Strongly-typed term representations in Coq

Nick Benton, Andrew Kennedy &
Chung-Kil Hur

Context

- Denotational semantics in Coq (with Carsten Varming, TPHOLs'09)
  - Constructive version of domain theory based on 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 λ-calculus
    - Proofs of soundness and adequacy
- Compositional compiler correctness for simply-typed language (ICFP'09)
  - Logical relations between domains and operational semantics of low-level code
  - Compositional, extensional
- Extension to polymorphic source
  - System F source language
  - Operational on both sides

This talk

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

Soundness. If $\vdash$ e:τ and e $\Downarrow$ v then $[\![e]\!] = \eta \circ [\![v]\!]$.

Adequacy. If $\vdash$ e:τ and $[\![e]\!]\ \emptyset$ = [x] then $\exists$ v, e $\Downarrow$ v.

Logical Relation.

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

- 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.
```

Oh no! binders!

- As usual, we must decide how to represent variables and binders
  - Concrete: de Bruijn indices
  - Concrete: names
  - Concrete: locally nameless
  - (Parametric) Higher-Order Abstract Syntax
  - Whatever
- Claim:
  - "strongly-typed de Bruijn" works quite nicely
  - At least for simple types, can be combined with typed terms to get representations of terms that are well-typed by construction
  - just Haskell-style GADTs, but we also prove theorems

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

- And intensional type theory starts to bite – proof objects inside terms mean you have to start worrying about proof irrelevance, etc.

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 call this a “GADT”
  - Dependent type theorists 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:

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

$\texttt{Infix}$ " -> " := Arrow.
$\texttt{Infix}$ " * " := Prod ($\texttt{at}$ *level* 55).

$\texttt{Definition}$ Env := *list* Ty.

- Now, define "typed" variables:

$\texttt{Inductive}$ Var : Env $\rightarrow$ Ty $\rightarrow$ $\texttt{Type}$:=
| $ZVAR : \forall\ \Gamma\ \tau,\ Var\ (\tau :: \Gamma)\ \tau$
| $SVAR : \forall\ \Gamma\ \tau\ \tau',\ Var\ \Gamma\ \tau \rightarrow\ Var\ (\tau' :: \Gamma)\ \tau.$

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

Terms

- Likewise, terms are indexed by type and environment:

$\texttt{Inductive}\ \mathit{Value} : \mathsf{Env} \to \mathsf{Ty} \to \texttt{Type} :=$
$|\ \mathit{TINT} : \forall\ \Gamma,\ \mathsf{nat} \to\ \mathit{Value}\ \Gamma\ \mathsf{Int}$
$|\ \mathit{TBOOL} : \forall\ \Gamma,\ \mathit{bool} \to\ \mathit{Value}\ \Gamma\ \mathsf{Bool}$
$|\ \mathit{TVAR} : \forall\ \Gamma\ \tau,\ \mathit{Var}\ \Gamma\ \tau \to\ \mathit{Value}\ \Gamma\ \tau$
$|\ \mathit{TFIX} : \forall\ \Gamma\ \tau_1\ \tau_2,\ \mathit{Exp}\ (\tau_1 :: \tau_1\ \texttt{->}\ \tau_2 :: \Gamma)\ \tau_2 \to\ \mathit{Value}\ \Gamma\ (\tau_1\ \texttt{->}\ \tau_2)$
$|\ \mathit{TPAIR} : \forall\ \Gamma\ \tau_1\ \tau_2,\ \mathit{Value}\ \Gamma\ \tau_1 \to\ \mathit{Value}\ \Gamma\ \tau_2 \to\ \mathit{Value}\ \Gamma\ (\tau_1\ \texttt{*}\ \tau_2)$
$\texttt{with}\ \mathit{Exp} : \mathsf{Env} \to \mathsf{Ty} \to \texttt{Type} :=$
$|\ \mathit{TFST} : \forall\ \Gamma\ \tau_1\ \tau_2,\ \mathit{Value}\ \Gamma\ (\tau_1\ \texttt{*}\ \tau_2) \to \mathit{Exp}\ \Gamma\ \tau_1$
$|\ \mathit{TSND} : \forall\ \Gamma\ \tau_1\ \tau_2,\ \mathit{Value}\ \Gamma\ (\tau_1\ \texttt{*}\ \tau_2) \to \mathit{Exp}\ \Gamma\ \tau_2$
$|\ \mathit{TOP} : \forall\ \Gamma,\ (\mathsf{nat} \to\ \mathsf{nat} \to\ \mathsf{nat}) \to\ \mathit{Value}\ \Gamma\ \mathsf{Int} \to\ \mathit{Value}\ \Gamma\ \mathsf{Int} \to \mathit{Exp}\ \Gamma\ \mathsf{Int}$
$|\ \mathit{TGT} : \forall\ \Gamma,\ \mathit{Value}\ \Gamma\ \mathsf{Int} \to\ \mathit{Value}\ \Gamma\ \mathsf{Int} \to \mathit{Exp}\ \Gamma\ \mathsf{Bool}$
$|\ \mathit{TVAL} : \forall\ \Gamma\ \tau,\ \mathit{Value}\ \Gamma\ \tau \to \mathit{Exp}\ \Gamma\ \tau$
$|\ \mathit{TLET} : \forall\ \Gamma\ \tau_1\ \tau_2,\ \mathit{Exp}\ \Gamma\ \tau_1 \to\ \mathit{Exp}\ (\tau_1 :: \Gamma)\ \tau_2 \to \mathit{Exp}\ \Gamma\ \tau_2$
$|\ \mathit{TAPP} : \forall\ \Gamma\ \tau_1\ \tau_2,\ \mathit{Value}\ \Gamma\ (\tau_1\ \texttt{->}\ \tau_2) \to\ \mathit{Value}\ \Gamma\ \tau_1 \to \mathit{Exp}\ \Gamma\ \tau_2$
$|\ \mathit{TIF} : \forall\ \Gamma\ \tau,\ \mathit{Value}\ \Gamma\ \mathsf{Bool} \to \mathit{Exp}\ \Gamma\ \tau \to \mathit{Exp}\ \Gamma\ \tau \to \mathit{Exp}\ \Gamma\ \tau.$

Beautiful definitions

$\texttt{Inductive}\ Ev\colon \forall\ \tau,\ CExp\ \tau \to CValue\ \tau \to \texttt{Prop} :=$
$|\ e\_Val\colon \forall\ \tau\ (v : CValue\ \tau),\ TVAL\ v \Downarrow v$
$|\ e\_Op\colon \forall\ op\ n_1\ n_2,\ TOP\ op\ (TINT\ n_1)\ (TINT\ n_2) \Downarrow TINT\ (op\ n_1\ n_2)$
$|\ e\_Gt : \forall\ n_1\ n_2,\ TGT\ (TINT\ n_1)\ (TINT\ n_2) \Downarrow TBOOL\ (ble\_nat\ n_2\ n_1)$
$|\ e\_Fst : \forall\ \tau_1\ \tau_2\ (v_1 : CValue\ \tau_1)\ (v_2 : CValue\ \tau_2),\ TFST\ (TPAIR\ v_1\ v_2) \Downarrow v_1$
$|\ e\_Snd : \forall\ \tau_1\ \tau_2\ (v_1 : CValue\ \tau_1)\ (v_2 : CValue\ \tau_2),\ TSND\ (TPAIR\ v_1\ v_2) \Downarrow v_2$
$|\ e\_App : \forall\ \tau_1\ \tau_2\ e\ (v_1 : CValue\ \tau_1)\ (v_2 : CValue\ \tau_2),\ substExp\ [\ v_1,\ TFIX\ \ e\ ]$
$e \Downarrow v_2 \to TAPP\ (TFIX\ \ e)\ v_1 \Downarrow v_2$
$|\ e\_Let : \forall\ \tau_1\ \tau_2\ e_1\ e_2\ (v_1 : CValue\ \tau_1)\ (v_2 : CValue\ \tau_2),\ e_1 \Downarrow v_1 \to substExp\ [$
$v_1\ ]\ e_2 \Downarrow v_2 \to TLET\ e_1\ e_2 \Downarrow v_2$
$|\ e\_IfTrue : \forall\ \tau\ (e_1\ e_2 : CExp\ \tau)\ v,\ e_1 \Downarrow v \to TIF\ (TBOOL\ true)\ e_1\ e_2 \Downarrow v$
$|\ e\_IfFalse : \forall\ \tau\ (e_1\ e_2 : CExp\ \tau)\ v,\ e_2 \Downarrow v \to TIF\ (TBOOL\ false)\ e_1\ e_2 \Downarrow v$
$\texttt{where}$ "e '$\Downarrow$' v" $:= (Ev\ \ e\ \ v)$.

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

Substitution: how *not* to do it

- First, define a shift (weaken) operation

  $\texttt{Definition}\ \mathit{shiftVar}\ \Gamma\ \tau'\ \Gamma' : \forall\ \tau,\ \mathit{Var}\ (\Gamma \mathbin{++} \Gamma')\ \tau \rightarrow \mathit{Var}\ (\Gamma \mathbin{++} \tau' :: \Gamma')\ \tau.$

  $\texttt{Program Fixpoint}\ \mathit{shiftVal}\ \Gamma\ \tau'\ \Gamma'\ \tau\ (v :\ \mathit{Value}\ (\Gamma \mathbin{++} \Gamma')\ \tau) :\ \mathit{Value}\ (\Gamma \mathbin{++}$
  $\tau' :: \Gamma')\ \tau :=$
  $\quad\texttt{match}\ v\ \texttt{with}$
  $\qquad|\ \mathit{TVAR}\ \_\ \_\ v \Rightarrow \mathit{TVAR}\ (\mathit{shiftVar}\ \_\ v)$
  $\qquad|\ \mathit{TFIX}\ \_\ \_\ \_\ e \Rightarrow \mathit{TFIX}\ (\mathit{shiftExp}\ (\Gamma{:=}\_{::}\_{::}\mathrm{env})\ \_\ e)$
  $\qquad|\ \mathit{TPAIR}\ \_\ \_\ \_\ e1\ \ e2 \Rightarrow \mathit{TPAIR}\ (\mathit{shiftVal}\ \_\ e1)\ (\mathit{shiftVal}\ \_\ e2)$
  $\quad\ldots$

- Then, define substitution, shifting under binders. Problem comes when proving lemmas of form $\forall\ \Gamma\ \Gamma'\ \tau\ (v :\ \mathit{Value}\ (\Gamma \mathbin{++} \Gamma'))\ \tau \ldots$

- This is not an instance of the general induction principle for terms. Instead, we must prove $\forall\ \Gamma_0\ (v :\ \mathit{Value}\ \Gamma_0)\ \tau,\ \forall\ \Gamma\ \Gamma',\ \Gamma_0 = \Gamma \mathbin{++} \Gamma' \rightarrow \ldots$

Ugh! Intensional type theory bites you again, lots of casting, etc.

M. Sozeau (2007). A dependently-typed formalization of simply-typed lambda-calculus: substitution, denotation, normalization.

Substitution: how to do it

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

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

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

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

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

  $\texttt{Definition}\ \mathit{Subst}\ \Gamma\ \Gamma' := \forall\ \tau,\ \mathit{Var}\ \Gamma\ \tau \rightarrow\ \mathit{Value}\ \Gamma'\ \tau.$
  $\texttt{Fixpoint}\ \mathit{substVal}\ \Gamma\ \Gamma'\ \tau\ (v :\ \mathit{Value}\ \Gamma\ \tau) :\ \mathit{Subst}\ \Gamma\ \Gamma' \rightarrow\ \mathit{Value}\ \Gamma'\ \tau :=$

- 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:

renameVal (r’ ○ r) v = renameVal r’ (renameVal r v)
substVal (s ○ r) v = substVal s (renameVal r v)
substVal (r ○ s) v = renameVal r (substVal s v)
substVal (s’ ○ s) v = substval s’ (substVal s v)

Experience

- Generally works very nicely in the simply typed case, extends smoothly to pattern matching
- 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
- Staged definitions are not completely encapsulated e.g. For the denotational semantics we proved a “renaming” lemma that was then used to prove the “substitution” lemma

Related work

- Lots of previous work on indexed families for representing terms. But even simply typed lambda calculus doesn't seem to have been done this way in Coq before
- Most relevant are:

Candidates for substitution, Goguen and McKinna. Edinburgh TR, 1997

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 (PTS, well-scoped by definition, separate typing judgement)

Type-Preserving Renaming and Substitution, McBride, 2005.

System F (types)

```
Inductive TyVar : nat -> Type :=
  | ZT : forall n, TyVar (S n)
  | ST : forall n, TyVar n -> TyVar (S n).

Inductive Ty (u: nat) : Type :=
  | Atom : TyVar u -> Ty u
  | Int : Ty u
  | Arrow : Ty u -> Ty u -> Ty u
  | All : Ty (S u) -> Ty u
  | Exist : Ty (S u) -> Ty u
....

Definition RenT u w := TyVar u -> TyVar w.

Definition SubT u w := TyVar u -> Ty w.
```

```
Program Definition RTyL u w (ren: RenT u w) :  RenT (S u) (S w) :=
fun var => match var with
    | ZT _ => (ZT _)
    | ST _ var' => ST (ren var')
  end.

Fixpoint RTyT u w (ren: RenT u w) (ty: Ty u) : Ty w :=
  match ty with
    | Atom v => Atom (ren v)
    | Arrow ty1 ty2 => Arrow (RTyT ren ty1) (RTyT ren ty2)
    | All ty => All (RTyT (RTyL ren) ty)
...
end.

Program Definition STyL u w (sub: SubT u w) : SubT (S u) (S w) :=

Fixpoint STyT u w (sub: SubT u w) (ty: Ty u) : Ty w :=
```

Similar sequence of lemmas about compositions and liftings

System F (terms)

```
Definition Env u := list (Ty u).
Fixpoint STyE u w (sub: SubT u w) (env: Env u) : Env w := ...

Inductive Var u : Env u -> Ty u -> Type :=
| ZV : forall env ty, Var (ty :: env) ty
| SV : forall env ty' ty, Var env ty -> Var (ty' :: env) ty.

Inductive Value u (env: Env u) : (Ty u) -> Type :=
| VAR   : forall ty, Var env ty -> Value env ty
| INT   : nat -> Value env (Int u)
| REC   : forall ty1 ty2, Exp (ty1 :: ty1 --> ty2 :: env) ty2 -> Value env (ty1 --> ty2)
| TLAM : forall ty, @Value (S u) (STyE (shsub _) env ) ty -> Value env (All ty)
| PACK : forall ty ty', Value env (STyT (singsub ty') ty) -> Value env (Exist ty)
...
with Exp u (env: Env u) : (Ty u) -> Type :=
| VAL   : forall ty, Value env ty -> Exp env ty
| LET   : forall ty1 ty2, Exp env ty1 -> Exp (ty1 :: env) ty2 -> Exp env ty2
| APP   : forall ty1 ty2, Value env (ty1 --> ty2) -> Value env ty1 -> Exp env ty2
| TAPP : forall ty (f: Value env (All ty)) (ty' : Ty u), Exp env (STyT (singsub ty') ty)
| UNPK : forall ty ty', Value env (Exist ty') ->
                @Exp (S u) (ty' :: STyE (shsub _) env) (STyT (shsub _) ty) -> Exp env ty
...
```

Again, no equality proofs, direct translation of paper rules

Type substitutions acting on terms

```
Program Fixpoint STyVal u w (sub: SubT u w) (env: Env u) ty  (tv: Value env ty)
 : Value (STyE sub env) (STyT sub ty) :=
  match tv with
    | VAR _ var => VAR (STyVar sub var)
    | INT i => INT _ i
    | REC _ _ e => REC (STyExp sub e)
    | TLAM _ v => TLAM (iso (iso1 _ _ _) (STyVal (STyL sub) v))
    | PACK _ t v => PACK (iso (i      _) (STyVal sub v))
...
  end

with STy                                                              :=
  match
    | VAL
    | APP
    | TAP
    | UNPK _ t v e => UNPK (STyVal sub v) (iso (iso2 _ _ _ _) (STyExp (STyL sub) e))
...
  end.
```

```
Lemma iso1 : forall u w (sub: SubT u w) (env: Env u) (ty : Ty (S u)) (ty' : Ty u),
  (Exp (STyE sub env) (STyT (singsub (STyT sub ty')) (STyT (STyL sub) ty)))
=
  (Exp (STyE sub env) (STyT sub (STyT (singsub ty') ty))).
```

Heterogeneous Equality

- Work with JMeq, show (easily) pretty much everything is a congruence for it in dependently-typed positions:

```
Lemma STyVal_JMeq: forall (u w : nat) (sub sub': SubT u w) (env env': Env u) (ty ty': Ty u)
                        (tv:Value env ty) (tv':Value env' ty'),
  JMeq tv tv' -> sub = sub' -> env = env' -> ty = ty' -> JMeq (STyVal sub tv) (STyVal sub' tv').
```

absorb isos into Jmeq:

```
Lemma iso_elim: forall (A B:Type) (pf: A = B) (a: A), JMeq (iso pf a) a.
```

Term renamings and substitutions

- As before, but abstract commonality into "variable domain maps", mapping variables into things, P

```
Variable P : forall u, (Env u) -> (Ty u) -> Type.
Definition Map u (E E': Env u) := forall t, Var E t -> P E' t.
```

- where P is equipped with operations ops:MapOps

```
Record MapOps :=
  {
    vr : forall u (env: Env u) ty, Var env ty -> P env ty;
    vl : forall u (env: Env u) ty, P env ty -> Value env ty;
    wk : forall u (env: Env u) ty' ty, P env ty -> P (ty' :: env) ty;
    sb : forall u w (sub: SubT u w) (env: Env u) ty, P env ty -> P (STyE sub env) (STyT sub ty)
  }.
```

Generic traversal

```
Program Fixpoint mapVal u (env:Env u) env' (m: Map env env') ty (v : Value env ty) : Value env' ty :=
  match v with
    | VAR _ v => vl ops (m _ v)
    | INT i => INT _ i
    | REC _ _ e => REC (mapExp (liftMap (liftMap m)) e)
    | TLAM _ v => TLAM (mapVal (substMap (shsub _) m) v)
    | PACK _ t v => PACK (mapVal m v)
...
  end
  with mapExp u (env:Env u) env' (m: Map env env') ty (e : Exp env ty) : Exp env' ty :=
   match e with
    | VAL _ v => VAL (mapVal m v)
....
```

liftMap uses wk, substMap uses sb from ops, etc.

Instantiating P with Var gives term Renaming, with Value gives term Subst

Then…

- Have action of term renamings and term substitutions on terms roughly as before
- But also action of type substitutions on term renamings and substitutions
- So *lots* of bread-and-butter lemmas to prove:

```
Lemma STyRen_ss : forall (u : nat) env v w (sub2:SubT v w) (sub1:SubT u v) env'
                                                  (Ren : Renaming env env'),
  JMeq (STyRen sub2 (STyRen sub1 Ren))
       (STyRen (sub2 @ss@ sub1) Ren).
```

Operational semantics still pretty

```
Inductive Tstep : forall (ty:Ty 0), CExp ty -> CExp ty -> Prop :=
(* Value *)
| step_OpN : forall op n1 n2, Tstep (OPN op (INT _ n1) (INT _ n2)) (VAL (INT (u:=0) _ (OpSemNat op n1 n2)))
| step_OpB : forall op n1 n2, Tstep (OPB op (INT _ n1) (INT _ n2)) (VAL (BOOL (u:=0) _ (OpSemBool op n1 n2)))
| step_Fst : forall (ty1:Ty 0) ty2 (v1 : CValue ty1) (v2 : CValue ty2), Tstep (FST (PAIR v1 v2)) (VAL v1)
| step_Snd : forall (ty1:Ty 0) ty2 (v1 : CValue ty1) (v2 : CValue ty2), Tstep (SND (PAIR v1 v2)) (VAL v2)

(* Exp *)
| step_IfTrue  : forall (ty:Ty 0) (e1 e2 : CExp ty), Tstep (IFTE (BOOL _ true) e1 e2) e1
| step_IfFalse : forall (ty:Ty 0) (e1 e2 : CExp ty), Tstep (IFTE (BOOL _ false) e1 e2) e2
| step_Let     : forall (ty1:Ty 0) ty2 (e: Exp [ty1] ty2) (v : CValue ty1), Tstep (LET (VAL v) e) (STmExp {| v |} e)
| step_RApp    : forall (ty1:Ty 0) ty2 (e: Exp [ty1, ty1 --> ty2] ty2) (v : CValue ty1),
                                        Tstep (APP (REC e) v) (STmExp {| v, REC e |} e)
| step_TApp    : forall (ty: Ty 1) (v: CValue ty) (ty' : Ty 0),
                        Tstep (TAPP (TLAM (env:=nil) v) ty') (VAL (STyVal (singsub ty') v))
| step_Unpk    : forall (ty1: Ty 1) ty2 ty' (v: CValue (STyT (singsub ty') ty1)) (e: Exp [ty1] (STyT (shsub _) ty2)),
               Tstep (UNPK (PACK v) e) (iso (Tstep_iso1 _ _) (STmExp {| v |} (STyExp (singsub ty') e)))
| step_Cong    : forall (ty1:Ty 0) ty2 (e1 e1': CExp ty1) (e2: Exp [ty1] ty2),
                                  Tstep e1 e1' -> Tstep (LET e1 e2) (LET e1' e2)
.
```

Experience

- Ends up about 2000 lines for strongly typed System F terms and all the results about substitutions
- JMeq stuff does escape, so rewrites in clients sometimes have to do (very stylised) JMeq congruence reasoning by hand – it'd be very nice to have this done automagically
- But really pays off in getting type and scoping right in e.g. logical relations