A Certified Interpreter for ML with Structural Polymorphism

Jacques Garrigue
Nagoya University Graduate School of Mathematics
garrigue@math.nagoya-u.ac.jp

August 27, 2009

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 and pure type systems, Objective Caml has unique features which are not covered by existing works. For instance, polymorphic object and variant types require some form of structural polymorphism, 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, which accommodates side-effects in a smoother way, first class polymorphism as used in polymorphic methods, labeled arguments, 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 [1], our interest is in working on a concrete type system, with advanced typing features, like in the mechanized metatheory of Standard ML [2]. We are not so much concerned about giving a full specification of the operational semantics, as in [3].

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 [4], by introducing a notion of *recursive kinding environment*. To help understand what we are working with, we give here the basic definitions.

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

A type is either a type variable or a function type. But 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 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).

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 Γ and τ. The typing rules are given in figure 1. $\mathrm{K} \vdash \theta : \mathrm{K}'$ means that θ preserves kinds between K and K′ (it is *admissible* between K and K′). Formally, if α has a concrete kind in K ($\alpha :: \kappa \in \mathrm{K},\ \kappa \neq \bullet$), then $\theta(\alpha) = \alpha'$ is a variable, and it has a more concrete

VARIABLE
$$\frac{\mathrm{K}, \mathrm{K}_0 \vdash \theta : \mathrm{K} \quad \mathrm{dom}(\theta) \subset B}{\mathrm{K}; \Gamma, x : \forall B.\mathrm{K}_0 \triangleright \tau \vdash x : \theta(\tau)}$$

ABSTRACTION
$$\frac{\mathrm{K}; \Gamma, x : \tau \vdash e : \tau'}{\mathrm{K}; \Gamma \vdash \lambda x.e : \tau \to \tau'}$$

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

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

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

CONSTANT
$$\frac{\mathrm{K}_0 \vdash \theta : \mathrm{K} \quad \mathrm{Tconst}(c) = \mathrm{K}_0 \triangleright \tau}{\mathrm{K}; \Gamma \vdash c : \theta(\tau)}$$

Figure 1: Typing rules

VARIABLE
$$\frac{\mathrm{K} \vdash \bar{\tau} :: \bar{\kappa}^{\bar{\tau}}}{\mathrm{K}; \Gamma, x : \bar{\kappa} \triangleright \tau_1 \vdash x : \tau_1^{\bar{\tau}}}$$

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

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

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

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

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

Figure 2: Typing rules using cofinite quantification

kind in K' ($\alpha' :: \kappa' \in \mathrm{K}'$ 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. We refer the reader to [4] for further details.

3 Type soundness

The first step of our mechanical proof, in Coq, was to prove type soundness for the system of [4], starting from Aydemir and others proof for Core ML included in [1], 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 types and terms.

τ	$::=$	$n \mid \alpha \mid \tau_1 \to \tau_2$	type
σ	$::=$	$\bar{\kappa} \triangleright \tau$	polytype

$\bar{\tau}$ 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 $1 \ldots n$ inside types. The actual implementation has indices starting from 0, but we will start from 1 in this explanation. $\tau_1^{\bar{\tau}}$ is τ_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. $\mathrm{K} \vdash \tau :: \kappa$ is true when either $\kappa = \bullet$, or $\tau = \alpha$, $\alpha :: \kappa' \in \mathrm{K}$ and $\kappa' \models \kappa$. $\mathrm{K} \vdash \bar{\tau} :: \bar{\kappa}$ enforces this for every member of $\bar{\tau}$ and $\bar{\kappa}$ at identical positions, which is just equivalent to our condition $\mathrm{K} \vdash \theta : \mathrm{K}'$ 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 1, but they coincide if we instantiate L appropriately. For instance, if we use $\mathrm{dom}(\Gamma)$ for L in $\forall x \notin L$, this just amounts to ensuring that x is not already bound. Inside GENERALIZE, we could use $\mathrm{dom}(\mathrm{K}) \cup \mathrm{FV}_K(\Gamma)$ 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 L's.

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

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

Figure 3: Constraints

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. I am only aware of one other proof of type soundness including recursive types, in [2].

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 of the system. This was formalized through the use of functors. The signature for constraints 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. Here is the structure for basic definitions.

```
Module Type CstrIntf
Module Type CstIntf
Module MkDefs(Cstr:CstrIntf)(Const:CstIntf)
 ...
 Module Type DeltaIntf
 Module MkJudge(Delta:DeltaIntf)
  ...
  Module Type SndHypIntf
  Module Soundness(SH:SndHypIntf)
```

KIND GC

$$\frac{\mathrm{K},\mathrm{K}';\Gamma \vdash e:\tau \quad \mathrm{FV}_{\mathrm{K}}(E,\tau) \cap \mathrm{dom}(\mathrm{K}') = \emptyset}{\mathrm{K};\Gamma \vdash e:\tau}$$

CO-FINITE KIND GC

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

Figure 4: Kind discarding

...

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. Type soundness itself is in another functor, that requires some lemmas whose proof uses infrastructure lemmas on type judgments... This instantiation has been done for a constraint domain containing both polymorphic variants and records, and a fixpoint operator.

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 4 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 name has no impact on typability. Proofs went on smoothly, until I realized I needed the following inversion lemma.

$$\forall \mathrm{K}\Gamma e\tau,\ \mathrm{K};\Gamma \vdash_{GC} e:\tau \Rightarrow \exists \mathrm{K}',\ \mathrm{K},\mathrm{K}';\Gamma \vdash e:\tau$$

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 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. Once the problem becomes clear, one can see a

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

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

typinf$(\mathrm{K}, \Gamma, \mathsf{let}\ e_1\ \mathsf{in}\ e_2, \tau, \theta, L) =$
  let $\alpha = \mathrm{fresh}(L)$ in
  match **typinf**$(\mathrm{K}, \Gamma, e_1, \alpha, \theta, L \cup \{\alpha\})$ with
  $| \langle \mathrm{K}', \theta', L' \rangle \Rightarrow$
    let $\mathrm{K}_1 = \theta'(\mathrm{K}')$ and $\Gamma_1 = \theta'(\Gamma)$ in
    let $L_1 = \theta'(\mathrm{dom}(\mathrm{K}))$ and $\tau_1 = \theta'(\tau')$ in
    let $\langle \mathrm{K}_A, \sigma \rangle =$ **generalize**$(\mathrm{K}_1, \Gamma_1, L_1, \tau_1)$ in
    let $x = \mathrm{fresh}(\mathrm{dom}(E) \cup \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2))$ in
    typinf$(\mathrm{K}_A, (\Gamma, x : \sigma), e_2^x, \tau, \theta', L')$
  $| \langle\rangle \Rightarrow \langle\rangle$
  end

Figure 5: Type inference algorithm

much simpler solution: in most situations, it is actually sufficient to have KIND GC occur only just above ABSTRACTION and GENERALIZE, and the canonicalization lemma is just 100 lines.

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.

Unification has been a target of formal verification for a long time, with formal proofs as early as 1985 [5]. Here I just wrote down the algorithm in Coq, and proved both partial-correctness and completeness. The proof comes close to 1900 lines, as kinds need particular treatment, but there was no major stumbling block. The proof basically follows the algorithms, but there are two useful tricks. One concerns substitutions. Rather than using the relation "θ is more general than θ'" ($\exists\theta_1,\ \theta' = \theta_1 \circ \theta$), I used a the more direct "θ' extends θ" ($\forall\alpha,\ \theta'(\theta(\alpha)) = \theta'(\alpha)$). In the following it is noted $\theta' \sqsubseteq \theta$. When θ 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. Since unification is about first-order terms, namelessness has no impact here.

The next step is type inference itself. Again, this

SOUNDNESS
$\mathrm{typinf}(\mathrm{K}, \Gamma, \mathrm{e}, \tau, \theta, \mathrm{L}) = \langle \mathrm{K}', \theta', \mathrm{L}' \rangle \rightarrow$
$\mathrm{dom}(\theta) \cap \mathrm{dom}(\mathrm{K}) = \emptyset \rightarrow$
$\mathrm{FV}(\theta, \mathrm{K}, \Gamma, \tau) \subset L \rightarrow$
$\theta' \sqsubseteq \theta \wedge \mathrm{dom}(\theta') \cap \mathrm{dom}(\mathrm{K}') = \emptyset \wedge$
$\mathrm{K} \vdash \theta' : \theta'(\mathrm{K}') \wedge \theta'(\mathrm{K}'); \theta'(\Gamma) \vdash e : \theta'(\tau) \wedge$
$\mathrm{FV}(\theta', \mathrm{K}', \Gamma) \cup L \subset L'.$

PRINCIPALITY
$\mathrm{K}; \Gamma \vdash e : \theta(\tau) \rightarrow$
$\mathrm{K} \vdash \theta(\Gamma_1) \leq \Gamma \rightarrow \theta \sqsubseteq \theta_1 \rightarrow \mathrm{K}_1 \vdash \theta : \mathrm{K} \rightarrow$
$\mathrm{dom}(\theta_1) \cap \mathrm{dom}(\mathrm{K}_1) = \emptyset \rightarrow$
$\mathrm{dom}(\theta) \cup \mathrm{FV}(\theta_1, \mathrm{K}_1, \Gamma_1, \tau) \subset L \rightarrow$
$\exists \mathrm{K}'\theta' L',\ \mathrm{typinf}(\mathrm{K}_1, \Gamma_1, e, \tau, \theta_1, L) = \langle \mathrm{K}', \theta', L' \rangle \wedge$
$\exists\theta'',\ \mathrm{dom}(\theta'') \subset L' \setminus L \wedge \theta\theta'' \sqsubseteq \theta' \wedge \mathrm{K}' \vdash \theta\theta'' : \mathrm{K}.$

Figure 6: Properties of type inference

has been proved before for Core ML [6, 7], but to my knowledge never for a system containing equirecursive types. Proving both soundness and principality was rather painful, and took 3400 very full lines. 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 5. $[\bar\alpha]\tau$ stands for the generalization of τ with respect to $\bar\alpha$, obtained by replacing the occurrences of variables of $\bar\alpha$ in τ 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 6 we show slightly simplified versions, discarding well-formedness properties. $\mathrm{K} \vdash \Gamma_1 \leq \Gamma$ means that the polytypes of Γ are instances of those in Γ_1 (*i.e.* Γ_1 is more general than Γ). Due to the presence of kinds, the definition of the instantiation order gets a bit complicated.

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

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

$$\begin{array}{l}\mathrm{K} \vdash \forall \bar\alpha_1.\mathrm{K}_1 \triangleright \tau_1 \leq \forall \bar\alpha_2.\mathrm{K}_2 \triangleright \tau_2 \stackrel{\mathrm{def}}{=} \\ \quad \exists\theta,\ \mathrm{dom}(\theta) \subset \bar\alpha_1 \wedge \\ \quad\quad \mathrm{K}, \mathrm{K}_1 \vdash \theta : \mathrm{K}, \mathrm{K}_2 \wedge \theta(\tau_1) = \tau_2.\end{array}$$

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. Both renaming lemmas were harder to prove than expected (100 lines each). Contrary to what was suggested in [1], 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.

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 a program by rewriting it, but rather interpreting it with a virtual machine. I defined a SECD-like 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. Here the nameless representation of terms was handy, as it maps naturally to a stack machine.

I also proved that, if the rewriting based evaluation reaches a normal form, then evaluation with the abstract machine terminates with the same normal form. This required building a bisimulation between the two evaluation, and was trickier than expected.

6 Dependent types

As we pointer in section 4, the statements of many lemmas and theorems include lots of well-formedness properties, that 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 be bound, kinds should be valid, *etc*... A natural impulse is to use dependent types to encode these properties. Yet proofs from [1] 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 type schemes, 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 (T U:typ),
  Cstr.unique kc x = true →
  In (x,T) kr → In (x,U) kr → T = U.
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'.
```

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.)

The termination condition impacted also the way to write the unification and type inference algorithm. Since the termination of unification requires proving, I added an extra parameter indicating the number of steps to compute, and later proved that this parameter could always be made big enough so that it never goes down to 0. 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 it. I later came up with a smaller value, but it would be better to have it disappear completely. There is a well known technique to do that in Coq, which works by moving it to the universe of proofs (Prop), so that it will disappear during extraction. This requires a rather intensive use of dependent types, but it also makes the proof of completeness simpler. As a result the 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%. Note that, while the Program environment is intended to make definitions using dependent types easier, it is hard to control the terms it produces in detail. Since rewriting on dependently typed terms is particularly fragile, the definition had to be done completely by hand.

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 Conclusion

We have reached our first goal, providing a fully certified type checker and interpreter, but 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/#certint0908`

References

[1] 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 Proofs at `http://www.chargueraud.org/arthur/research/2007/binders/`.

[2] 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

[3] Owens, S.: A sound semantics for OCaml light. In: Proc. European Symposium on Programming. Volume 4960 of LNCS. (2008) 1–15

[4] Garrigue, J.: Simple type inference for structural polymorphism. In: The Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, Oregon (2002)

[5] Paulson, L.: Verifying the unification algorithm in LCF. Science of Computer Programming **5** (1985) 143–169

[6] Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm W in Isabelle/HOL. Journal of Automated Reasoning **23** (1999) 299–318

[7] 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