# A Certified Implementation of ML with Structural Polymorphism

Jacques Garrigue

Graduate School of Mathematical Sciences,
Nagoya University, Chikusa-ku, Nagoya 464-8602
garrigue@math.nagoya-u.ac.jp

**Abstract.** The type system of Objective Caml has many unique features, which make ensuring the correctness of its implementation difficult. One of these features is structurally polymorphic types, such as polymorphic object and variant types, which have the extra specificity of allowing recursion. I implemented in Coq a certified interpreter for Core ML extended with structural polymorphism and recursion. Along with type soundness of evaluation, soundness and principality of type inference are also proved.

## 1 Introduction

While many results have already been obtained in the mechanization of metatheory for ML [1–4] and pure type systems [5, 6], Objective Caml [7] has unique features which are not covered by existing works. For instance, polymorphic object and variant types require some form of structural polymorphism [8], combined with recursive types, and both of these do not map directly to usual type systems. Among the many other features, let us just cite the relaxed valued restriction [9], which accommodates side-effects in a smoother way, first class polymorphism [10] as used in polymorphic methods, labeled arguments [11], structural and nominal subtyping (the latter obtained through private abbreviations). There is plenty to do, and we are interested not only in type safety, but also in the correctness of type inference, as it gets more and more involved with each added feature.

Since it seems difficult to ensure the correctness of the current implementation, it would be nice to have a fully certified reference implementation at least for a subset of the language, so that one could check how it is supposed to work. As a first step, I certified type inference and evaluation for Core ML extended with local constraints, a form of structural polymorphism which allows inference of recursive types, such as polymorphic variants or objects. The formal proofs cover soundness of evaluation, both through rewriting rules and using a stack-based abstract machine, and soundness and completeness of the type inference algorithm.

While we based our developments on the "Engineering metatheory" methodology [6], our interest is in working on a concrete type system, with advanced

typing features, like in the mechanized metatheory of Standard ML [4]. We are not so much concerned about giving a full specification of the operational semantics, as in [12].

The contribution of this paper is two-fold. First, the proofs presented here are original, and in particular it is to our knowledge the first proof of correctness of type inference for a type system containing recursive types. Second, we have used extensively the techniques proposed in [6] to handle binding, and it is interesting to see how they fare in a system containing recursion, or when working on properties other than soundness.

## 2 Structural polymorphism

Structural polymorphism, embodied by polymorphic variants and objects, enriches types with both a form of width subsumption, and mutual recursive types. Structural polymorphism was formalized on paper in [8], by introducing a notion of *recursive kinding environment*. To help understand what we are working with, we give here the basic definitions.

Terms are the usual ones: variables, constants and functions. We intend to provide all other constructs through constants and $\delta$-rules.

$$e ::= x \mid c \mid \lambda x.e$$

Types are less usual.

| | |
|---|---|
| $\tau ::= \alpha \mid \tau_1 \to \tau_2$ | type |
| $\kappa ::= \bullet \mid (C, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\})$ | kind |
| $K ::= \alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n$ | kinding environment |
| $\sigma ::= \forall \bar{\alpha}.K \triangleright \tau$ | polytype |

A type is either a type variable or a function type. This may seem not expressive enough, but in this system type variables need not be abstract. When they are associated with a concrete kind, they actually denote structural types, like records or variants. Such types are described by the pairing of a local constraint $C$ and a mapping[1] from labels to types. On the other hand $\bullet$ just denotes an (abstract) type variable. As you can see, type variables may appear inside kinds, and since kinding environments associate type variables to kinds, we can use them to define recursive types (where the recursion must necessarily go through kinds.) A good way to understand this definition is to see types as directed graphs, where variables are just labels for nodes.

This type system is actually a framework, where the concrete definition of local constraints, and how they interact with types, is kept abstract. One can then apply this framework to an appropriate *constraint domain* to implement various flavours of polymorphic variants and records. A constraint domain $\mathcal{C}$ is a set of constraints combined with an entailment relation $\models$ on these constraints,

[1] In order to make type inference principal, this "mapping" is not always a function; this will not matter much in this paper.

$$\textsc{Variable}\quad \frac{K, K_0 \vdash \theta : K \quad \mathsf{dom}(\theta) \subset B}{K; \Gamma, x : \forall B.K_0 \triangleright \tau \vdash x : \theta(\tau)}$$

$$\textsc{Abstraction}\quad \frac{K; \Gamma, x : \tau \vdash e : \tau'}{K; \Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\textsc{Application}\quad \frac{K; \Gamma \vdash e_1 : \tau \to \tau' \quad K; \Gamma \vdash e_2 : \tau}{K; \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\textsc{Generalize}\quad \frac{K; \Gamma \vdash e : \tau \quad B = \mathsf{FV}_K(\tau) \setminus \mathsf{FV}_K(\Gamma)}{K|_{\overline{B}}; \Gamma \vdash e : \forall B.K|_B \triangleright \tau}$$

$$\textsc{Let}\quad \frac{K; \Gamma \vdash e_1 : \sigma \quad K; \Gamma, x : \sigma \vdash e_2 : \tau}{K; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\textsc{Constant}\quad \frac{K_0 \vdash \theta : K \quad \mathrm{Tconst}(c) = K_0 \triangleright \tau}{K; \Gamma \vdash c : \theta(\tau)}$$

**Fig. 1.** Typing rules (original)

$$\textsc{Variable}\quad \frac{K \vdash \bar{\tau} :: \bar{\kappa}^{\bar{\tau}}}{K; \Gamma, x : \bar{\kappa} \triangleright \tau_1 \vdash x : \tau_1^{\bar{\tau}}}$$

$$\textsc{Abstraction}\quad \frac{\forall x \notin L \qquad K; \Gamma, x : \tau \vdash e^x : \tau'}{K; \Gamma \vdash \lambda e : \tau \to \tau'}$$

$$\textsc{Application}\quad \frac{K; \Gamma \vdash e_1 : \tau \to \tau' \quad K; \Gamma \vdash e_2 : \tau}{K; \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\textsc{Generalize}\quad \frac{\forall \bar{\alpha} \notin L \quad K, \bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}}; \Gamma \vdash e : \tau^{\bar{\alpha}}}{K; \Gamma \vdash e : \bar{\kappa} \triangleright \tau}$$

$$\textsc{Let}\quad \frac{K; \Gamma \vdash e_1 : \sigma \quad \forall x \notin L \quad K; \Gamma, x : \sigma \vdash e_2^x : \tau}{K; \Gamma \vdash \mathsf{let}\ e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\textsc{Constant}\quad \frac{K \vdash \bar{\tau} :: \bar{\kappa}^{\bar{\tau}} \quad \mathrm{Tconst}(c) = \bar{\kappa} \triangleright \tau_0}{K; \Gamma \vdash c : \tau_0^{\bar{\tau}}}$$

**Fig. 2.** Typing rules using cofinite quantification

and a predicate $\mathsf{unique}(C, l)$, satisfying some properties. By extension we also use the notation $\kappa' \models \kappa$ for kinds, *i.e.* $(C', R') \models (C, R)$ iff $C' \models C$ and $R \subset R'$.

Kinding environments are used in two places: in polytypes where they associate kinds to quantified type variables, and in typing judgments, which are of the form $K; \Gamma \vdash e : \tau$, where the variables kinded in $K$ may appear in both $\Gamma$ and $\tau$. The typing rules are given in figure 1. $K \vdash \theta : K'$ means that $\theta$ preserves kinds between $K$ and $K'$ (it is *admissible* between $K$ and $K'$). Formally, if $\alpha$ has a concrete kind in $K$ ($\alpha :: \kappa \in K,\ \kappa \neq \bullet$), then $\theta(\alpha) = \alpha'$ is a variable, and it has a more concrete kind in $K'$ ($\alpha' :: \kappa' \in K'$ and $\kappa' \models \theta(\kappa)$). The main difference with Core ML is that Generalize has to split the kinding environment into a generalized part, which contains the kinds associated to generalized type variables, and a non-generalized part for the rest; since the generalized part shall not be accessible from the non-generalized part, we need to look into the kinding environment when deciding which variables can be generalized. For this reason $\mathsf{FV}$ takes a kinding environment as parameter; if $\alpha :: \kappa \in K$, then $\mathsf{FV}_K(\alpha) = \{\alpha\} \cup \mathsf{FV}_K(\kappa)$. We refer the reader to [8] for further details.

## 3 Type soundness

The first step of our mechanical proof, using Coq [13], was to prove type soundness for the system described in the previous section, starting from Aydemir and others proof for Core ML included in [6], which uses *locally nameless cofinite*

*quantification*. This proof uses de Bruijn indices for local quantification inside terms and polytypes, and quantifies over an abstract avoidance set for avoiding name conflicts.

Figure 2 contains the typing rules adapted to locally nameless cofinite quantification, using a modified definition of terms and types.

$$\begin{array}{ll} e ::= n \mid x \mid c \mid \lambda e & \text{term} \\ \tau ::= n \mid \alpha \mid \tau_1 \to \tau_2 & \text{type} \\ \sigma ::= \bar{\kappa} \triangleright \tau & \text{polytype} \end{array}$$

$\bar{\tau}$ and $\bar{\kappa}$ represent sequences of types and kinds. When we write $\bar{\alpha}$, we also assume that all type variables inside the sequence are distinct. Polytypes are now written $\bar{\kappa} \triangleright \tau$, where the length of $\bar{\kappa}$ is the number of generalized type variables, represented as de Bruijn indices $1 \dots n$ inside types. The actual implementation has indices starting from 0, but we will start from 1 in this explanation. $\tau_1^{\bar{\tau}}$ is $\tau_1$ where de Bruijn indices were substituted with types of $\bar{\tau}$, accessed by their position. Similarly $\bar{\kappa}^{\bar{\tau}}$ substitute all the indices inside the sequence $\bar{\kappa}$. $e^x$ only substitutes $x$ for the index 1. $K \vdash \tau :: \kappa$ is true when either $\kappa = \bullet$, or $\tau = \alpha$, $\alpha :: \kappa' \in K$ and $\kappa' \models \kappa$. $K \vdash \bar{\tau} :: \bar{\kappa}$ enforces this for every member of $\bar{\tau}$ and $\bar{\kappa}$ at identical positions, which is just equivalent to our condition $K \vdash \theta : K'$ for the preservation of kinds.

$\forall x \notin L$ and $\forall \bar{\alpha} \notin L$ are cofinite quantifications. At first, the rules may look very different from those in figure 1, but they coincide if we instantiate $L$ appropriately. For instance, if we use $\mathsf{dom}(\Gamma)$ for $L$ in $\forall x \notin L$, this just amounts to ensuring that $x$ is not already bound. Inside Generalize, we could use $\mathsf{dom}(K) \cup \mathsf{FV}_K(\Gamma)$ for $L$ to ensure that the newly introduced variables are locally fresh. This may not be intuitive, but this is actually a very clever way to encode naming constraints implicitly. Moreover, when we build a new typing derivation from an old one, we can avoid renaming variables by just enlarging the avoidance sets.

Starting from an existing proof was a tremendous help, but many new definitions were needed to accommodate kinds, and some existing ones had to be modified. For instance, in order to accommodate the mutually recursive nature of kinding environments, we need simultaneous type substitutions, rather than the iterated ones of the original proof. The freshness of individual variables (or sequences of variables: $\bar{\alpha} \notin L$) becomes insufficient, and we need to handle disjointness conditions on sets ($L_1 \cap L_2 = \emptyset$). As a result, the handling of freshness, which was almost fully automatized in the proof of Core ML, required an important amount of work with kinds, even after developing some tactics for disjointness.

I also added a formalism for constants and $\delta$-rules, which are needed to give an operational semantics to structural types. Overall, the result was a doubling of the size of the proof, from 1000 lines to more than 2000, but the changes were mostly straightforward. This does not include the extra metatheory lemmas and set inclusion tactics that we use for all proofs.

The formalism of local constraints was defined as a framework, able to handle various flavours of variant and object types, just by changing the constraint part

```
Module Type CstrIntf.
 Parameter cstr attr : Set.
 Parameter valid : cstr → Prop.
 Parameter valid_dec : ∀c, {valid c} + {¬valid c}.
 Parameter eq_dec : ∀xy : attr, {x = y} + {x ≠ y}.
 Parameter unique : cstr → attr → bool.
 Parameter ⊔ : cstr → cstr → cstr.
 Parameter ⊨ : cstr → cstr → Prop.
 Parameter entails_refl : ∀c, c ⊨ c.
 Parameter entails_trans : ∀c1c2c2, c1 ⊨ c2 → c2. ⊨ c3 → c1 ⊨ c3.
 Parameter entails_lub : ∀cc1c2, c ⊨ c1 ∧ c ⊨ c2 ↔ c ⊨ c1 ⊔ c2.
 Parameter entails_unique : ∀vc1c2, c1 ⊨ c2 → unique c2 v = true → unique c1 v = true.
 Parameter entails_valid : ∀c1c2, c1 ⊨ c2 → valid c1 → valid c2.
End CstrIntf.

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

**Fig. 3.** Interfaces for constraints and constants

of the system. This was formalized through the use of functors. The signature for constraints and constants is in figure 3.

This worked well, but there are some drawbacks. One is that since some type definitions depend on parameters, and some required proofs depend on those definitions, we need nested functors, and the instantiation of the framework with a *constraint domain* looks like a "dialogue". The problem comes not so much for constraints themselves, but rather from constants and delta-rules. We show the basic module structure of the proof in figure 4. In order to obtain the definitions for typing judgments, one has to provide implementations for constraints and constants, extract the definition of types and terms, and use them to provide constant types and $\delta$-rules. We enforce the completeness of $\delta$-rules by requiring a function reduce which will be applied to a list of values of length $(1 + \mathsf{Const.arity}\ c)$; through well-typedness they will be only used if Const.arity $c$ is smaller than the arity of type $c$. Type soundness itself is another functor, that requires some lemmas whose proofs may use infrastructure lemmas on type judgments, and returns proofs of preservation and progress. The real structure is even more complex, because the proofs span several files, and each file must mimick this structure.

This instantiation has been done for a constraint domain containing both polymorphic variants and records, and a fixpoint operator. We show the constraint domain in figure 5; we write $\langle\rangle$ for None, which denotes here the set of all possible labels. Constants and $\delta$-rules are in figure 6, using the nameful syntax for types. You can see the duality between variants and records, at least for tag and get.

```
Module Type CstrIntf ...
Module Type CstIntf ...
Module MkDefs(Cstr:CstrIntf)(Const:CstIntf).
 Inductive typ : Set := ...
 Inductive type : Set := ...
 Inductive trm : Set := ...
 ...
 Module Type DeltaIntf.
  Parameter type : Const.const → sch.
  Parameter closed : ∀c, sch_fv(type c) = ∅.
  Parameter scheme : ∀c, scheme(type c).
  Parameter reduce : ∀c tl, (list_for_n value (1 + Const.arity c) tl) → trm.
  Parameter term : ∀c tl vl, term(reduce c tl vl).
 End DeltaIntf.
 Module MkJudge(Delta:DeltaIntf).
  Inductive ⊢ : kenv → env → trm → typ → Prop := ...
  Inductive ⟶ : trm → trm → Prop := ...
  Inductive value : trm → Prop := ...
  ...
  Module Type SndHypIntf.
   Parameter delta_typed : ∀c tl vl K Γ gc τ,
     (K; Γ ⊢ const_app c tl : τ) → (K; Γ ⊢ Delta.reduce c tl vl : τ).
  End SndHypIntf.
  Module MkSound(SH:SndHypIntf).
   Theorem preservation : ∀K Γ e e' τ, (K; Γ ⊢ e : τ) → (e ⟶ e') → (K; Γ ⊢ e' : τ).
   Theorem progress : ∀K e τ, (K; ∅ ⊢ e : τ) → (value e ∨ ∃e', e ⟶ e').
  End MkSound.
 End MkJudge.
End MkDefs.
```

**Fig. 4.** Module structure

Both in the framework and domain proofs, cofinite quantification demonstrated its power, as no renaming of type or term variables was needed at all. It helped also in an indirect way: in the original rule for Generalize, one has to close the set of free variables of a type with the free variables of their kinds; but the cofinite quantification takes care of that implicitly, without any extra definitions.

While cofinite quantification may seem perfect, there is a pitfall in this perfection itself. One forgets that some proof transformations intrinsically require variable renaming. Concretely, to make typing more modular, I added a rule that discards irrelevant kinds from the kinding environment. Figure 7 shows both the normal and cofinite forms. Again one can see the elegance of the cofinite version, where there is no need to say which kinds are irrelevant: just the ones whose names have no impact on typability. Proofs went on smoothly, until I realized

```
Module Cstr.
 Definition attr := nat.
 Inductive ksort : Set := Ksum | Kprod | Kbot.
 Record cstr : Set := C{sort : ksort; low : list nat; high : option(list nat)}.
 Definition valid c := sort c ≠ Kbot ∧ (high c = ⟨⟩ ∨ low c ⊂ high c).
 Definition s1 ≤ s2 := s1 = Kbot ∨ s1 = s2.
 Definition c1 ⊨ c2 :=
    sort s2 ≤ sort s1 ∧ low c2 ⊂ low c1 ∧ (high c2 = ⟨⟩ ∨ high c1 ⊂ high c2).
 ...
EndCstr.
```

**Fig. 5.** Constraint domain for polymorphic variants and records

$$
\begin{aligned}
\mathsf{type}(\mathsf{tag}_l) &= \alpha :: (\langle \mathsf{Ksum}, \{l\}, \langle\rangle\rangle, \{l \mapsto \beta\}) \triangleright \beta \to \alpha \\
\mathsf{type}(\mathsf{match}_{l_1 \ldots l_n}) &= \alpha :: (\langle \mathsf{Ksum}, \emptyset, \{l_1, ..., l_n\}\rangle, \{l_1 \mapsto \alpha_1, \ldots, l_n \mapsto \alpha_n\}) \\
&\quad \triangleright (\alpha_1 \to \beta) \to \ldots \to (\alpha_n \to \beta) \to \alpha \to \beta \\
\mathsf{type}(\mathsf{record}_{l_1 \ldots l_n}) &= \alpha :: (\langle \mathsf{Kprod}, \emptyset, \{l_1, ..., l_n\}\rangle, \{l_1 \mapsto \alpha_1, \ldots, l_n \mapsto \alpha_n\}) \\
&\quad \triangleright \alpha_1 \to \ldots \to \alpha_n \to \alpha \\
\mathsf{type}(\mathsf{get}_l) &= \alpha :: (\langle \mathsf{Kprod}, \{l\}, \langle\rangle\rangle, \{l \mapsto \beta\}) \triangleright \alpha \to \beta \\
\mathsf{type}(\mathsf{recf}) &= ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{match}_{l_1 \ldots l_n}\ f_1\ \ldots\ f_n\ (\mathsf{tag}_{l_i}\ e) &\longrightarrow f_i\ e \\
\mathsf{get}_{l_i}\ (\mathsf{record}_{l_1 \ldots l_n}\ e_1\ \ldots\ e_n) &\longrightarrow e_i \\
\mathsf{recf}\ f\ e &\longrightarrow f\ (\mathsf{recf}\ f)\ e
\end{aligned}
$$

**Fig. 6.** Types and $\delta$-rules for constants

that I needed the following inversion lemma.

$$\forall K \Gamma e \tau,\ (K; \Gamma \vdash_{GC} e : \tau) \to \exists K',\ (K, K'; \Gamma \vdash e : \tau)$$

Namely, by putting back the kinds we discarded, we shall be able to obtain a derivation that does not rely on Kind GC. This is very intuitive, but since this requires making Kind GC commute with Generalize, we end up commuting quantifiers. And this is just impossible without a true renaming lemma. I got stuck there for a while, unable to see what was going wrong. Even more confusing, the same problem occurs when we try to make Kind GC commute with Abstraction, whereas intuitively the choice of names for term variables is independent of the choice of names for type variables. Finally this lemma required about 1000 lines to prove it, including renaming lemmas for both term and type variables. The renaming lemmas were harder to prove than expected (100 lines each). Contrary to what was suggested in [6], we found it rather difficult to prove these lemmas starting from the substitution lemmas of the soundness proof; while renaming for types used this approach, renaming for terms was proved directly, and they ended up being of the same length. Once the problem becomes clear, one can see a much simpler solution: in most situations, it

$$\textsc{Kind GC}\quad \frac{K, K'; \Gamma \vdash e : \tau \quad \mathsf{FV}_K(E, \tau) \cap \mathsf{dom}(K') = \emptyset}{K; \Gamma \vdash e : \tau} \qquad \textsc{Co-finite Kind GC}\quad \frac{\forall \bar{\alpha} \notin L \quad K, \bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}}; \Gamma \vdash e : \tau}{K; \Gamma \vdash e : \tau}$$

**Fig. 7.** Kind discarding

is actually sufficient to have Kind GC occur only just above Abstraction and Generalize, and the canonicalization lemma is just 100 lines. This also raises the issue of how to handle several variants of a type system in the same proof. Here this was done by parameterizing the predicate $\vdash$ with the canonicity of the derivation, and whether Kind GC is allowed at this point. This gives 4 cases for the availability of Kind GC: allowed nowhere, allowed everywhere, or inside a canonical derivation where it is allowed or not at the current point. Functions gc_ok, gc_raise and gc_lower allow to manipulate this state transparently.

## 4 Type inference

The main goal of using local constraints was to keep the simplicity of unification-based type inference. Of course, unification has to be extended in order to handle kinding, but the algorithms for unification and type inference stay reasonably simple.

### 4.1 Unification

Unification has been a target of formal verification for a long time, with formal proofs as early as 1985 [14]. Here I just wrote down the algorithm in Coq, and proved both partial-correctness and completeness. A rule-based version of the algorithm can be found in [8]. The following statements were proved:

Definition unifies $\theta\ l :=$ $\forall \tau_1 \tau_2, \mathsf{In}\ (\tau_1, \tau_2)\ l \rightarrow\ \theta(\tau_1) = \theta(\tau_2)$.

Theorem unify_types : $\forall h\, l\, K\, \theta$, unify $h\ l\ K\ \theta = \langle K', \theta' \rangle \rightarrow$ unifies $\theta'\ l$.

Theorem unify_kinds : $\forall h\, l\, K\, \theta$,
unify $h\ l\ K\ \theta = \langle K', \theta' \rangle \rightarrow \mathsf{dom}(\theta) \cap \mathsf{dom}(K) = \emptyset \rightarrow$
$K \vdash \theta' : \theta'(K') \wedge \mathsf{dom}(\theta') \cap \mathsf{dom}(K') = \emptyset$.

Theorem unify_mgu : $\forall h\, l\, K_0\, K\, \theta$,
unify $h\ l\ K_0$ id $= \langle K, \theta \rangle \rightarrow$ unifies $\theta'\ l \rightarrow K_0 \vdash \theta' : K' \rightarrow \theta' \sqsupseteq \theta \wedge K \vdash \theta : K'$.

Theorem unify_complete : $\forall K\, \theta\, K_0\, l\, h$,
unifies $\theta\ l \rightarrow K_0 \vdash \theta : K \rightarrow$ size_pairs id $K_0\ l < h \rightarrow$ unify $h\ l\ K_0$ id $\neq \langle\rangle$.

The first argument to unify is the number of type variables, which is used to enforce termination. Then come a list of type pairs to unify and the original kind environment. Last is a starting substitution, so that the algorithm is tail-recursive. To keep the statement clear, well-formedness conditions are omitted here. The proof is rather long, as kinds need particular treatment, but there was no major stumbling block. The proof basically follows the algorithms, but

$[\bar\alpha]\tau = \tau_*$ such that $\tau_*^{\bar\alpha} = \tau$
and $\mathsf{FV}(\tau_*) \cap \bar\alpha = \emptyset$
$[\bar\alpha](\bar\kappa \triangleright \tau) = ([\bar\alpha]\bar\kappa \triangleright [\bar\alpha]\tau)$

Definition generalize$(K, \Gamma, L, \tau) :=$
  let $A = \mathsf{FV}_K(\Gamma)$ and $B = \mathsf{FV}_K(\tau)$ in
  let $K' = K|_{\overline{A}}$ in
  let $\bar\alpha :: \bar\kappa = K'|_B$ in
  let $\bar\alpha' = B \setminus (A \cup \bar\alpha)$ in
  let $\bar\kappa' = \mathsf{map}\ (\lambda\_.\bullet)\ \bar\alpha'$ in
  $\langle (K|_A, K'|_L), [\bar\alpha\bar\alpha'](\bar\kappa\bar\kappa' \triangleright \tau)\rangle$.

Definition typinf$(K, \Gamma, \mathsf{let}\ e_1\ \mathsf{in}\ e_2, \tau, \theta, L) :=$
  let $\alpha = \mathsf{fresh}(L)$ in
  match typinf$(K, \Gamma, e_1, \alpha, \theta, L \cup \{\alpha\})$ with
  | $\langle K', \theta', L'\rangle \Rightarrow$
    let $K_1 = \theta'(K')$ and $\Gamma_1 = \theta'(\Gamma)$ in
    let $L_1 = \theta'(\mathsf{dom}(K))$ and $\tau_1 = \theta'(\tau')$ in
    let $\langle K_A, \sigma\rangle =$ generalize$(K_1, \Gamma_1, L_1, \tau_1)$ in
    let $x = \mathsf{fresh}(\mathsf{dom}(E) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2))$ in
    typinf$(K_A, (\Gamma, x : \sigma), e_2^x, \tau, \theta', L')$
  | $\langle\rangle \Rightarrow \langle\rangle$
  end.

**Fig. 8.** Type inference algorithm

there are two useful tricks. One concerns substitutions. Rather than using the relation "$\theta$ is more general than $\theta'$" ($\exists\theta_1,\ \theta' = \theta_1 \circ \theta$), I used the more direct "$\theta'$ extends $\theta$" ($\forall\alpha,\ \theta'(\theta(\alpha)) = \theta'(\alpha)$). In the above theorem it is noted $\theta' \sqsupseteq \theta$. When $\theta$ is idempotent, the two definitions are equivalent, but the latter can be used directly through rewriting. The other idea was to define a special induction lemma for successful unification, which uses symmetries to reduce the number of cases to check. Unification being done on first-order terms, the types we are unifying shall contain no de Bruijn indices, but only global variables. Since we started with a representation allowing both kinds of variables, there was no need to change it.

### 4.2 Inference

The next step is type inference itself. Again, correctness has been proved before for Core ML [1, 2], but to my knowledge never for a system containing equi-recursive types. Proving both soundness and principality was rather painful. This time one problem was the complexity of the algorithm itself, in particular the behaviour of type generalization. The usual behaviour for ML is just to find the variables that are not free in the typing environment and generalize them, but with a kinding environment several extra steps are required. First, the free variables should be closed transitively using the kinding environment. Then, the kinding environment also should be split into generalizable and non-generalizable parts. Last, some generalizable parts of the kinding environment need to be duplicated, as they might be used independently in some other parts of the typing derivation. The definitions for generalize and the let case of typinf are shown in figure 8. $[\bar\alpha]\tau$ stands for the generalization of $\tau$ with respect to $\bar\alpha$, obtained by replacing the occurrences of variables of $\bar\alpha$ in $\tau$ by their indices.

Due to the large number side-conditions required, the statements for the inductive versions of soundness of principality become very long. In figure 9 we show slightly simplified versions, omitting well-formedness properties. These statements can be proved directly by induction. From those, we can derive the

$$\begin{array}{l}\mathsf{Theorem\ soundness} : \forall K\Gamma e\tau\theta L K'\theta' L',\\ \quad \mathsf{typinf}(K,\Gamma,e,\tau,\theta,L) = \langle K',\theta',L'\rangle \rightarrow\\ \quad \mathsf{dom}(\theta)\cap\mathsf{dom}(K) = \emptyset \rightarrow\\ \quad \mathsf{FV}(\theta,K,\Gamma,\tau) \subset L \rightarrow\\ \quad \theta'(K');\theta'(\Gamma) \vdash e : \theta'(\tau) \wedge\\ \quad K \vdash \theta' : \theta'(K') \wedge \theta' \sqsupseteq \theta \wedge\\ \quad \mathsf{FV}(\theta',K',\Gamma)\cup L \subset L' \wedge\\ \quad \mathsf{dom}(\theta')\cap\mathsf{dom}(K') = \emptyset.\end{array}$$

$$\begin{array}{l}\mathsf{Theorem\ principality} : \forall K\Gamma e\tau\theta K_1\theta_1 L,\\ \quad K;\theta(\Gamma) \vdash e : \theta(\tau) \rightarrow K_1 \vdash \theta : K \rightarrow\\ \quad \theta \sqsupseteq \theta_1 \rightarrow \mathsf{dom}(\theta_1)\cap\mathsf{dom}(K_1) = \emptyset \rightarrow\\ \quad \mathsf{dom}(\theta)\cup\mathsf{FV}(\theta_1,K_1,\Gamma,\tau) \subset L \rightarrow\\ \quad \exists K'\theta' L',\\ \quad\ \mathsf{typinf}(K_1,\Gamma,e,\tau,\theta_1,L) = \langle K',\theta',L'\rangle \wedge\\ \quad\ \exists\theta'',\ K' \vdash \theta\theta'' : K \wedge \theta\theta'' \sqsupseteq \theta' \wedge\\ \qquad \mathsf{dom}(\theta'') \subset L' \setminus L.\end{array}$$

**Fig. 9.** Properties of type inference

following corollaries for a simplified version of typinf, taking only a term and a closed environment as arguments.

$$\mathsf{Corrolary\ soundness'}\ : \forall K\Gamma e\tau,\ \mathsf{FV}(\Gamma) = \emptyset \rightarrow \mathsf{typinf'}\ E\ e = \langle K,\tau\rangle \rightarrow K;\Gamma \vdash e : \tau.$$

$$\begin{array}{l}\mathsf{Corollary\ principality'} : \forall K\Gamma e\tau,\ \mathsf{FV}(\Gamma) = \emptyset \rightarrow K;\Gamma \vdash e : \tau \rightarrow\\ \qquad \exists K', \exists T',\ \mathsf{typinf'}\ \Gamma\ e = \langle K',T'\rangle \wedge \exists\,\theta,\ K' \vdash \theta : K \wedge \tau = \theta(\tau').\end{array}$$

As usual, the proof of principality requires the following lemma, which states that if a term $e$ has a type $\tau$ under an environment $\Gamma$, then we can give it the same type under a more general environment $\Gamma_1$.

$$\mathsf{Lemma\ typing\_moregen} : \forall K\Gamma\Gamma_1 e\tau,\ K;\Gamma \vdash e : \tau \rightarrow \Gamma \vdash \Gamma_1 \leq \Gamma \rightarrow K;\Gamma_1 \vdash e : \tau.$$

$K \vdash \Gamma_1 \leq \Gamma$ means that the polytypes of $\Gamma$ are instances of those in $\Gamma_1$. Due to the presence of kinds, the definition of the instantiation order gets a bit complicated.

$$\begin{array}{l}K \vdash \bar\kappa_1 \triangleright \tau_1 \leq \bar\kappa \triangleright \tau \stackrel{\mathrm{def}}{=}\\ \quad \forall\bar\alpha, \mathsf{dom}(K)\cap\bar\alpha = \emptyset \rightarrow \exists\bar\tau,\ K,\bar\alpha :: \bar\kappa^{\bar\alpha} \vdash \bar\tau :: \bar\kappa_1^{\bar\tau} \wedge \tau_1^{\bar\tau} = \tau^{\bar\alpha}.\end{array}$$

It may be easier to consider the version without de Bruijn indices.

$$\begin{array}{l}K \vdash \forall\bar\alpha_1.K_1 \triangleright \tau_1 \leq \forall\bar\alpha_2.K_2 \triangleright \tau_2 \stackrel{\mathrm{def}}{=}\\ \qquad \exists\theta,\ \mathsf{dom}(\theta) \subset \bar\alpha_1\ \ \wedge K, K_1 \vdash \theta : K, K_2 \wedge \theta(\tau_1) = \tau_2.\end{array}$$

Another difficulty is that, since we are building a derivation, cofinite quantification appears as a requirement rather than a given, and we need renaming for both terms and types in many places. This is true both for soundness and principality, since in the latter the type variables of the inferred derivation and of the provided derivation are different. As a result, while we could finally avoid using the renaming lemmas for type soundness, they were ultimately needed for type inference.

## 5 Interpreter

Type soundness ensures that evaluation according to a set of source code rewriting rules cannot go wrong. However, programming languages do not evaluate

```
Inductive clos : Set :=
  | clos_abs   : trm → list clos → clos
  | clos_const : Const.const → list clos → clos.
Fixpoint clos2trm(c : clos) : trm :=
 match c with
  | clos_abs e l   ⇒ trm_inst (λe) (map clos2trm l)
  | clos_const c l ⇒ const_app c (map clos2trm l)
 end.
Record frame : Set := Frame {frm_benv : list clos; frm_app : list clos; frm_trm : trm}.
Inductive eval_res : Set :=
  | Result : nat → clos → eval_res
  | Inter  : list frame → eval_res.
Fixpoint eval (h : nat) (benv : list clos) (app : list clos) (e : trm) (stack : list frame)
   {struct h} : eval_res := ...
Theorem eval_sound : ∀h K e τ,
   (K; Γ ⊢ e : τ) → (K; Γ ⊢ res2trm (eval h nil nil t nil) : τ).
Theorem eval_complete : ∀K e e' τ,
   (K; Γ ⊢ e : τ) → (e →* e') → value t' →
   ∃h, ∃cl, eval h nil nil t nil = Result 0 cl ∧ e' = clos2trm cl.
```

**Fig. 10.** Definitions and theorems for stack-based evaluation

a program by rewriting it, but rather interpreting it with a virtual machine. I defined a stack-based abstract machine, and proved that at every step the state of the abstract machine could be converted back to a term whose typability was a direct consequence of the typability of the reduced program. This ensures that evaluation cannot go wrong, and the final result, if reached, shall be either a constant or a function closure. Once the relation between program and state was properly specified, the proof was mostly straightforward.

The basic definitions and the statements for soundness and completeness are in figure 10. A closure is either a function body paired with its environment, or a partially applied constant. clos2trm converts back a closure to an equivalent term. Since evaluation may not terminate, eval takes as argument the number $h$ of reduction steps to compute. The remaining arguments are the environment *benv*, accessed through de Bruijn indices, the application stack *app* which contains the arguments to the term being evaluated, the term $e$ itself, which provides an efficient representation of code thanks to de Bruijn variables, and the control stack *stack*. Here the nameless representation of terms was handy, as it maps naturally to a stack machine. The result of eval is either a closure, with the number of evaluation steps remaining, or the current state of the machine.

I also proved completeness with respect to the rewriting rules, *i.e.* if the rewriting based evaluation reaches a normal form, then evaluation by the abstract machine terminates with the same normal form. This required building

a bisimulation between the two evaluations, and was trickier than expected. Namely we need to prove the following lemma:

```
Definition inst t benv := trm_inst t (map clos2trm benv).
Lemma complete_rec : ∀args args' fl fl' e e' benv benv' τ,
  args ≡ args' → fl ≡ fl' → (inst e benv ⟶ inst e' benv') →
  K; Γ ⊢ stack2trm (app2trm (inst e benv) args) fl : τ →
  ∃h, ∃h', eval h benv args e fl ≡ eval h' benv' args' e' fl'.
```

where $\equiv$ denotes the equality of closures after substitution by their environment, *i.e.* clos_abs $e$ $benv$ $\equiv$ clos_abs $e'$ $benv'$ iff inst $(\lambda e)$ $benv$ $=$ inst $(\lambda e')$ $benv'$. Proving this by case analysis on $e$ and $e'$ ended up being very time consuming. The proofs being rather repetitive, they may profit from better lemmas.

## 6 Dependent types

As we pointed in section 4, the statements of many lemmas and theorems include lots of well-formedness properties, which are expected to be true of any value of a given type. For instance, substitutions should be idempotent, environments should not bind the same variable twice, de Bruijn indices should not escape, kinds should be valid, *etc.*.. A natural impulse is to use dependent types to encode these properties. Yet proofs from [6] only use dependent types for the generation of fresh variables. The reason is simple enough: as soon as a value is defined as a dependent sum, using rewriting on it becomes much more cumbersome. I attempted using it for the well-formedness of polytypes, but had to abandon the idea because there were too many things to prove upfront. On the other hand, using dependent types to make sure that kinds are valid and coherent was not so hard, and helped streamline the proofs. This is probably due to the abstract nature of constraint domains, which limits interactions between kinds and other features. The definition of kinds becomes:

```
Definition coherent kc kr := ∀x (τ τ' : typ),
  Cstr.unique kc x = true → In (x, τ) kr → In (x, τ') kr → τ = τ'.
Record ckind : Set := Kind{
  kind_cstr : Cstr.cstr; kind_valid : Cstr.valid kind_cstr;
  kind_rel : list (Cstr.attr × typ); kind_coherent : coherent kind_cstr kind_rel}.
Definition kind := option ckind.
```

We still need to apply substitutions to kinds, but this is not a problem as substitutions do not change the constraint, and preserve the coherence. We just need the following function.

```
Definition ckind_map_spec : ∀(f : typ → typ)(k : ckind),
  {k' : ckind | kind_cstr k = kind_cstr k' ∧ kind_rel k' = map_snd f (kind_rel k)}.
```

We also sometimes have to prove the equality of two kinds obtained independently. This requires the following lemma, which can be proved using proof irrelevance.

```
Lemma kind_pi : ∀k k' : ckind,
  kind_cstr k = kind_cstr k' → kind_rel k = kind_rel k' → Some k = Some k'.
```

Another application of dependent types is ensuring termination for the unification and type inference algorithms. In Coq all functions must be complete. Originally, this was ensured by adding a step counter, and proving separately that one can choose a number of steps sufficient to obtain a result. This is the style used in section 4.1. This approach is simple, but this extra parameter stays in the extracted code. In a first version of the proof, the parameter was so big that the unification algorithm would just take forever trying to compute the number of steps it needed. I later came up with a smaller value, but it would be better to have it disappear completely. This is supported in Coq through well-founded recursion. In practice this works by moving the extra parameter to the universe of proofs (Prop), so that it will disappear during extraction. The Function command automates this, but there is a pitfall: while it generates dependent types, it doesn't support them in its input. The termination argument for unification being rather complex, this limitation proved problematic. Attempts with Program Fixpoint didn't succeed either. Finally I built the dependently typed function by hand. While this requires a rather intensive use of dependent types, the basic principle is straightforward, and it makes the proof of completeness simpler. As a result the overall size of the proof for unification didn't change. However, since the type inference algorithm calls unification, it had to be modified too, and its size grew by about 10%. An advantage of building our functions by hand is that we control exactly the term produced; since rewriting on dependently typed terms is particularly fragile, this full control proves useful.

## 7 Program extraction

Both the type checker and interpreter can be extracted to Objective Caml code. This lets us build a fully certified implementation for a fragment of Objective Caml's type system. Note that there is no parser or read-eval-print loop yet, making it just a one-shot interpreter for programs written directly in abstract syntax. Moreover, since Coq requires all programs to terminate, one has to indicate the number of steps to be evaluated explicitly. (Actually, Objective Caml allows one to define cyclic constants, so that we can build a value representing infinity, and remove the need for an explicit number of steps. However, this goes around the soundness of Coq.)

Here is an example of program written in abstract syntax (with a few abbreviations), and its inferred type (using lots of pretty printing).

```
# 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)]);
    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}>);
  (30, any);
  (26, <Kprod, {}, {0; 1}, {0 => tv 30; 1 => tv 29}>);
  (15, any)],
 tv 10 @> tv 29 @> tv 29)
```

Here `recf` is an extra constant which implements the fixpoint operator. Our encoding of lists uses 0 and 1 as labels for both variants and records, but we could have used any other natural numbers: their meaning is not positional, but associative. Since de Bruijn indices can be rather confusing, here is a version translated to a syntax closer to Objective Caml, with meaningful variable names and labels.

```
let rec rev_append l1 l2 =
  match l1 with
  | `Nil _ -> l2
  | `Cons c ->
    rev_append c.tl (`Cons {hd=c.hd; tl=l2})
val rev_append :
([< `Nil of '15 | `Cons of {hd:'30; tl:'10; ..}] as '10) ->
([> `Cons of {hd:'30; tl:'29}] as '29) -> '29
```

## 8 Related works

The mechanization of type safety proofs for programming languages has been extensively studied. Existing works include Core ML using Coq [3], Java using Isabelle/HOL [15], and more recently full specification fo OCaml light using HOL-light [12] and Standard ML using Twelf [4, 16]. The main difference in our system is the presence of structural polymorphism and recursion. In particular, among the above works, only [4] handles iso-recursive types. It is also the work closest to our goal of handling advanced type features (it already handles fully Standard ML). OCaml-light rather focuses on subtle points in the dynamic semantics of the language. Typed Scheme [17] has a type system remarkably similar to ours, and part of the soundness proof was mechanized in Isabelle/HOL, but the mechanized part does not contain recursive types.

Concerning unification and type inference, we have already mentioned the works of Paulson in LCF [14], Dubois and Ménissier-Morain in Coq [2], and Naraschewski and Nipkow in Isabelle [1]. The main difference is the introduction of structural polymorphism, which results in much extended statements to handle admissible substitutions. Even in the absence of structural polymorphism, just handling equi-recursive types makes type inference more complex, and I am aware of no proof of principality including them.

| File | Lines | Contents |
|---|---|---|
| Lib_* | 1706 | Auxiliary lemmas and tactics from [6] |
| Metatheory | 1376 | Metatheory lemmas and tactics from [6] |
| Metatheory_SP | 1304 | Additional lemmas and tactics |
| Definitions | 458 | Definition of the type system |
| Infrastructure | 1152 | Common lemmas |
| Soundness | 633 | Soundness proof |
| Rename | 985 | Renaming and inversion lemmas |
| Eval | 2935 | Stack-based evaluation |
| Unify | 1832 | Unification |
| Inference | 3159 | Type inference |
| Domain | 1085 | Constraint domain specific proofs |
| Unify_wf | 1827 | Unification using dependent measure |
| Inference_wf | 3443 | Inference using dependent measure |

**Table 1.** Structure of the proof

## 9 Conclusion

We have reached our first goal, providing a fully certified type checker and interpreter. We show the size and contents of the various components of the proof in table 1. While this is a good start, it currently handles only a very small subset of Objective Caml. The next goal is of course to add new features. A natural next target would be the addition of side-effects, with the relaxed value restriction. Note that since the value restriction relies on subtyping, it would be natural to also add type constructors, with variance annotations, at this point. Considering the difficulties we have met up to know, we do not expect it to be an easy task.

All the proofs and the extracted code can be found at:

`http://www.math.nagoya-u.ac.jp/~garrigue/papers/#certint1001`

## References

1. Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm W in Isabelle/HOL. Journal of Automated Reasoning **23** (1999) 299–318
2. Dubois, C., Ménissier-Morain, V.: Certification of a type inference tool for ML: Damas-Milner within Coq. Journal of Automated Reasoning **23** (1999) 319–346
3. Dubois, C.: Proving ML type soundness within Coq. In: Proc. of the International Conference on Theorem Proving in Higher Order Logics. Volume 1869 of Springer LNCS. (2000) 126–144
4. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of standard ML. In: Proc. ACM Symposium on Principles of Programming Languages. (2007) 173–184
5. Barras, B.: Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat, Université Paris 7 (1999)

6. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Proc. ACM Symposium on Principles of Programming Languages. (2008) 3–15
7. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.11, Documentation and user's manual. Projet Gallium, INRIA. (2008)
8. Garrigue, J.: Simple type inference for structural polymorphism. In: The Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, Oregon (2002)
9. Garrigue, J.: Relaxing the value restriction. In: Proc. International Symposium on Functional and Logic Programming. Volume 2998 of Springer LNCS., Nara (2004)
10. Garrigue, J., Rémy, D.: Extending ML with semi-explicit higher order polymorphism. Information and Computation **155** (1999) 134–171
11. Furuse, J.P., Garrigue, J.: A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University (1995)
12. Owens, S.: A sound semantics for OCaml light. In: Proc. European Symposium on Programming. Volume 4960 of Springer LNCS. (2008) 1–15
13. The Coq Team: The Coq Proof Assistant, Version 8.2. INRIA. (2009)
14. Paulson, L.: Verifying the unification algorithm in LCF. Science of Computer Programming **5** (1985) 143–169
15. Oheimb, D.v., Nipkow, T.: Machine-checking the Java specification : Proving type-safety. In Alves-Foss, J., ed.: Formal Syntax and Semantics of Java. Volume 1523 of LNCS. Springer (1999) 119–156
16. Crary, K., Harper, B.: Mechanized definition of Standard ML alpha release. Twelf proof scripts (2009)
17. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proc. ACM Symposium on Principles of Programming Languages. (2008)