A Certified Interpreter for ML with Structural Polymorphism

Jacques Garrigue
Nagoya University

http://www.math.nagoya-u.ac.jp/~garrig
What’s in OCaml’s type system

– Core ML with relaxed value restriction
– Recursive types
– Polymorphic objects and variants
– Structural subtyping (with variance annotations)
– Modules and applicative functors
– Private types: private datatypes, rows and abbreviations
– Recursive modules . . .
What are the guarantees?

Proved before (by many)

- Type soundness and principality of type inference for various subsets (by hand).
- Mechanical proof of type soundness for the core OCaml-Light project (without the relaxed value restriction).

What I have done in Coq over the last 2 years

- A certified interpreter for ML with structural polymorphism.
- Includes type soundness and principality of inference.
- Covers polymorphic objects and variants, with recursion.
- Mechanization is based on “Engineering formal...”
A typing framework for polymorphic variants:

- Faithful description of the core of OCaml.
- Polymorphism is described by local constraints.
- Constraints are kept in a recursive kinding environment.
- Constraints are abstract, and constraint domains and their $\delta$-rules can be defined independently.
Types and kinds

Types are mixed with kinds in a mutually recursive way:

\[ T ::= \alpha \quad \text{type variable} \]
\[ \quad | \quad T \rightarrow T \quad \text{function type} \]
\[ \sigma ::= \forall \alpha. K \triangleright T \quad \text{polytypes} \]
\[ K ::= \emptyset | K, \alpha :: \kappa \quad \text{kinding environment} \]
\[ \kappa ::= \bullet | (C; R) \quad \text{kind} \]
\[ R ::= \{a : T, \ldots\} \quad \text{relation set} \]

Type judgments contain both a type and a kinding environment:

\[ K; E \vdash e : T \]
Kinds have the form \((L, U; R)\), such that \(L \subseteq U\).

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

\(l_2 = [\text{Number}(5), \text{Face}("King")])\)

\(l_2 : \alpha :: (\{Number, Face\}, L; \{Number : int, \text{Face : string}\})\)

\(\text{length} = \text{function } \text{Nil}() \to 0 | \text{Cons}(a, l) \to 1 + \text{length } l\)

\(\text{length} : \alpha :: (\emptyset, \{\text{Nil, Cons}\}; \{\text{Nil : unit, Cons : } \beta \times \alpha\})\)

\(\text{length}' = \text{function } \text{Nil}() \to 0 | \text{Cons}(l) \to 1 + \text{length' } l\)

\(\text{length}' : \alpha :: (\emptyset, \{\text{Nil, Cons}\}; \{\text{Nil : unit, Cons : } \alpha\}) \triangleright \alpha\)

\(f \ l = \text{length } l + \text{length' } l\)

\(f : \alpha :: (\emptyset, \{\text{Nil, Cons}\}; \{\text{Nil : unit, Cons : } \beta \times \alpha, \text{Cons : } \alpha\})\)
Typing rules

Variable
\[ K, K_0 \vdash \theta : K \quad \text{dom}(\theta) \subseteq B \]
\[ K; E, x : \forall B. K_0 \supset T \vdash x : \theta(T) \]

Abstraction
\[ K; E, x : T \vdash e : T' \]
\[ K; E \vdash \text{fun } x \to e : T \to T' \]

Application
\[ K; E \vdash e_1 : T \to T' \quad K; E \vdash e_2 : T \]
\[ K; E \vdash e_1. e_2 : T' \]

Generalize
\[ K; E \vdash e : T \quad B \cap \text{dom}(\theta) \neq \emptyset \]
\[ K|_B; E \vdash e : \forall B. K_0 \]

Let
\[ K; E \vdash e_1 : \sigma \quad K; E \vdash e : \forall B. K_0 \]
\[ K; E \vdash \text{let } x = e_1 \in e : T' \]

Constant
\[ K_0 \vdash \theta : K \quad \text{type} \]
\[ K; E \vdash c : \theta(T) \]

\[ K_0 \vdash \theta : K \text{ iff } \alpha :: \kappa \in K_0 \text{ implies } \theta(\alpha) :: \kappa' \in K \text{ and} \]

\[ K_0 \vdash \theta : K \text{ iff } \alpha :: \kappa \in K_0 \text{ implies } \theta(\alpha) :: \kappa' \in K \text{ and} \]
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**
  Variables local to a branch are quantified universally.
  This allows reuse of derivations in different contexts.

Formalization is not always intuitive, but streamlines proofs of type soundness.
Typing rules (co-finite)

Variable
\[ K \vdash \overline{T} :: \overline{\kappa}^T \]
\[ K; E, x : \overline{\kappa} \not\in T_1 \vdash x : T_1^\overline{T} \]

Abstraction
\[ \forall x \not\in L \quad K; E, x : T \vdash e^x : T' \]
\[ K; E \vdash \lambda e : T \rightarrow T' \]

Application
\[ K; E \vdash e_1 : T \rightarrow T' \quad K; E \vdash e_2 : T \]
\[ K; E \vdash e_1 \ e_2 : T' \]

Generalize
\[ \forall \overline{\alpha} \not\in L \quad K, \overline{\alpha} :: \overline{\kappa}^\overline{\alpha} \]
\[ K; E \vdash e : \overline{\kappa} \not\in T \]

Let
\[ K; E \vdash e_1 : \sigma \quad K; E \]
\[ K; E \vdash \text{let } e_1 \text{ in } e_2 \]

Constant
\[ K \vdash \overline{T} :: \overline{\kappa}^T \quad \text{Tco} \]
\[ K; E \vdash c : T_1^\overline{T} \]

\[ K \vdash \alpha :: \kappa \quad \text{when } \alpha :: \kappa' \in K \text{ and } \kappa' \models \]
\[ K \vdash T :: \bullet \quad \text{always} \]
started from *Engineering formal metatheory* ML proof, with many modifications to accommodate mutual recursion. No renaming needed for soundness!

**Lemma preservation**: \( \forall K E e e' T, \)
\[
K ; E \models e : T \rightarrow \\
e \rightarrow e' \rightarrow \\
K ; E \models e' : T.
\]

**Lemma progress**: \( \forall K e T, \)
\[
K ; \text{empty} \models e : T \rightarrow \\
\text{value } e \lor \exists e', e \rightarrow e'.
\]

**Lemma value_irreducible**: \( \forall e e', \)
\[
\text{value } e \rightarrow \neg (e \rightarrow e').
\]
Need **simultaneous substitutions** rather than iterated.

As a consequence, freshness of sequences of variables is insufficient, and we need **disjointness conditions** ($L_1 \setminus L_2$).

Also added a framework for **constants and δ-rules**.

Overall **size just doubled**, with no significant jump in complexity.

This does not include:

- **Additions to the metatheory**, with tactics for finite set inclusion, disjointness, etc... (1300 lines)

- **Domain proofs**, for concrete constraints and constants
Constraint domain proofs

Instantiation of the framework to a constraint domain results in the following “dialog”. This was done for the domain of variants and records.

Module Cstr. (* Define constraints *) End Cstr.
Module Const. (* Constants and arities *) End Const.
Module Sound1 := MkSound(Cstr)(Const).
Import Sound1 Infra Defs.

Module Delta. (* Constant types and delta-rules *)
Module Sound2 := Mk2(Delta).
Import Sound2 JudgInfra Judge.

Module SndHyp. (* Domain proofs *) End SndHyp.
Module Soundness := Mk3(SndHyp).
Adding a non-structural rule

Kind GC

\[ \text{FV}_K(E, T) \cap \text{dom}(K') = \emptyset \]

\[ K, K'; E \vdash e : T \]

\[ K; E \vdash e : T \]

- Formalizes the intuition that kinds not appearing in \( E \) or \( T \) are not relevant to the typing judgment.

- Good for modularity.

- Not derivable in the original type system, as all derivation must be in \( K \) from the beginning.

- Again, the co-finite version is implicit.

Cofinite Kind GC

\[ \forall \bar{\alpha} \notin L \]

\[ K, \bar{\alpha} :: \bar{\kappa} \bar{\alpha}; \]

\[ K; E \vdash_{GC} \]
Framework proofs are still easy (induction on derivations), but domain proofs become much harder (inversion no longer works).

One would like to prove the following lemma:

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

I got completely stuck in the co-finite system, as co-finite quantification in Generalize does not commute with Kind GC.

I could finally prove it in more than 1300 lines, including renaming lemmas for both terms and types.

Afterwards, I realized that I only needed canonicization of proofs, which is only 100 lines, as it does not require renaming.
Type inference

Type inference is done in the usual ML way:

- **W-like algorithm** relying on type unification.
- All functions return both a **normalized substitution** and an updated **kinding environment**.
- Statements of inductive theorems become much more complex.
- Simpler statements as corrolary.
- **Renaming lemmas** are needed.
Unification

Formal proofs in LCF by Paulson as early as 1985.

Here we also need to handle the kinding environment, making the algorithm much more complicated.

Rather than $\theta$ is more general than $\theta'$ ($\exists \theta_1, \theta' = \theta_1 \circ \theta$), simpler $\theta'$ extends $\theta$ ($\theta' \circ \theta = \theta'$). They are equivalent when $\theta$ is idempotent.

900 lines for definitions and soundness, thanks to an induction lemma exploiting symmetries. 1000 more lines for completeness, with a large part for termination.
Type inference

For core ML, W’s correctness was proved about 10 years ago, both in Isabelle and Coq.

The original paper on type inference structural polymorphism contained only proofs about unification.

The practical type inference algorithm is very complex due to subtleties of generalize.

Both soundness and principality require renaming. Standard generalize renames type variables twice!

More than 3000 lines of proof, with lots of lemmas about free variables.
generalize\((K, E, T, L)\) = 
let \(A = FV_K(E)\) and \(B = FV_K(T)\) in
let \(K' = K|_A\) in let \(\bar{\alpha} :: \bar{\kappa} = K'|_B\) in
let \(\{\bar{\alpha}'\} = B \setminus (A \cup \{\bar{\alpha}\})\) in let \(\bar{\kappa}' = \text{map} (\lambda_\cdot) \bar{\alpha}'\) in
\(\langle(K|_A, K'|_L), [\bar{\alpha}\bar{\alpha}'][\bar{\kappa}\bar{\kappa}']_T\rangle\)

\(\text{typinf}(K, E, \text{let } e_1 \text{ in } e_2, T, \theta, L) = \)
let \(\alpha = \text{fresh}(L)\) in
match \(\text{typinf}(K, E, e_1, \alpha, \theta, L \cup \{\alpha\})\) with
| \(\langle K', \theta', L' \rangle \Rightarrow \)
  let \(\langle K'', \sigma \rangle = \text{generalize}(\theta'(K'), \theta'(E), \theta'(T), \theta'(\text{dom}(K')))\) in
  let \(x = \text{fresh}(\text{dom}(E) \cup \text{FV}(e_1) \cup \text{FV}(e_2))\) in
  \(\text{typinf}(K'', (E, x : \sigma), e^x_2, T, \theta', L')\)
| \(\langle \rangle \Rightarrow \langle \rangle\)
Properties of type inference

### Soundness

\[ \text{typinf}'(E, e) = \langle K, T \rangle \rightarrow FV(E) = \emptyset \rightarrow K; E \vdash e : T \]

\[ \text{typinf}(K, E, e, T, \theta, L) = \langle K', \theta', L' \rangle \rightarrow \]

\[ \text{dom}(\theta) \cap \text{dom}(K) = \emptyset \rightarrow FV(\theta, K, E, T) \subset L \rightarrow \]

\[ \theta'(K'); \theta'(E) \vdash e : \theta'(T) \land \theta' \sqsubseteq \theta \land K \vdash \theta': \theta'(K') \land \]

\[ \text{dom}(\theta') \cap \text{dom}(K') = \emptyset \land FV(\theta', K', E) \cup L \subset L' \]

### Principality

\[ K; E \vdash e : T \rightarrow FV(E) = \emptyset \rightarrow \exists K'T', \text{typinf}'(E, e) = \langle K', T' \rangle \land \exists \theta, T = \theta(T') \land K' \vdash e : T \]

\[ K; E \vdash e : \theta(T) \rightarrow K \vdash \theta(E_1) \leq E \rightarrow \theta \sqsubseteq \theta_1 \rightarrow K_1 \vdash \theta \land \]

\[ \text{dom}(\theta_1) \cap \text{dom}(K_1) = \emptyset \rightarrow \text{dom}(\theta) \cup FV(\theta_1, K_1, E_1, T, \theta_1, L) \rightarrow \]

\[ \exists K'\theta' L', \text{typinf}(K_1, E_1, e, T, \theta_1, L) = \langle K', \theta', L' \rangle \land \]

\[ \exists \theta'', \theta\theta'' \sqsubseteq \theta' \land K' \vdash \theta\theta' : K \land \text{dom}(\theta'') \subset L' \setminus L \]
Defined a stack based abstract machine. Since variables are de Bruijn indices, we can use terms as code.

Theorem eval_sound_rec :
\[ \forall (h:nat) (fl:list frame) (benv args:list clos)
\text{closed_n (length benv) t} \rightarrow
K ; E |= \text{stack2trm (app2trm (inst t benv) args)} \rightarrow
K ; E |= \text{res2trm (eval fenv h benv args t fl)} \rightarrow T. \]

Theorem eval_complete :
\[ \forall K t t' T,
K ; E |= t \rightarrow T \rightarrow
clos_refl_trans_1n _ red t t' \rightarrow \text{value t'} \rightarrow
\exists h : nat, \exists cl : clos,
\text{eval fenv h [] [] t [] = Result 0 cl \wedge t' = cl}. \]
Impact of locally nameless and co-finite

Since local and global variables are distinct, many definitions need to 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 in type inference, only few proofs use renaming lemmas:

- principality only requires term variable renaming.

- soundness requires both term and type variables renaming, surprising since we build a co-finite proof from a
Dependent types in values

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

Lemma \text{var\_fresh} : \forall L : \text{vars}, \{ x : \text{var} \mid x \not\in 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.

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

Some proofs are still much bigger than expected: evaluation, type inference, . . .

- The value predicate is complex, as it handles constant arity. It might have been better to define constants as n-ary constructors from the start. This would require writing the induction principles by hand; already done for closures.

- Using functions to represent algorithms is dirty. In some cases, adding input-output inductive relations helped, but in general it does not change the proof size significantly.

- Still wondering...
Using the algorithm

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

(* This example is equivalent to the ocaml term \([\text{fun} \ x \rightarrow 'A0 \ x]\) *)

```ocaml
# typinf1 (Coq_trm_cst (Const.Coq_tag (Variables.var_of_nat O)));;
- : (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))
```


# let rev_append =
recf (abs (abs (abs
  (matches [0;1] [abs (bvar 1);
    abs(apps(bvar 3)[sub 1 (bvar 0);cons(sub 0 (bvar 0);bvar 1)]))))));

val rev_append : trm = ...

# typinf2 Nil rev_append;;

- : (var * kind) list * typ =
  ([[10, <Ksum, {}], {0; 1}, {0 => tv 15; 1 => tv 34}];
   (29, <Ksum, {1}, any, {1 => tv 26}>);
   (34, <Kprod, {1; 0}, any, {0 => tv 30; 1 => tv 10}>;
    (26, <Kprod, {}], {0; 1}, {0 => tv 30; 1 => tv 29});
   tv 10 @> tv 29 @> tv 29)
Conclusion

- Formalized completely structural polymorphism.

- Proved not only type soundness, but also soundness and principality of inference, and correctness of evaluation through an abstract machine.

- First step towards a certified reference implementation of OCaml. Next step might be type constructors and the relaxed value restriction.

- The techniques in *Engineering formal metatheory* proved useful, but had to redo the automation.

- Extractable proof scripts at http://www.math.nagoya-u.ac.jp/~garrigue