Soundness and principality of type inference for structural polymorphism

TPP'08 2008.11.26

Jacques Garrigue

Nagoya University

http://www.math.nagoya-u.ac.jp/~garrigue/papers/

Structural polymorphism [FOOL02]

A typing framework for polymorphic variants and records

- faithful description of the core of OCaml
- polymorphism is described by local constraints
- constraints may be recursive;
  they are kept in a recursive kinding environment
- constraints are abstract, and constraint domains with their δ-rules can be defined independently
- the paper only proves completeness of unification;
  it is assumed to be sufficient for inference

What I have been doing

Proved type soundness in Coq.

- last year's TPP
- proof is based on "Engineering formal metatheory"

Proved soundness and principality of type inference.

- extends the above proof of type soundness
- inference algorithm can be extracted and run as ocaml code

Synopsis

- Structural polymorphism
- Types and kinds
- Constraint domains
- Typing rules
- *"Engineering formal metatheory"*
- Type soundness
- Soundness and completeness of unification
- Soundness and principality of inference
- Using the algorithm.
- Concluding remarks

Types and kinds

Types are mixed with kinds in a mutually recursive way.

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

Type judgments contain both a type and a kinding environment.

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

Example: polymorphic variants

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

$$Number(5) \ : \ \alpha :: (\{Number\}, \mathcal{L}; \{Number : int\}) \triangleright \alpha$$

$$l_2 = [Number(5), Face(\texttt{"King"})]$$
$$l_2 \ : \ \alpha :: (\{Number, Face\}, \mathcal{L}; \{Number : int, Face : string\}) \triangleright \alpha \ list$$

$$length = \text{function } Nil() \rightarrow 0 \mid Cons(a, l) \rightarrow 1 + length \ l$$
$$length \ : \ \alpha :: (\emptyset, \{Nil, Cons\}; \{Nil : unit, Cons : \beta \times \alpha\}) \triangleright \alpha \rightarrow int$$

$$length' = \text{function } Nil() \rightarrow 0 \mid Cons(l) \rightarrow 1 + length \ l$$
$$length' \ : \ \alpha :: (\emptyset, \{Nil, Cons\}; \{Nil : unit, Cons : \alpha\}) \triangleright \alpha \rightarrow int$$

$$f \ l = length \ l + length2 \ l$$
$$f \ : \ \alpha :: (\emptyset, \{Nil, Cons\}; \{Nil : unit, Cons : \beta \times \alpha, Cons : \alpha\}) \triangleright \alpha \rightarrow int$$

Constraint domain

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

- $\bot \in \mathcal{C}$ such that $\forall C.\bot \models C$ and $C \models \bot$ 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 $r(a, T)$ with propagation rules of the form:

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

Typed constants and δ-rules, which should satisfy subject reduction

Admissible substitution

$K \vdash \theta : K'$ (θ admissible), if for all $\alpha :: (C, R)$ in K, $\theta(\alpha)$ is a type variable α' 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

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

Abstraction

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

Application

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

Generalize

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

Let

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

Constant

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

Engineering formal metatheory [POPL08]

Aydemir, Charguéraud, Pierce, Weirich

Soundness for various type systems ($F_{\leq}$, ML, CoC)
Two main ideas to avoid renaming:

- Locally nameless definitions
  Use de-bruijn indices inside terms and types,
  but named variables for environments.
- Co-finite quantification
  Allows reuse of derivations in different contexts.

Typing rules (co-finite)

Variable

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

Abstraction

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

Application

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

Generalize

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

Let

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

Constant

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

Equivalence of finite and co-finite quantif.

While the co-finite approach may not be very intuitive, it is easy to see that one can build a finite derivation from a co-finite one (you just have to pick variables)

The opposite tranformation requires renaming lemmas, for terms and types. In "Engineering metatheory" it is claimed that these lemmas can be built from the substitution lemmas, but in this system there is still lots of work to obtain them.

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').
```

Adding a non-structural rule

Kind GC

$$\frac{K, K'; E \vdash e : T \quad \mathsf{FV}_K(E, T) \cap \mathsf{Dom}(K') = \emptyset}{K; E \vdash e : T}$$

Co-finite version

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

– Formalizes the intuition that kinds not appearing in either E or T are not relevant to the typing judgment

– The original type system requires all kinds used in a derivation to be in K from the beginning

Looking for an inversion lemma

For domain proofs, we would like to prove the following lemma:

$$K; E \vdash_{GC} e : T \Rightarrow \exists K',\ K, K'; E \vdash e : T$$

The proof in the co-finite system is difficult, as co-finite quantifications do not commute. The proof is more than 1300 lines, with renaming lemmas for terms and types.

A much simpler approach is to only limit Kind GC to appear just above Let or Abstraction. The proof is about 100 lines, and requires no renaming.

Type inference

Type inference can be done in the usual way:

- W-like algorithm relying on type unification
- Unification updates the kinding environment

  `kenv -> subs -> list (typ*typ) -> option (kenv*subs)`
- All substitutions must be admissible
- Termination measure $\langle number\ of\ variables, size\ of\ types \rangle$

Unification (abstract version)

Incompatible

$$\frac{\varphi \wedge T_1 \doteq T_2}{\bot} \text{ when } sort_\varphi(T_1) \neq sort_\varphi(T_2)$$

Redundancy

$$\frac{\varphi \wedge T \doteq T}{\varphi}$$

Cyclic

$$\frac{\varphi \wedge \alpha \doteq T}{\bot} \text{ when } \alpha \neq T \text{ and } \alpha \in \mathsf{FV}_\emptyset(T)$$

Function

$$\frac{\varphi \wedge T_1 \to T_2 \doteq T_1' \to T_2'}{\varphi \wedge T_1 \doteq T_1' \wedge T_2 \doteq T_2'}$$

Substitution

$$\frac{\varphi \wedge \alpha \doteq T}{\varphi[T/\alpha] \wedge \alpha \doteq T} \text{ when } \alpha :: (C,R) \notin \varphi \text{ and } \alpha \notin \mathsf{FV}_\emptyset(T) \text{ and } \alpha \in \mathsf{FV}(\varphi) \text{ and } T \neq \beta \vee \beta \in \mathsf{FV}(\varphi)$$

Bad constraint

$$\frac{\varphi \wedge \alpha :: (C,R)}{\bot} \text{ when } C \models \bot$$

Constraint

$$\frac{\varphi \wedge \alpha_1 :: (C_1,R_1) \wedge \alpha_2 :: (C_2,R_2) \wedge \alpha_1 \doteq \alpha_2}{\varphi \wedge \alpha :: (C_1 \wedge C_2, R_1 \cup R_2) \wedge \alpha_1 \doteq \alpha \wedge \alpha_2 \doteq \alpha} \ \alpha \text{ fresh}$$

Propagation

$$r(x,\alpha_1) \wedge r(x,\alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2 \in \mathcal{E}$$

$$\frac{\varphi \wedge \alpha :: (C,R) \quad r(a,T_1) \in R \quad r(a,T_2) \in R \quad C \vdash p(a)}{\varphi \wedge \alpha :: (C,R) \wedge T_1 \doteq T_2} \text{ when } T_1 \neq T_2$$

```
Fixpoint unify0 unify (h:nat)(pairs:list(typ*typ))(K:kenv)(S:subs)
  {struct h} : option (kenv * subs) :=
  match h with 0 => None
  | S h' =>
    match pairs with nil => Some (K,S)
    | (typ_fvar x, typ_fvar y) :: pairs' =>
      if x == y then unify0 unify h' pairs' K S else
      match unify_vars K x y with None => None
      | Some (K', pairs) =>
        unify (pairs ++ pairs') K' (compose (x ~ typ_fvar y) S)
      end
    | ...
  end end.
Fixpoint unify (h:nat) pairs K S {struct h} :=
  match h with 0 => None
  | S h' =>
    unify0 (unify h')(pairs_size pairs+1)(subst_pairs S pairs) K S
  end.
```

Properties of unification

$\mathrm{Dom}(S) \cap \mathrm{FV}_\emptyset(\mathrm{Img}(S)) = \emptyset$

```
Theorem unify_sound : forall h pairs K S K' S',
  unify h pairs K S = Some (K',S') ->
  is_subst S -> disjoint (dom S) (dom K) ->
  unifies S' pairs ∧ well_subst K K' S' ∧
  extends S' S ∧ disjoint (dom S') (dom K').
Theorem unify_mgu : forall h pairs K0 K S,
  unifies S' pairs ->
  well_subst K0 K' S' ->
  unify h pairs K0 id = Some (K,S) ->
  extends S' S ∧ well_subst K K' S'.
Theorem unify_complete : forall h pairs K0 K S,
  unifies S pairs ->
  well_subst K0 K S ->
  cardinal (all_fv K0 pairs) < h ->
  unify h pairs K0 id <> None.
```

$K \vdash S' : K'$

Type inference

```
(* L is a set of used variables, for fresh variable generation *)
Fixpoint typinf (K:kenv) (E:env sch) (t:trm) (T:typ) (L:vars)
  (S:subs) (h:nat) {struct h} : option (kenv * subs) * vars := ...

(* Simpler version, inferring the most general scheme of a term *)
Definition typinf' trm :=
  let v  :=  var_fresh {} in
  let Lv := S.singleton v in
  let V := typ_fvar v in
  match typinf empty empty trm V Lv empty (trm_depth trm + 1)
  with (None, _) => None
  | (Some (K, S), _) =>
    Some (map (kind_subst S) K, typ_subst S V)
  end.
```

Generalization

```
Definition typinf_generalize K' E' L T1 :=
  (* Closure of variables free in the environment *)
  let ftve := close_fvk K' (env_fv E') in
  let (K'', KA) := split_env ftve K' in
  (* Closure of variables free in the result type *)
  let B := close_fvk K' (typ_fv T1) in
  (* Keep variables in B but not in ftve *)
  let (_, KB) := split_env B K'' in
  let (Bs, Ks) := split KB in
  let Bs' := S.elements (S.diff B (ftve ∪ dom KB)) in
  let Ks' := List.map (fun x:var => @None ckind) Bs' in
  (* Duplicate kinds imported from the original environment *)
  let (_, KC) := split_env L K'' in
  (KA & KC, sch_generalize (Bs++Bs') T1 (Ks++Ks')).
```

Soundness of type inference

```
Theorem typinf_sound : forall h t K0 E T L0 S0 K S L,
  typinf K0 E t T L0 S0 h = (Some (K, S), L) ->
  is_subst S0 -> env_prop type S0 ->
  kenv_ok K0 -> disjoint (dom S0) (dom K0) ->
  fvs S0 K0 E ∪ typ_fv T << L0 ->
  env_ok E -> type T ->
  extends S S0 ∧ env_prop type S ∧ is_subst S ∧
  disjoint (dom S) (dom K) ∧ fvs S K E ∪ L0 << L ∧
  well_subst K0 K S ∧
  K; map (sch_subst S) E |(false,GcLet)|= t ~: typ_subst S T.

Corollary typinf_sound' : forall t K T,
  typinf' t = Some (K, T) -> K; empty |(false,GcLet)|= t ~: T.
```

Principality of type inference

```
Definition principality : forall S0 K0 E0 S K E t T L h,
  is_subst S0 -> env_prop type S0 ->
  kenv_ok K0 -> disjoint (dom S0) (dom K0) ->
  env_ok E0 -> moregen_env K (map (sch_subst S) E0) E ->
  env_prop type S -> dom S ∪ fvs S0 K0 E0 ∪ typ_fv T << L ->
  extends S S0 -> well_subst K0 K S -> trm_depth t < h ->
  K; E |(false,GcAny)|= t ~: typ_subst S T ->
  exists K', exists S', exists L',
    typinf K0 E0 t T L S0 h = (Some (K', S'), L') ∧ extends S' S0 ∧
    exists S'', dom S'' << S.diff L' L ∧ env_prop type S'' ∧
      extends (S & S'') S' ∧ well_subst K' K (S & S'').
Corollary typinf_principal' : forall K t T,
  K; empty |(false,GcAny)|= t ~: T ->
  exists K', exists T', typinf' t = Some (K', T') ∧
    exists S, well_subst K' K S ∧ T = typ_subst S T'.
```

About the proofs

Proofs are large: about 2000 lines for unification, 3500 lines for type inference, including many small lemmas.

They work directly on the algorithms, but some functional induction schemes are defined (`unify_ind` for `unify`, `soundness_ind` for `typinf`).

The main difficulty is having to maintain many invariants simultaneously.

Use finite sets, but the library has only a bare minimum of lemmas.

Lots of proofs are about set invariants. Developed some tactics for set inclusion and disjointness that helped a lot.

Impact of locally nameless and co-finite

Since local and global variables are distinct, many definitions must be duplicated, and we need lemmas to connect them.

- This is particularly painful for kinding environments, as they are recursive.
- Yet having to handle explicitly names of bound type variables would probably be even more painful.

Co-finite approach seems to be always a boon. Even for type inference, only few proofs use renaming lemmas:

- principality only requires term variable renaming once.
- soundness requires both term and value variables renaming, not surprising since we build a co-finite proof from a finite one.

Dependent types in values

They are used in the "engineering metatheory" framework only when generating fresh variables:

```
Lemma var_fresh : forall (L : vars), { x : var | x ∉ L }.
```

I used dependent types in values in one other place: all kinds are valid and coherent by construction.

- A bit more complexity in domain proofs.
- But a big win since this property is kept by substitution.

```
Record ckind : Set := Kind {
  kcstr : Cstr.cstr;
  kvalid : Cstr.valid kcstr;
  krel  : list (var*typ);
  kcoherent : coherent kcstr krel }.
```

Also attempted to use dependent types for schemes (enforcing that they are well-formed), but dropped them as it made proofs about the type inference algorithm more complex.

Instantiating the framework

The Coq proof uses functors to reflect faithfully constraint domain parameterization.

The final proof is obtained by applying all functors to domain proofs. The domain definitions and proofs are $\sim$ 800 lines, mostly for δ-rules. Only 50 extra lines were needed for type inference.

Once the framework is instantiated, one can extract the type inference algorithm to ocaml, and run it.

Using the algorithm

```
(* This example is equivalent to the ocaml term [fun x -> `A0 x] *)
# typinf1 (Coq_trm_cst (Const.Coq_tag (Variables.var_of_nat 0)));;
- : (var * kind) list * typ =
([(1, None);
  (2,
   Some
    {kind_cstr = {cstr_low = {0}; cstr_high = None};
     kind_rel = Cons (Pair (0, Coq_typ_fvar 1), Nil)})],
 Coq_typ_arrow (Coq_typ_fvar 1, Coq_typ_fvar 2))
```

Conclusion

– Formalized completely strutural polymorphism

– Proved not only type soundness, but also soundness and principality of inference

– First step towards a certified reference implementation of OCaml

– The techniques in "engineering formal metatheory" proved useful

Locally nameless definitions

- α-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.

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

Co-finite quantification

- we need to change non-locally bound names

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

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

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

Example with weakening

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

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

No renaming needed if we enlarge L !

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

Co-finite quantification and ML let

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

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

becomes

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

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 explictly consider only relevant bindings.

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

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

Engineering formal metatheory

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. Soundess 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.