥ end
  | Σ A1 A2 =>
    match v with inj b v' => closev (V (if b then A1 else A2)) v' | _ => ⊥ end
  | A1 * A2 =>
    match v with pair v1 v2 => closev (V A1) v1 ∧ closev (V A2) v2 | _ => ⊥ end
  end

with C {n: ℕ} (B: comtype) (c: comp n) :=
  match B with
  | cone =>
    match c with cu => ⊤ | _ => ⊥ end
  | F A =>
    match c with ret v => closev (V A) v | _ => ⊥ end
  | Π B1 B2 =>
    match c with tuple c1 c2 => close (C B1) c1 ∧ close (C B2) c2 | _ => ⊥ end
  | A → B =>
    match c with
    | λ c' → ∀ k (f : fin n → fin k) (v : value k),
      closev (V A) v → close (C B) (c'[v, f >> ids])
    | _ => ⊥
    end
  end.

Notation E B c := (close (C B) c).
Notation VV A v := (closev (V A) v).
```

Revisiting Strong Normalization for CBPV

```
emacs25@kathrin-HP-EliteBook-820-G3
File Edit Options Buffers Tools Coq Proof-General Tokens Holes Outline Hide/Show YASnippet Help
GState GContext GGoal GRetract GUndo GNext GUse GGoTo GQed GHome GFind GInfo GCommand GProofTree GInterrupt GRestart GHelp

Qed.

Lemma compat_force v:
  Γ ⊨ v ::: U B → Γ ⊨ v! ::: B.
Proof.
  intros H1 m γ H. asimpl. apply compat_force_E. now apply H1.
Qed.

Lemma compat_caseZ v:
  Γ ⊨ v ::: zero → Γ ⊢ caseZ v ::: B.
Proof.
  intros H1 m γ H. asimpl. apply compat_caseZ_E. now apply H1.
Qed.

Lemma compat_caseS v c1 c2:
  Γ ⊨ v ::: Σ A1 A2 →
  A1 .: Γ ⊢ c1 ::: B →
  A2 .: Γ ⊢ c2 ::: B →
  Γ ⊢ caseS v c1 c2 ::: B.
Proof.
  intros H' H1 H2 m γ H; specialize (H' m γ H). asimpl.
  apply (compat_caseS_E (A1 := A1) (A2 := A2)).
  - assumption.
  - specialize (H1 _ _ (G_ext _ H)). asimpl in H1. eapply close_sn, H1.
  - specialize (H2 _ _ (G_ext _ H)). asimpl in H2. eapply close_sn, H2.
  - intros v' Vv'. asimpl. apply H1. now apply G_scons.
  - intros v' Vv'. asimpl. apply H2. now apply G_scons.
Qed.

Lemma compat_caseP v c:
  Γ ⊨ v ::: A1 * A2 →
  A2 .: (A1 .: Γ) ⊢ c ::: B →
  Γ ⊢ caseP v c ::: B.
Proof.
  intros H' H1 m γ H; specialize (H' m γ H). asimpl.
  apply (compat_caseP_E (A1 := A1) (A2 := A2)).
  - exact H'.
  - intros H'. asimpl. apply H1. now apply G_scons.
  - intros H'. asimpl. apply H2. now apply G_scons.
Qed.

1 subgoal (ID 1396)
- n : ℕ
- Γ : ctx n
- A, A1, A2 : valtype
- B, B1, B2 : comptype
- v : value n
- c1, c2 : comp (S n)
- m : ℕ
- γ : fin n → value m
- H' : VV (Σ A1 A2) v[γ]
- H1 : A1, Γ ⊢ c1 ::: B
- H2 : A2, Γ ⊢ c2 ::: B
- H : G Γ γ
- v' : value m
- Vv' : VV A1 v'

E B c1[var var_zero, γ >> ⟨↑⟩][v', ids]
```

Revisiting Strong Normalization for CBPV

The screenshot shows the Emacs editor with the Coq development environment. The title bar indicates the file is `emacs25@kathrin-HP-EliteBook-820-G3`. The menu bar includes File, Edit, Options, Buffers, Tools, Coq, Proof-General, Tokens, Holes, Outline, Hide/Show, YASnippet, and Help. The toolbar contains icons for State, Context, Goal, Retract, Undo, Next, Use, Goto, QED, Home, Find, Info, Command, Proof tree, Interrupt, Restart, and Help.

Left Pane (Coq Code):

```
Qed.

Lemma compat_force v:
  Γ ⊨ v ::: U B → Γ ⊨ v! ::: B.
Proof.
  intros H1 m γ H. asimpl. apply compat_force_E. now apply H1.
Qed.

Lemma compat_caseZ v:
  Γ ⊨ v ::: zero → Γ ⊨ caseZ v ::: B.
Proof.
  intros H1 m γ H. asimpl. apply compat_caseZ_E. now apply H1.
Qed.

Lemma compat_caseS v c1 c2:
  Γ ⊨ v ::: Σ A1 A2 →
  A1 .: Γ ⊨ c1 ::: B →
  A2 .: Γ ⊨ c2 ::: B →
  Γ ⊨ caseS v c1 c2 ::: B.
Proof.
  intros H' H1 H2 m γ H; specialize (H' m γ H). asimpl.
  apply (compat_caseS_E (A1 := A1) (A2 := A2)).
  - assumption.
  - specialize (H1 _ _ (G_ext _ H)). asimpl in H1. eapply close_sn, H1.
  - specialize (H2 _ _ (G_ext _ H)). asimpl in H2. eapply close_sn, H2.
  - intros v' Vv'. asimpl. apply H1. now apply G_scons.
  - intros v' Vv'. asimpl. apply H2. now apply G_scons.
Qed.

Lemma compat_caseP v c:
  Γ ⊨ v ::: A1 * A2 →
  A2 .: (A1 .: Γ) ⊨ c ::: B →
  Γ ⊨ caseP v c ::: B.
Proof.
  intros H' H1 m γ H; specialize (H' m γ H). asimpl.
```

Right Pane (Subgoal):

```
1 subgoal (ID 2045)
- n : ℕ
- Γ : ctx n
- A, A1, A2 : valtype
- B, B1, B2 : comptype
- v : value n
- c1, c2 : comp (S n)
- m : ℕ
- γ : fin n → value m
- H' : VV (Σ A1 A2) v[γ]
- H1 : A1, Γ ⊨ c1 ::: B
- H2 : A2, Γ ⊨ c2 ::: B
- H : G Γ γ
- v' : value m
- Vv' : VV A1 v'

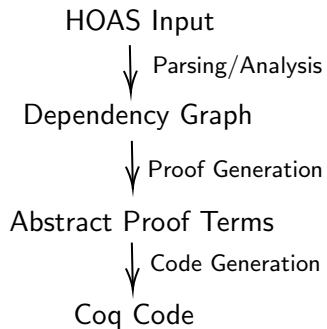
E B c1[v', γ]
```

Bottom Status Bar:

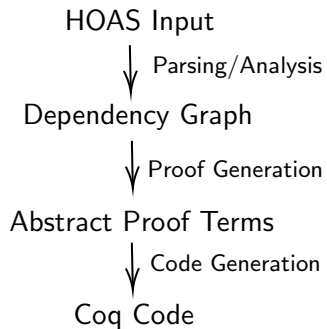
```
uU:%%- *goals* All L18 (Coq Goals Utoks)
```

The Autosubst 2 Compiler

Generation of Code

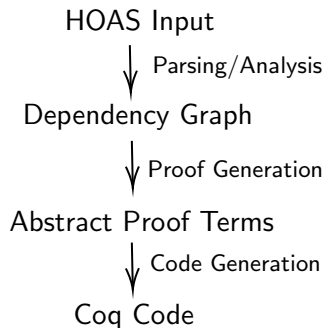


Generation of Code

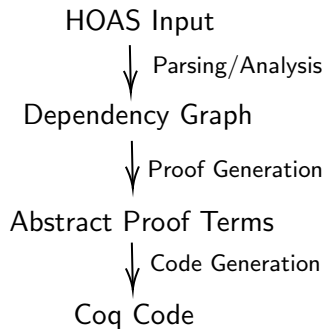


Generation of Code

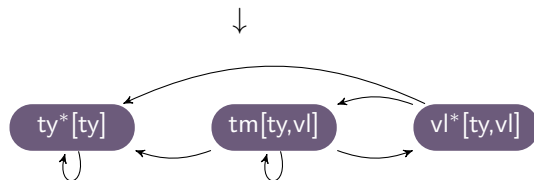
```
arr : ty → ty → ty  
all : (ty → ty) → ty  
...
```



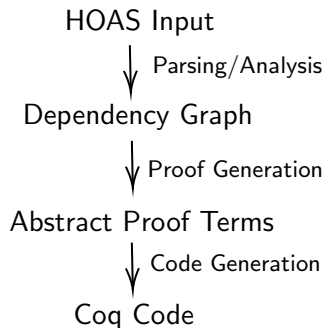
Generation of Code



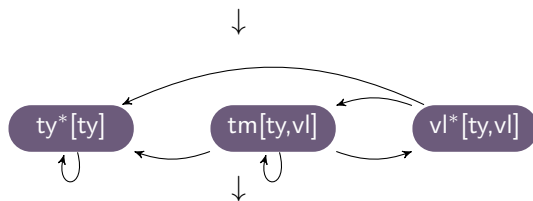
```
arr : ty → ty → ty  
all : (ty → ty) → ty  
...
```



Generation of Code

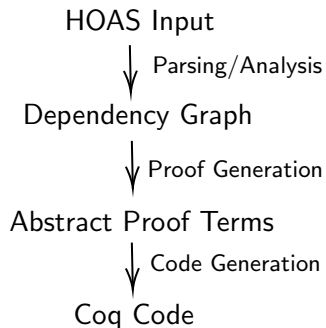


```
arr : ty → ty → ty  
all : (ty → ty) → ty  
...
```

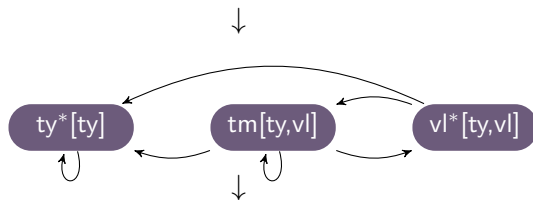


```
[SentenceInductive (Inductive  
[InductiveBody "ty" [("n", TermConst Nat)]  
TermType [...]]), ...]
```

Generation of Code



```
arr : ty → ty → ty
all : (ty → ty) → ty
...
```



```
[SentenceInductive (Inductive
[InductiveBody "ty" [("n", TermConst Nat)]
TermType [...]]), ...]
```

↓

```
Inductive ty (n : nat) : Type :=
| var_ty : fin n → ty n
| arr : ty n → ty n → ty n
| all : ty (1 + n) → ty n.
```

Well-Scoped Syntax

Well-Scoped Syntax [Bird/Paterson '99]

Example

Terms are indexed by the number of free variables:

Inductive `tm (n : nat) : Type :=`

- | `var_tm : fin n → tm n`
- | `app : tm n → tm n → tm n`
- | `lam : tm (1 + n) → tm n.`

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

$$(- \vdash - : -) : \mathcal{L}(\text{tm}) \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \mathcal{P}$$

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

$$(_ \vdash _ : _) : \mathcal{L}(\text{tm}) \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \mathcal{P}$$

$$(_ \vdash _ : _) : (\text{fin } n \rightarrow \text{ty } m) \rightarrow \text{tm } m \ n \rightarrow \text{ty } n \rightarrow \mathcal{P}$$

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

$$(- \vdash - : -) : \mathcal{L}(\text{tm}) \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \mathcal{P}$$

$$(- \vdash - : -) : (\text{fin } n \rightarrow \text{ty } m) \rightarrow \text{tm } m \ n \rightarrow \text{ty } n \rightarrow \mathcal{P}$$

$$\frac{x < |\Gamma|}{\Gamma \vdash x : \text{nth } x \ \Gamma} \quad \text{vs.} \quad \frac{}{\Gamma \vdash x : \Gamma \ x}$$

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

$$(- \vdash - : -) : \mathcal{L}(\text{tm}) \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \mathcal{P}$$

$$(- \vdash - : -) : (\text{fin } n \rightarrow \text{ty } m) \rightarrow \text{tm } m \, n \rightarrow \text{ty } n \rightarrow \mathcal{P}$$

$$\frac{x < |\Gamma|}{\Gamma \vdash x : \text{nth } x \, \Gamma}$$

vs.

$$\frac{}{\Gamma \vdash x : \Gamma \, x}$$

$$\frac{\text{map } \langle \uparrow \rangle \, \Gamma \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

vs.

$$\frac{\Gamma \circ \langle \uparrow \rangle \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

$$(- \vdash - : -) : \mathcal{L}(\text{tm}) \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \mathcal{P}$$

$$(- \vdash - : -) : (\text{fin } n \rightarrow \text{ty } m) \rightarrow \text{tm } m \ n \rightarrow \text{ty } n \rightarrow \mathcal{P}$$

$$\frac{x < |\Gamma|}{\Gamma \vdash x : \text{nth } x \ \Gamma}$$

vs.

$$\frac{}{\Gamma \vdash x : \Gamma \ x}$$

$$\frac{\text{map } \langle \uparrow \rangle \ \Gamma \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

vs.

$$\frac{\Gamma \circ \langle \uparrow \rangle \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

- Statements about closed terms:

Well-Scoped Syntax [Bird/Paterson '99]

- Allows functional reasoning:

$$(- \vdash - : -) : \mathcal{L}(\text{tm}) \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \mathcal{P}$$

$$(- \vdash - : -) : (\text{fin } n \rightarrow \text{ty } m) \rightarrow \text{tm } m \, n \rightarrow \text{ty } n \rightarrow \mathcal{P}$$

$$\frac{x < |\Gamma|}{\Gamma \vdash x : \text{nth } x \, \Gamma}$$

vs.

$$\frac{}{\Gamma \vdash x : \Gamma \, x}$$

$$\frac{\text{map } \langle \uparrow \rangle \, \Gamma \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

vs.

$$\frac{\Gamma \circ \langle \uparrow \rangle \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

- Statements about closed terms:

$$\forall (s : \text{tm } m \, 0) \, A. \vdash s : A \rightarrow \text{SN } s$$

Well-Scoped Syntax [Bird/Paterson '99]

- More type safety:

Well-Scoped Syntax [Bird/Paterson '99]

- More type safety:

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash \Lambda X.s : \forall X.A} \quad \text{vs.} \quad \frac{\Gamma \circ \langle \uparrow \rangle \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

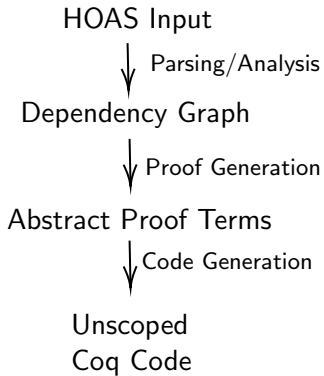
- More type safety:

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash \Lambda X.s : \forall X.A} \quad \text{vs.} \quad \frac{\Gamma \circ \langle \uparrow \rangle \vdash s : A}{\Gamma \vdash \Lambda.s : \forall.A}$$

$$\begin{array}{l} \overline{M} \overline{N} = \text{let } x \leftarrow \overline{M} \text{ in} \\ \quad \text{let } y \leftarrow \overline{N} \text{ in} \\ \quad (!x) y \end{array} \quad \text{vs.} \quad \begin{array}{l} \overline{M} \overline{N} = \text{let } \leftarrow \overline{M} \text{ in} \\ \quad \text{let } \leftarrow \overline{N} \langle \uparrow \rangle \text{ in} \\ \quad (!1) 0 \end{array}$$

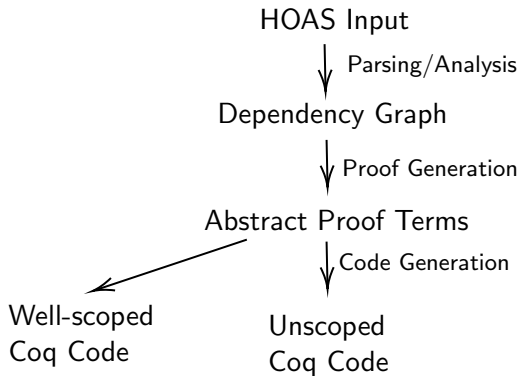
Well-Scoped Syntax

Implementation



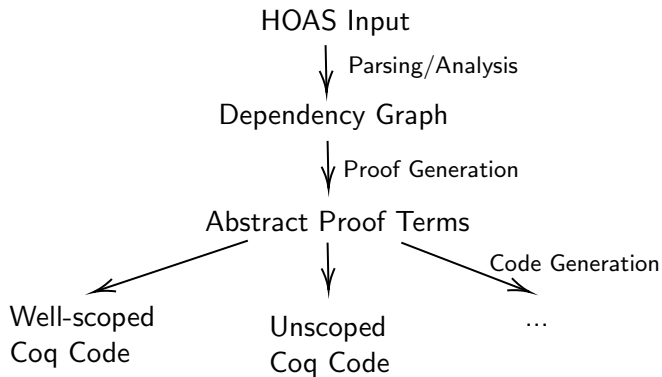
Well-Scoped Syntax

Implementation



Well-Scoped Syntax

Implementation



Vector Substitutions

Vector Substitutions

Motivation

How to formalize instantiation for multivariate, potentially mutually inductive systems – e.g. call-by-value System F (F_{CBV})?

$A, B \in ty ::= \textcolor{blue}{X} \mid A \rightarrow B \mid \forall \textcolor{blue}{X}.A$

Types

$s, t \in tm ::= s \ t \mid s \ A \mid v$

Terms

$u, v \in vl ::= \textcolor{orange}{x} \mid \lambda(\textcolor{orange}{x} : \textcolor{orange}{A}).s \mid \textcolor{blue}{\Lambda} \textcolor{blue}{X}.s$

Values

Vector Substitutions

Motivation

How to formalize instantiation for multivariate, potentially mutually inductive systems – e.g. call-by-value System F (F_{CBV})?

$$A, B \in ty ::= X \mid A \rightarrow B \mid \forall X. A$$

Types

$$s, t \in tm ::= s \ t \mid s \ A \mid v$$

Terms

$$u, v \in vl ::= x \mid \lambda(x : A). s \mid \bigwedge X. s$$

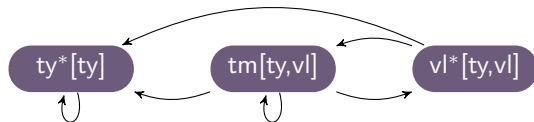
Values

Solution: *Vectorise* parallel substitutions:

$$-[-; -] : vl \rightarrow (\mathbb{N} \rightarrow ty) \rightarrow (\mathbb{N} \rightarrow vl) \rightarrow vl$$

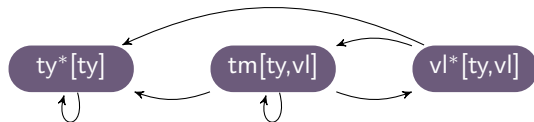
Vector Substitutions

Implementation



Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

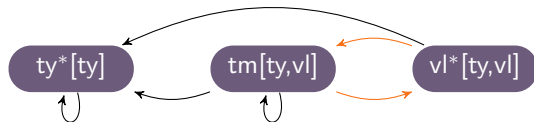
$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

■ Traverses values

- homomorphically

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

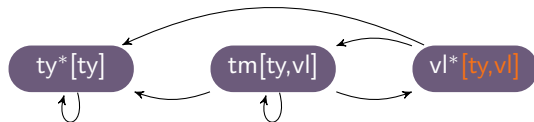
$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

■ Traverses values

- ▶ homomorphically
- ▶ mutually recursive

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

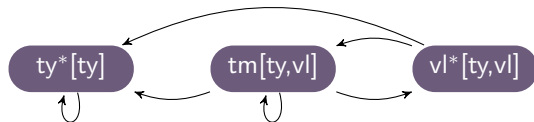
$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

■ Traverses values

- ▶ homomorphically
- ▶ mutually recursive
- ▶ with the inferred vector

Vector Substitutions

Implementation

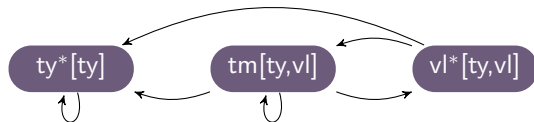


$$\begin{aligned}x[\sigma, \tau] &= \tau x \\(\lambda A. s)[\sigma, \tau] &= \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)] \\(\Lambda. s)[\sigma, \tau] &= \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]\end{aligned}$$

- Traverses values
 - ▶ homomorphically
 - ▶ mutually recursive
 - ▶ with the inferred vector
- Take care of:

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

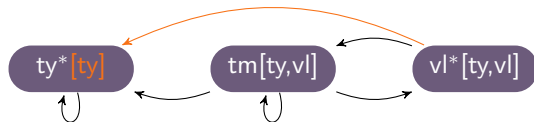
$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

- Traverses values
 - ▶ homomorphically
 - ▶ mutually recursive
 - ▶ with the inferred vector
- Take care of:
 - ▶ Projections

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

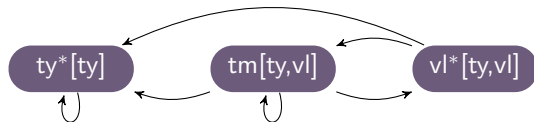
$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

- Traverses values
 - ▶ homomorphically
 - ▶ mutually recursive
 - ▶ with the inferred vector
- Take care of:
 - ▶ Projections
 - ▶ Castings

Vector Substitutions

Implementation

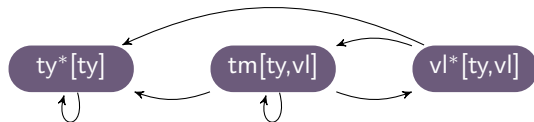


$$\begin{aligned}x[\sigma, \tau] &= \tau x \\(\lambda A. s)[\sigma, \tau] &= \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)] \\(\Lambda. s)[\sigma, \tau] &= \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]\end{aligned}$$

- Traverses values
 - ▶ homomorphically
 - ▶ mutually recursive
 - ▶ with the inferred vector
- Take care of:
 - ▶ Projections
 - ▶ Castings
 - ▶ Traversals of binders

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

$$\uparrow_{tm}^{vl}(\sigma, \tau) = (\sigma, 0_{vl} \cdot \tau \circ [\text{id}_{ty}, \uparrow])$$

$$\uparrow_{tm}^{ty}(\sigma, \tau) = (0_{ty} \cdot \sigma \circ \uparrow, \tau \circ [\uparrow, \text{id}_{vl}])$$

■ Traverses values

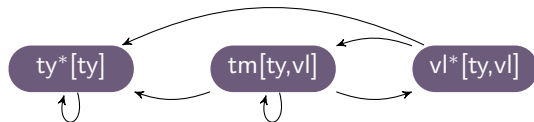
- ▶ homomorphically
- ▶ mutually recursive
- ▶ with the inferred vector

■ Take care of:

- ▶ Projections
- ▶ Castings
- ▶ Traversals of binders

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

$$\uparrow_{tm}^{vl}(\sigma, \tau) = (\sigma, \mathbf{0}_{vl} \cdot \tau \circ [\text{id}_{ty}, \uparrow])$$

$$\uparrow_{tm}^{ty}(\sigma, \tau) = (\mathbf{0}_{ty} \cdot \sigma \circ \uparrow, \tau \circ [\uparrow, \text{id}_{vl}])$$

■ Traverses values

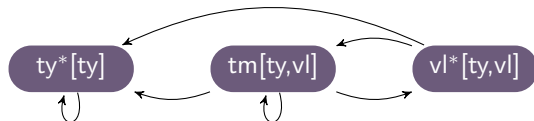
- ▶ homomorphically
- ▶ mutually recursive
- ▶ with the inferred vector

■ Take care of:

- ▶ Projections
- ▶ Castings
- ▶ Traversals of binders

Vector Substitutions

Implementation



$$x[\sigma, \tau] = \tau x$$

$$(\lambda A. s)[\sigma, \tau] = \lambda A[\sigma]. s[\uparrow_{tm}^{vl}(\sigma, \tau)]$$

$$(\Lambda. s)[\sigma, \tau] = \Lambda. s[\uparrow_{tm}^{ty}(\sigma, \tau)]$$

$$\uparrow_{tm}^{vl}(\sigma, \tau) = (\sigma, 0_{vl} \cdot \tau \circ [\text{id}_{ty}, \uparrow])$$

$$\uparrow_{tm}^{ty}(\sigma, \tau) = (0_{ty} \cdot \sigma \circ \uparrow, \tau \circ [\uparrow, \text{id}_{vl}])$$

■ Traverses values

- ▶ homomorphically
- ▶ mutually recursive
- ▶ with the inferred vector

■ Take care of:

- ▶ Projections
- ▶ Castings
- ▶ Traversals of binders

Why vector substitutions?

Why vector substitutions?

Autosubst 1: Separate instantiation, e.g.

$$-[-]_{ty} : vl \rightarrow (\mathbb{N} \rightarrow ty) \rightarrow vl$$

$$-[-]_{vl} : vl \rightarrow (\mathbb{N} \rightarrow vl) \rightarrow vl$$

Why vector substitutions?

Autosubst 1: Separate instantiation, e.g.

$$-[-]_{ty} : vl \rightarrow (\mathbb{N} \rightarrow ty) \rightarrow vl$$

$$-[-]_{vl} : vl \rightarrow (\mathbb{N} \rightarrow vl) \rightarrow vl$$

Problems:

- We might need instantiation on both sorts to go under binders
 $\Rightarrow$ No mutual inductive syntax

Why vector substitutions?

Autosubst 1: Separate instantiation, e.g.

$$-[_]_{ty} : vl \rightarrow (\mathbb{N} \rightarrow ty) \rightarrow vl$$

$$-[_]_{vl} : vl \rightarrow (\mathbb{N} \rightarrow vl) \rightarrow vl$$

Problems:

- We might need instantiation on both sorts to go under binders
 $\Rightarrow$ No mutual inductive syntax
- How to get an elegant equational theory, e.g. how to commute the different kinds of instantiation?

$$s[\tau]_{vl}[\sigma]_{ty} = s[\sigma]_{ty}[\sigma \circ [\tau]_{ty}]_{vl}$$

Why vector substitutions?

Autosubst 1: Separate instantiation, e.g.

$$-[_]_{ty} : vl \rightarrow (\mathbb{N} \rightarrow ty) \rightarrow vl$$

$$-[_]_{vl} : vl \rightarrow (\mathbb{N} \rightarrow vl) \rightarrow vl$$

Problems:

- We might need instantiation on both sorts to go under binders
 $\Rightarrow$ No mutual inductive syntax
- How to get an elegant equational theory, e.g. how to commute the different kinds of instantiation?

$$s[\tau]_{vl}[\sigma]_{ty} = s[\sigma]_{ty}[\sigma \circ [\tau]_{ty}]_{vl}$$

Not all terms come with instantiation of all substitutions, e.g. for types:

$$-[_] : ty \rightarrow (\mathbb{N} \rightarrow ty) \rightarrow ty$$

Case Studies

Case Studies for Autosubst 2

Contents	Spec	Proofs
POPLMark challenge, part A [Aydemir et al. '05]	151	165
Well-scoped variant of the POPLMark Reloaded Challenge, strong normalization for STLC + Sums [Abel et al. '17]	248	312
Weak normalisation of call-by-value System F	114	60
Equivalence of algorithmic and definitional equivalence [Crary '05, Cave and Pientka '15]	88	135
Call-By-Push-Value [Levy '99, Forster et al. '19]	3950	3750

Restrictions and Future Work

Extensions of the input language:

- To recursive functions [Kaiser et al. '18],
e.g. logical relations
- To more complex binders, e.g. for patterns:
Part B of the POPLMark Challenge
- To dependent types [Schäfer et al. '18]:
Context renaming/context morphism lemmas for typing for free

Wrap-up

Contributions:

- Re-implementation of Autosubst to increase flexibility
- Extension of Autosubst to
 - ▶ vector substitutions
 - ▶ first-order renamings (in the paper!)
 - ▶ well-scoped syntax.
- Several (larger) case studies

Wrap-up

Contributions:

- Re-implementation of Autosubst to increase flexibility
- Extension of Autosubst to
 - ▶ vector substitutions
 - ▶ first-order renamings (in the paper!)
 - ▶ well-scoped syntax.
- Several (larger) case studies

Take-Home Message:

If done right, formalising meta-theory with de Bruijn is easy!

Wrap-up

Contributions:

- Re-implementation of Autosubst to increase flexibility
- Extension of Autosubst to
 - ▶ vector substitutions
 - ▶ first-order renamings (in the paper!)
 - ▶ well-scoped syntax.
- Several (larger) case studies

Take-Home Message:

If done right, formalising meta-theory with de Bruijn is easy!

Available online:

www.ps.uni-saarland.de/extras/autosubst2