

~~Semantics~~ Syntax Lunch:  
Strongly-typed term  
representations in Coq

Andrew Kennedy  
Microsoft Research  
Cambridge

# Context

- Doing denotational semantics in Coq (with Nick Benton and Carsten Varming)
  - Constructive version of domain theory based on Christine Paulin-Mohring's Coq library
  - Extended to support predomains, lifting and solution of recursive domain equations
  - Operational & denotational semantics for call-by-value PCF
    - Proofs of soundness and adequacy
  - Operational & denotational semantics for cbv untyped  $\lambda$ -calculus
    - Proofs of soundness and adequacy
- See our submission to TPHOLs 2009
- Although this is Coq-centric, similar techniques would apply elsewhere (e.g. Agda, Haskell GADTs)

# This talk

- Doing syntax in Coq
- We want crisp theorems and definitions. As on paper:

**Soundness.** If  $\vdash e:\tau$  and  $e \Downarrow v$  then  $\llbracket e \rrbracket = \eta \circ \llbracket v \rrbracket$ .

**Adequacy.** If  $\vdash e:\tau$  and  $\llbracket e \rrbracket \neq \emptyset = [x]$  then  $\exists v, e \Downarrow v$ .

**Logical Relation.**

$$R_{\tau_1 \rightarrow \tau_2} = \{(d, \text{fix } f(x).e) \mid \forall d_1, v_1, (d_1, v_1) \in R_{\tau_1} \Rightarrow (d \ d_1, e[v_1/x, v/f]) \in (R_{\tau_2})_{\perp}\}$$

- In Coq:

**Theorem Soundness:** forall ty (e : CExp ty) v, e ==> v -> SemExp e == eta << SemVal v.

**Corollary Adequacy:** forall ty (e : CExp ty) d, SemExp e tt == val d -> exists v, e ==> v.

**Fixpoint relVal ty : SemTy ty -> CValue ty -> Prop :=**

**match ty with ...**

**| ty1 --> ty2 =>**

**fun d v => exists e, v = TFIX e /\**

**forall d1 v1, relVal ty1 d1 v1 -> liftRel (relVal ty2) (d d1) (substExp [ v1, v ] e)**

**end.**

# Binders (again!)

- As usual, we must decide how to represent variables and binders
  - Concrete: de Bruijn indices
  - Concrete: names
  - Concrete: locally nameless
  - Higher-Order Abstract Syntax
  - Whatever
- Claim:
  - “strongly-typed de Bruijn” works very nicely
  - At least for simple types, can be combined with typed terms to get representations of terms that are **well-typed by construction**
  - “But that’s just GADTs!” say the Haskell cool kids
  - Well, yes, but just try proving theorems with them...

# First attempt

- “Pre-terms” are just abstract syntax, with nats for variables (de Bruijn index)

```
Inductive value :=
| VAR: nat -> value
| LAMBDA: Ty -> Exp -> value
...
with Exp :=
| APP : Val -> Val -> Exp
...
```

- Separate inductive type for typing judgments, with proofs of well-scoped-ness in instances

```
Inductive Vtype (env:Env) (t:Ty) :=
| TVAR: forall m , nth_error env m = Some t -> VType env (VAR m) t
| TLAMBDA: forall a b e, t = a --> b -> Etype (a :: env) e b -> Vtype env (LAMBDA a e) t
...
with Etype (env:Env) (t:Ty) :=
| TAPP: forall t' v1 v2, Vtype env v1 (t'-->t) -> Vtype env v2 t' -> Etype env (APP v1 v2) t
```

# First attempt, cont.

- This works OK, but statements and proofs become bogged down with de Bruijn index management e.g.

Theorem FundamentalTheorem:

```
(forall E t' v (tv:E |v- v ::: t') t (teq: LV t = t') (d:SemEnv E) s1, length s1 = length E ->
(forall i s (h:nth_error s1 i = value s) ti, nth_error E i = Some ti -> nil |v- s ::: ti) ->
(forall i ti (h:nth_error E i = Some (LV ti)) si (hs:nth_error s1 i = Some si),
  @grel ti (projenv h d) si) ->
@vrel t (typeCoersion (sym_equal teq) (SemVal tv d)) (ssubstV s1 v)) /\
(forall E t' e (te:E |e- e ::: t') t (teq : LVE t = t') (d:SemEnv E) s1, length s1 = length E ->
(forall i s (h:nth_error s1 i = value s) ti, nth_error E i = Some ti -> nil |v- s ::: ti) ->
(forall i ti (h:nth_error E i = Some (LV ti)) si (hs:nth_error s1 i = Some si),
  @grel ti (projenv h d) si) ->
@erel t (liftedTypeCoersion (sym_equal teq) (SemExp te d)) (ssubstE s1 e)).
```

- Worse, issues of “proof irrelevance” arise, as there are proof objects inside the term representation

# Second attempt: typed syntax

- Terms are **well-scoped** by definition
  - (no proofs of well-scoped-ness buried inside)
- Terms are **well-typed** by definition (no separate typing judgment)
  - Haskell programmers would call this a “GADT”
  - Dependent type fans would call it an “internal” representation

- Statements become much smaller:

Theorem FundamentalTheorem:

(forall env ty v senv s, relEnv env senv s -> relVal ty (SemVal v senv) (substVal s v))

∧

(forall env ty e senv s, relEnv env senv s -> liftRel (relVal ty) (SemExp e senv) (substExp s e)).

- Getting the right definitions and lemmas for substitution is crucial.

# Variables

- First, define syntax for types and environments:

Inductive Ty := Int | Bool | Arrow ( $\tau_1 \tau_2 : \text{Ty}$ ) | Prod ( $\tau_1 \tau_2 : \text{Ty}$ ).

Infix " -> " := Arrow.

Infix " \* " := Prod (at level 55).

Definition Env := list Ty.

- Now, define “typed” variables:

Inductive Var : Env  $\rightarrow$  Ty  $\rightarrow$  Type :=

| ZVAR :  $\forall \Gamma \tau, \text{Var } (\tau :: \Gamma) \tau$

| SVAR :  $\forall \Gamma \tau \tau', \text{Var } \Gamma \tau \rightarrow \text{Var } (\tau' :: \Gamma) \tau$ .

- Variables are indexed by their type and environment
- The structure of a variable of type  $\text{Var } \Gamma \tau$  is a proof that  $\tau$  is at some position  $i$  in the environment  $\Gamma$ .

# Terms

- Likewise, terms are indexed by type and environment:

Inductive *Value* : Env → Ty → Type :=

| *TINT* : ∀ Γ, nat → Value Γ Int

| *TBOOL* : ∀ Γ, bool → Value Γ Bool

| *TVAR* : ∀ Γ τ, Var Γ τ → Value Γ τ

| *TFIX* : ∀ Γ τ<sub>1</sub> τ<sub>2</sub>, Exp (τ<sub>1</sub> :: τ<sub>1</sub> → τ<sub>2</sub> :: Γ) τ<sub>2</sub> → Value Γ (τ<sub>1</sub> → τ<sub>2</sub>)

| *TPAIR* : ∀ Γ τ<sub>1</sub> τ<sub>2</sub>, Value Γ τ<sub>1</sub> → Value Γ τ<sub>2</sub> → Value Γ (τ<sub>1</sub> \* τ<sub>2</sub>)

with *Exp* : Env → Ty → Type :=

| *TFST* : ∀ Γ τ<sub>1</sub> τ<sub>2</sub>, Value Γ (τ<sub>1</sub> \* τ<sub>2</sub>) → Exp Γ τ<sub>1</sub>

| *TSND* : ∀ Γ τ<sub>1</sub> τ<sub>2</sub>, Value Γ (τ<sub>1</sub> \* τ<sub>2</sub>) → Exp Γ τ<sub>2</sub>

| *TOP* : ∀ Γ, (nat → nat → nat) → Value Γ Int → Value Γ Int → Exp Γ Int

| *TGT* : ∀ Γ, Value Γ Int → Value Γ Int → Exp Γ Bool

| *TVAL* : ∀ Γ τ, Value Γ τ → Exp Γ τ

| *TLET* : ∀ Γ τ<sub>1</sub> τ<sub>2</sub>, Exp Γ τ<sub>1</sub> → Exp (τ<sub>1</sub> :: Γ) τ<sub>2</sub> → Exp Γ τ<sub>2</sub>

| *TAPP* : ∀ Γ τ<sub>1</sub> τ<sub>2</sub>, Value Γ (τ<sub>1</sub> → τ<sub>2</sub>) → Value Γ τ<sub>1</sub> → Exp Γ τ<sub>2</sub>

| *TIF* : ∀ Γ τ, Value Γ Bool → Exp Γ τ → Exp Γ τ → Exp Γ τ.

# Beautiful definitions

```
Inductive Ev:  $\forall \tau, CExp \tau \rightarrow CValue \tau \rightarrow Prop :=$   
| e_Val:  $\forall \tau (v : CValue \tau), TVAL v \Downarrow v$   
| e_Op:  $\forall op n_1 n_2, TOP op (TINT n_1) (TINT n_2) \Downarrow TINT (op n_1 n_2)$   
| e_Gt:  $\forall n_1 n_2, TGT (TINT n_1) (TINT n_2) \Downarrow TBOOL (ble\_nat n_2 n_1)$   
| e_Fst:  $\forall \tau_1 \tau_2 (v_1 : CValue \tau_1) (v_2 : CValue \tau_2), TFST (TPAIR v_1 v_2) \Downarrow v_1$   
| e_Snd:  $\forall \tau_1 \tau_2 (v_1 : CValue \tau_1) (v_2 : CValue \tau_2), TSND (TPAIR v_1 v_2) \Downarrow v_2$   
| e_App:  $\forall \tau_1 \tau_2 e (v_1 : CValue \tau_1) (v_2 : CValue \tau_2), substExp [ v_1, TFIX e ]$   
 $e \Downarrow v_2 \rightarrow TAPP (TFIX e) v_1 \Downarrow v_2$   
| e_Let:  $\forall \tau_1 \tau_2 e_1 e_2 (v_1 : CValue \tau_1) (v_2 : CValue \tau_2), e_1 \Downarrow v_1 \rightarrow substExp [$   
 $v_1 ] e_2 \Downarrow v_2 \rightarrow TLET e_1 e_2 \Downarrow v_2$   
| e_IfTrue:  $\forall \tau (e_1 e_2 : CExp \tau) v, e_1 \Downarrow v \rightarrow TIF (TBOOL true) e_1 e_2 \Downarrow v$   
| e_IfFalse:  $\forall \tau (e_1 e_2 : CExp \tau) v, e_2 \Downarrow v \rightarrow TIF (TBOOL false) e_1 e_2 \Downarrow v$   
where "e 'Downarrow' v" := (Ev e v).
```

```
Fixpoint relVal  $\tau : SemTy \tau \rightarrow CValue \tau \rightarrow Prop :=$   
match  $\tau$  with  
| Int  $\Rightarrow$  fun d v  $\Rightarrow v = TINT d$   
| Bool  $\Rightarrow$  fun d v  $\Rightarrow v = TBOOL d$   
|  $\tau_1 \rightarrow \tau_2 \Rightarrow$  fun d v  $\Rightarrow \exists e, v = TFIX e \wedge \forall d_1 v_1, relVal \tau_1 d_1 v_1 \rightarrow liftRel$   
(relVal  $\tau_2$ ) (d d_1) (substExp [ v_1, v ] e)  
|  $\tau_1 * \tau_2 \Rightarrow$  fun d v  $\Rightarrow \exists v_1, \exists v_2, v = TPAIR v_1 v_2 \wedge relVal \tau_1 (FST d) v_1 \wedge$   
relVal  $\tau_2 (SND d) v_2$   
end.
```

# Substitution: how *not* to do it

- First, define a shift (weaken) operation

Definition  $shiftVar \Gamma \tau' \Gamma' : \forall \tau, Var (\Gamma ++ \Gamma') \tau \rightarrow Var (\Gamma ++ \tau' :: \Gamma') \tau$ .

Program Fixpoint  $shiftVal \Gamma \tau' \Gamma' \tau (v : Value (\Gamma ++ \Gamma') \tau) : Value (\Gamma ++ \tau' :: \Gamma') \tau :=$

  match  $v$  with

    |  $TVAR \_ \_ v \Rightarrow TVAR (shiftVar \_ v)$

    |  $TFIX \_ \_ \_ e \Rightarrow TFIX (shiftExp (\Gamma := \_ :: \_ :: env) \_ e)$

    |  $TPAIR \_ \_ \_ e1 e2 \Rightarrow TPAIR (shiftVal \_ e1) (shiftVal \_ e2)$

  ...

- Then, define substitution, shifting under binders. Problem comes when proving lemmas of form  $\forall \Gamma \Gamma' \tau (v : Value (\Gamma ++ \Gamma')) \tau \dots$
- This is not an instance of the general induction principle for terms. Instead, we must prove  $\forall \Gamma_0 (v : Value \Gamma_0) \tau, \forall \Gamma \Gamma', \Gamma_0 = \Gamma ++ \Gamma' \rightarrow \dots$
- Welcome to the weird and wonderful world of equality, casts, and perhaps even former British Prime Ministers...

# Substitution: how to do it

- Instead of defining a special shift/weaken operation, define a more general notion of *renaming*

Definition *Renaming*  $\Gamma \Gamma' := \forall \tau, \text{Var } \Gamma \tau \rightarrow \text{Var } \Gamma' \tau.$

- “Lifting” of a renaming to a larger environment (e.g. under a binder) is just another renaming, so we can then define

Fixpoint *renameVal*  $\Gamma \Gamma' \tau (v : \text{Value } \Gamma \tau) : \text{Renaming } \Gamma \Gamma' \rightarrow \text{Value } \Gamma' \tau :=$

- We can then define substitutions, and the “apply substitution” function:

Definition *Subst*  $\Gamma \Gamma' := \forall \tau, \text{Var } \Gamma \tau \rightarrow \text{Value } \Gamma' \tau.$

Fixpoint *substVal*  $\Gamma \Gamma' \tau (v : \text{Value } \Gamma \tau) : \text{Subst } \Gamma \Gamma' \rightarrow \text{Value } \Gamma' \tau :=$

- In order to define “lifting” of substitution in the above, we use *renameVal*. We have “bootstrapped” substitution using renaming.

# Substitution: how to do it

- We now define 4 notions of composition (renaming with renaming, renaming with substitution, substitution with renaming, and substitution with substitution)
- Associated with these notions we have four lemmas. The trick here is: prove these *in order*, each building on the last. Roughly speaking:

$\text{renameVal } (r' \circ r) v = \text{renameVal } r' (\text{renameVal } r v)$

$\text{substVal } (s \circ r) v = \text{substVal } s (\text{renameVal } r v)$

$\text{substVal } (r \circ s) v = \text{renameVal } r (\text{substVal } s v)$

$\text{substVal } (s' \circ s) v = \text{substVal } s' (\text{substVal } s v)$

# Summary

- Index variables and terms by type and environment
  - Untyped variant would use “bounded natural numbers” for environment
- Bootstrap definition of substitution by using renaming
- Bootstrap composition lemmas in sequence

# Drawbacks?

- Dependencies everywhere. Fortunately, Coq 8.2 helps out with new tactics (“dependent destruction”) and definitional mechanisms (“Program”)
- It’s a bit painful to have to define both renamings and substitutions, and their compositions
- This leaks out into the semantics too e.g. For the denotational semantics we proved a “renaming” lemma that was then used to prove the “substitution” lemma

# Applicability

- More complex first-order binding forms work fine e.g. ML-style pattern matching
- Currently attempting to formalize System F, with terms well-typed by construction
- For more complex type systems, can just use well-scoped-by-definition, and a separate inductive type to represent typing judgments e.g. See

*Formalized Metatheory with Terms Represented by an Indexed Family of Types*,  
Robin Adams, TYPES 2004

in which PTS is formalized in Coq.

# Related work

- Lots of previous work on indexed families for representing terms. But (unless I've missed it), nothing direct in Coq even for the simply-typed lambda-calculus
- Most relevant is:

*Monadic Presentations of Lambda Terms Using Generalized Inductive Types*,  
Altenkirch & Reus, CSL'99

*Formalized Metatheory with Terms Represented by an Indexed Family of Types*,  
Adams, TYPES 2004

*Type-Preserving Renaming and Substitution*, McBride, unpublished.