

# Engineering does help: a parameterized proof of soundness for structural polymorphism with recursive types

---

Jacques Garrigue

名古屋大学 多元数理科学研究科

# Synopsis

---

- Engineering formal metatheory
- Structural polymorphism
- Reusing formal metatheory
- Encoding a framework with modules

# Engineering formal metatheory

---

**Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, Stephanie Weirich [POPL08]**

Proving soundness for various type systems ( $F_{\leq}$ , ML, CoC) in Coq

Two main ideas to avoid renaming:

- Locally nameless definitions
- Co-finite quantification

## Locally nameless definitions

---

- $\alpha$ -conversion is a pain
- de Bruijn indices in derivations not so nice

Idea: use de Bruijn indices only for bound variables in terms (or type schemes), and name free variables.

$$\frac{x \notin \text{Dom}(E) \cup \text{FV}(t) \quad E, x:S \vdash t^x : T}{E \vdash \lambda t : S \rightarrow T}$$

## Co-finite quantification

---

- we need to change non-locally bound names

Idea: quantify bound names universally, using a co-finite exclusion set

$$\frac{\forall x \notin L \quad E, x:S \vdash t^x : T}{E \vdash \lambda t : S \rightarrow T}$$

Intuition:  $L$  should be a superset of  $\text{Dom}(E) \cup \text{FV}(T)$ , so that there is no conflict, but we can grow  $L$  as needed when transforming proofs.

## Example with weakening

---

Usually weakening requires renaming if  $x \in \text{Dom}(E')$

$$\frac{x \notin \text{Dom}(E) \cup \text{FV}(t) \quad E, x:S \vdash t^x : T}{E \vdash \lambda t : S \rightarrow T} \longrightarrow \frac{y \notin \text{Dom}(E, E') \cup \text{FV}(t) \quad E, E', y:S \vdash t^y : T}{E, E' \vdash \lambda t : S \rightarrow T}$$

No renaming needed if we enlarge  $L$  !

$$\frac{\forall x \notin L \quad E, x:S \vdash t^x : T}{E \vdash \lambda t : S \rightarrow T} \longrightarrow \frac{\forall x \notin L \cup \text{Dom}(E') \quad E, E', x:S \vdash t^x : T}{E, E' \vdash \lambda t : S \rightarrow T}$$

## Co-finite quantification and ML let

---

The translation of ML's let is a bit more involved:

$$\frac{E \vdash t_1 : T_1 \quad \bar{\alpha} \cap \text{FV}(E) = \emptyset \quad E, x : \forall \bar{\alpha}. T_1 \vdash t_2 : T}{E \vdash \text{let } x = t_1 \text{ in } t_2 : T}$$

becomes

$$\frac{\forall \bar{\alpha} \notin L_1 \quad E \vdash t_1 : T_1^{\bar{\alpha}} \quad \forall x \notin L_2 \quad E, x : \forall^{|\bar{\alpha}|} T_1 \vdash t_2^x : T}{E \vdash \text{let } t_1 \text{ in } t_2 : T}$$

The only condition on  $\bar{\alpha}$  is the derivability of  $E \vdash t_1 : T_1^{\bar{\alpha}}$

## Example with weakening

---

Again, without co-finite quantification, one has to rename the  $\bar{\alpha}$  if  $E$  grows, as they may be referred by new bindings. This is particularly stupid as the new bindings do not contribute to the derivation.

An alternative approach would be to explicitly consider only relevant bindings.

$$\frac{E \vdash t_1 : T_1 \quad \bar{\alpha} \cap \text{FV}(E |_{\text{FV}(t_1)}) = \emptyset \quad E, x : \forall \bar{\alpha}. T_1 \vdash t_2 : T}{E \vdash \text{let } x = t_1 \text{ in } t_2 : T}$$

The co-finite approach, where the constraint on  $\bar{\alpha}$  is left implicit, is much smarter.

# Engineering formal metatheory

---

All the proofs are extremely short.

- Thanks to clever automation of the notion of freshness used by co-finite quantification, maintaining the conditions is easy.
- Many simple lemmas are required, but they are about types and terms, not derivations.
- Renaming inside derivations is very rarely needed. Soundness of  $F_{\leq}$  or ML doesn't involve it.
- It is claimed that renaming lemmas for derivations can be obtained from substitution lemmas if needed.

# Structural polymorphism

---

A typing framework for polymorphic variants and records

- sufficient to describe most of **Objective Caml**
- polymorphism is described by **local constraints**
- constraints are abstract, and **constraint domains** can be defined independently of the framework
- constraints are kept in a **kinding environment**
- the kinding environment can be **recursive**

## Types and kinds

---

Types are mixed with kinds in a mutually recursive way.

|          |       |   |                     |               |
|----------|-------|---|---------------------|---------------|
| $T$      | $::=$ | $\alpha$  | type variable       |               |
|          |       | $ $   | $u$                 | base type     |
|          |       | $ $   | $T \rightarrow T$   | function type |
| $\sigma$ | $::=$ | $T \mid \forall \bar{\alpha}. K \triangleright T$ | polytypes           |               |
| $K$      | $::=$ | $\emptyset \mid K, \alpha :: k$                   | kinding environment |               |
| $k$      | $::=$ | $\bullet \mid (C; R)$                             | kind                |               |
| $R$      | $::=$ | $\{r(a, T), \dots\}$                              | relation set        |               |

Type judgments contain both a type and a kind environment.

$$K; E \vdash e : T$$

## Example: polymorphic records

---

Kinds have the form  $(L, U; T)$ , such that  $L \subset U$ .

$jac = \{name = \text{"Jacques"}, age = 36\}$

$jac : \forall \alpha :: (\emptyset, \{name, age\}; \{name : string, age : int\}) \triangleright \alpha$

$birthday = \text{fun } x \rightarrow x.age + 1$

$birthday : \forall \alpha :: (\{age\}, \mathcal{L}; \{age : int\}) \triangleright \alpha \rightarrow int$

$l = [\{name = \text{"Jacques"}, age = 36\}, \{name = \text{"Hugo"}, weight = 16\}]$

$l : \forall \alpha :: (\emptyset, \{name\}; \{name : string, age : int, weight : int\}) \triangleright \alpha \text{ list}$

$nats = \text{fun } n \rightarrow \{hd = n, tl = nats (n + 1)\}$

$nats : \forall \alpha :: (\emptyset, \{hd, tl\}; \{hd : int, tl : \alpha\}) \triangleright int \rightarrow \alpha$

## Constraint domain

---

A set of abstract constraints  $\mathcal{C}$  with entailment  $\models$

- $\perp \in \mathcal{C}$  such that  $\forall C. \perp \models C$  and  $C \models \perp$  decidable
- $\models$  reflexive and transitive
- for any  $C$  and  $C'$ ,  $C \wedge C'$  is the weakest constraint entailing both  $C$  and  $C'$

Observations  $C \vdash p(a)$  ( $a$  a symbol) compatible with entailment

Relating predicates  $r(a, T)$

Propagation rules of the form:

$$\forall x. (r(x, \alpha_1) \wedge r(x, \alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2)$$

## Admissible substitution

---

$K \vdash \theta : K'$  ( $\theta$  admissible), if for all  $\alpha :: (C, R)$  in  $K$ ,  $\theta(\alpha)$  is a type variable  $\alpha'$  and it satisfies the following properties.

1.  $\alpha' :: (C', R') \in K'$  *keep kinding*
2.  $C' \models C$  *entailment of constraints*
3.  $\theta(R) \subseteq R'$  *keep types*

Every  $C$  in  $K'$  shall be valid, and  $R$  satisfy propagation.

## Typing rules

---

Variable

$$\frac{K, K_0 \vdash \theta : K \quad \text{Dom}(\theta) \subset B}{K; E, x : \forall B. K_0 \triangleright T \vdash x : \theta(T)}$$

Abstraction

$$\frac{K; E, x : T \vdash e : T'}{K; E \vdash \text{fun } x \rightarrow e : T \rightarrow T'}$$

Application

$$\frac{K; E \vdash e_1 : T \rightarrow T' \quad K; E \vdash e_2 : T}{K; E \vdash e_1 e_2 : T'}$$

Generalize

$$\frac{K; E \vdash e : T \quad B \cap \text{FV}_K(E) = \emptyset}{K|_{\overline{B}}; E \vdash e : \forall B. K|_B \triangleright T}$$

Let

$$\frac{K; E \vdash e_1 : \sigma \quad K; E, x : \sigma \vdash e_2 : T}{K; E \vdash \text{let } x = e_1 \text{ in } e_2 : T}$$

Constant

$$\frac{K_0 \vdash \theta : K \quad \text{Tconst}(c) = K_0 \triangleright T}{K; E \vdash c : \theta(T)}$$

## Typing rules (co-finite)

---

Variable

$$\frac{K, K_0^{\bar{\alpha}} \vdash \theta : K \quad \text{Dom}(\theta) = \bar{\alpha}}{K; E, x : K_0 \triangleright T \vdash x : T^{\theta(\bar{\alpha})}}$$

Abstraction

$$\frac{\forall x \notin L \quad K; E, x : T \vdash e^x : T'}{K; E \vdash \lambda e : T \rightarrow T'}$$

Application

$$\frac{K; E \vdash e_1 : T \rightarrow T' \quad K; E \vdash e_2 : T}{K; E \vdash e_1 e_2 : T'}$$

Generalize

$$\frac{\forall \bar{\alpha} \notin L \quad K, K_0^{\bar{\alpha}}; E \vdash e : T^{\bar{\alpha}}}{K; E \vdash e : K_0 \triangleright T}$$

Let

$$\frac{\forall x \notin L \quad K; E \vdash e_1 : \sigma \quad K; E, x : \sigma \vdash e_2^x : T}{K; E \vdash \text{let } e_1 \text{ in } e_2 : T}$$

Constant

$$\frac{K_0^{\bar{\alpha}} \vdash \theta : K \quad \text{Tconst}(c) = K_0 \triangleright T}{K; E \vdash c : T^{\theta(\bar{\alpha})}}$$

## Differences with proof for ML

---

Modifications are massive

- from a total of 970 lines for Core ML, 259 were modified and 1290 were added (excluding the 653 lines for instance domain proofs)
- the main technical modification was converting iterated substitutions into simultaneous ones
- used signatures and functors to allow instantiating the framework
- allowed adding constants and delta-rules modularly

Following the original proof plan, framework proofs were straightforward. Instance domain proofs were harder.

## Simultaneous substitution

---

While opening local variables in a type scheme was already a simultaneous operation, global variable substitution was incremental.

```
Lemma typ_subst_open : forall (X:var) (U T:typ) (Ts:list typ),
  type U ->
  typ_subst X U (typ_open T Ts) =
  typ_open (typ_subst X U T) (List.map (typ_subst X U) Ts).
```

```
Lemma typ_subst_open : forall (S:env typ) (T:typ) (Ts:list typ),
  env_prop type S ->
  typ_subst S (typ_open T Ts) =
  typ_open (typ_subst S T) (List.map (typ_subst S) Ts).
```

The change was easy, using the environment type, but dozens of lemmas required modification.

## Kinding proofs

---

**Definition** `well_subst` (K K':env kind) (S:env typ) :=  
`forall` (Z:var) (k:kind), binds Z k K ->  
`well_kinded` K' (kind\_subst S k) (typ\_subst S (typ\_fvar Z)).

**Lemma** `well_kinded_subst`: `forall` S K K' k T,  
`well_subst` K K' S ->  
`well_kinded` K k T ->  
`well_kinded` K' (kind\_subst S k) (typ\_subst S T).

**Lemma** `well_subst_fresh` : `forall` K K' K'' S Ys L1 M,  
`well_subst` (K & K' & K'') (K & map (kind\_subst S) K'') S ->  
`fresh` (L1 ∪ dom S ∪ dom (K & K'')) (length K0) Ys ->  
`well_subst` (K & K' & K'' & kinds\_open\_vars K0 Ys)  
(K & map (kind\_subst S) (K'' & kinds\_open\_vars K0 Ys)) S.

# Constraint domain and constants

---

```
Module Type CstrIntf.  
  Parameter cstr : Set.  
  Parameter valid : cstr -> Prop.  
  Parameter entails : cstr -> cstr -> Prop.  
  Parameter entails_refl : forall c, entails c c.  
  Parameter entails_trans : forall c1 c2 c3,  
    entails c1 c2 -> entails c2 c3 -> entails c1 c3.  
  Parameter unique : cstr -> var -> Prop.  
End CstrIntf.
```

```
Module Type CstIntf.  
  Parameter const : Set.  
  Parameter arity : const -> nat.  
End CstIntf.
```

## Modularity and delta-rules

---

```
Module MkDefs(Cstr:CstrIntf)(Const:CstIntf).
  ...
  Module Type DeltaIntf.
    Parameter type : Const.const -> sch.
    Parameter rule : nat -> trm -> trm -> Prop.
    Parameter term : forall n t1 t2 t1,
      rule n t1 t2 -> list_for_n term n t1 ->
      term (trm_inst t1 t1) / term (trm_inst t2 t1).
  End DeltaIntf.
  Module MkJudge(Delta:DeltaIntf).
    Inductive typing : kenv -> env -> trm -> typ -> Prop := ...
    Inductive value : nat -> trm -> Prop := ...
    Inductive red : trm -> trm -> Prop := ...
  End MkJudge.
End MkDef.
```

# Soundness proof

---

```
Module MkSound(Cstr:CstrIntf)(Const:CstIntf).
  Module Infra := MkInfra(Cstr)(Const).
  Import Infra Defs.
  Module Mk2(Delta:DeltaIntf).
    Module JudgInfra := MkJudgInfra(Delta).
    Import JudgInfra Judge.
    Module Type SndHypIntf.
      Parameter const_closed : forall c, sch_fv (Delta.type c) = {}.
      Parameter delta_typed : forall n t1 t2 t1 K E T, ...
      Parameter const_arity_ok : forall c v1 K T, ...
      Parameter delta_arity : forall n t1 t2, ...
    End SndHypIntf.
  Module Mk3(SH:SndHypIntf).
    ...
```

## Soundness results

---

**Lemma preservation** : forall K E t t' T,  
K ; E |= t ~: T ->  
t --> t' ->  
K ; E |= t' ~: T.

**Lemma progress** := forall K t T,  
K ; empty |= t ~: T ->  
value t  
∨ exists t', t --> t'.

**Lemma value\_irreducible** : forall n t t',  
value n t -> ~(t --> t').

## Constraint domain proofs

---

```
Module Cstr. ... End Cstr.
Module Const. ... End Const.
Module Sound1 := MkSound(Cstr)(Const).
Import Sound1 Infra Defs.

Module Delta. ... End Delta.
Module Sound2 := Mk2(Delta).
Import Sound2 JudgInfra Judge.

Module SndHyp.
  ... ..
End SndHyp.
Module Soundness := Mk3(SndHyp).
```

## Conclusion

---

Proved soundness of structural polymorphism, including the constraint domain for polymorphic variants and records.

- reusing proofs and tactics was a tremendous help
- recursive types were not a problem at all!
- using functors for constructing the framework did work, but it is heavy

## Adding a non-structural rule

---

Kind GC

$$\frac{\forall \bar{\alpha} \notin L \quad K, K_0^{\bar{\alpha}}; E \vdash e : T}{K; E \vdash e : T}$$

- framework proofs are still easy
- domain proofs become much harder:  
inversion no longer works directly
- in some cases, a renaming lemma could help, but  
much harder to prove than what they claim...