A Certified Implementation of ML with Structural Polymorphism and Recursive Types

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. We 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, and correctness of a stack-based interpreter, are also proved.[1]

1 Introduction

While many results have already been obtained in the mechanization of metatheory for ML [15, 6, 5, 13, 24] and pure type systems [3, 1], Objective Caml [14] has unique features which are not covered by existing works. For instance, polymorphic object and variant types require some form of structural polymorphism [9], combined with recursive types, and both of them do not map directly to usual type systems. Among the many other features, let us just cite the relaxed valued restriction [10], which accommodates side-effects in a smoother way, first class polymorphism [12] as used in polymorphic methods, labeled arguments [7], 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, we 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

[1] This article is an extended version of [11]; it also partly subsumes [9].

features, like in the mechanized metatheory of Standard ML [13]. We are not so much concerned about giving a full specification of the operational semantics, as in [19].

The contribution of this article 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, and even of type soundness for a system combining recursive types and a form of structural subsumption. Second, we have used extensively the techniques proposed in [1] 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. On the one hand we have been agreeably surprised by the compatibility of these techniques with explicit renaming (as necessary for type inference), but on the other hand one can easily get entangled in the plethora of quantifiers.

This article is organized as follows. In sections 2 and 3 we give examples of structural polymorphism, and define the notion of *constraint domain* which is central to our local constraint framework. In section 4 we define the type system, which is parameterized over constraint domains. Sections 5, 6 and 7 describe proofs for respectively type soundness, correctness of type inference, and safety and completeness of a stack-based interpreter. In section 8 we discuss our use of dependent types in proofs. Section 9 discusses the extraction process, and gives some examples using the extracted typechecker. Finally we discuss related works in section 10, before concluding.

The Coq proof scripts and the extracted code can be found at:

```
http://www.math.nagoya-u.ac.jp/~garrigue/papers/#certint1009
```

Having them at hand while reading this article should clarify many points. In particular, due to the size of some definitions, we could only include part of them in this article, and we refer the reader to the proof scripts for all the missing details.

2 Structural polymorphism

Before getting into the details of the type system, we give a short account of what we mean by structural polymorphism, how it can be formalized using a kind-based approach [18, 8], and how it applies to Objective Caml. We assume that the reader is already familiar with Core ML and its type system.

We present here various forms of structural polymorphism, on a gradual scale of increasing complexity. In order to uniformize the presentation, we write $K \triangleright \tau$ for types, where K is a kinding environment, containing constraints for individual type variables, under which the open type τ is to be understood. We use a few built-in functions, to obtain interesting types: integer addition $+$, string concatenation $\uparrow$, conversions $\mathit{string\_of\_int} : \mathit{string} \rightarrow \mathit{int}$ and $\mathit{float} : \mathit{int} \rightarrow \mathit{float}$.

2.1 Records à la Ohori

Arguably this is the simplest form of structural polymorphism considered in the literature. Record values have monomorphic types, and polymorphism is only used for typing field access.

$$\begin{array}{ll}
\{name = \texttt{"Jacques"}, age = 40\} & : \{name : string, age : int\} \\
\mathsf{fun}\ x \rightarrow x.age + 1 & : \alpha :: \{age : int\} \triangleright \alpha \rightarrow int \\
\mathsf{fun}\ x \rightarrow x.name \uparrow \texttt{" is "} \uparrow string\_of\_int\ x.age & \\
\qquad : \alpha :: \{name : string, age : int\} \triangleright \alpha \rightarrow string &
\end{array}$$

Intuitively, the kinding $\alpha :: \{age : int\}$ means that α should at least have a field *age*, and this field should have type *int*. Accessing several fields result in a kind containing all of them. Two kinds are compatible if they agree on the types of their common fields. A type satisfies a kind if all the required fields are provided, with the correct types.

Note that the above formulation is a direct adaptation of Ohori's. Fitting it inside the local constraint framework of section 3 will require a small change in presentation, without changing expressive power.

Objective Caml objects can be seen as a direct application of Ohori's record polymorphism (eventhough the implementation is based on Rémy's work [21].) The concrete syntax inlines kinds inside types.

```
object method name = "Jacques" method age = 40 end;;
- : < age : int; name : string > = <obj>
let show x = x#name ^ " is " ^ string_of_int x#age;;
val show : < age : int; name : string; .. > -> string
```

2.2 Records and variants with masking

Ohori's records are weak in that record values have only structurally monomorphic types. We might want to allow building lists of records containing different fields, with only part of them common to all members. This requires giving polymorphic types not only to field access, but also to record values. A type system allowing it was first proposed by Rémy [21], but here we use a more intuitive formalism [8], which is also closer to Ohori's.

$$\begin{array}{l}
\{name = \texttt{"Jacques"}, age = 40\} \\
\qquad\qquad : \alpha :: (\{name : string, age : int\}, \emptyset, \{name, age\}) \triangleright \alpha \\
\mathsf{fun}\ x \rightarrow x.age + 1 \qquad : \ \alpha :: (\{age : int\}, \{age\}, \mathscr{L}) \triangleright \alpha \rightarrow int \\
\mathsf{let}\ l_1 = [\{name = \texttt{"Jacques"}, age = 40\}, \{name = \texttt{"Marie"}, weight = 16\}] \\
l_1\ :\ \alpha :: (\{name : string, age : int, weight : int\}, \emptyset, \{name\}) \triangleright \alpha\ list
\end{array}$$

Kinds are now represented by a triple (T, L, U). Along with the type of each field, we have two sets of labels. Required labels L (union of all field accesses, a lower bound) form a subset of available labels U (intersection of the possible record values, an upper bound). $\mathscr{L}$ is the set of all labels. You can see how fields are masked in the third example: while we have types for the *name, age* and *weight* fields, only the *name* field is accessible.

The combined type $\alpha :: (\{name : string, age : int\}, \{age\}, \{name, age\})$ is an acceptable description for both $\{name = \texttt{"Jacques"}, age = 40\}$, which makes *name* and *age* available, and $\mathsf{fun}\ x \rightarrow x.age + 1$, which only requires *age*. The need for two distinct sets of labels stems from the natural appearance of such combined types during type inference. Since attempting to access an unavailable label would be a type error, this also explains why, for a constraint kind to be meaningful (to have a solution), the set of required labels should be included in the set of available labels.

By duality, the same types can be used to describe polymorphic variants. The basic idea is that case-analysis of a variant can receive the same type as a record, while the variant itself would get the type of a field accessor.

$$\begin{array}{l}
\mathit{Number}(5) \qquad\qquad\qquad : \ \alpha :: (\{\mathit{Number} : \mathit{int}\}, \{\mathit{Number}\}, \mathscr{L}) \triangleright \alpha \\
\mathsf{let}\ l_2 = [\mathit{Number}(5), \mathit{Face}(\text{"}\mathtt{King}\text{"})] \\
l_2 \ : \ \alpha :: (\{\mathit{Number} : \mathit{int}, \mathit{Face} : \mathit{string}\}, \{\mathit{Number}, \mathit{Face}\}, \mathscr{L}) \triangleright \alpha\ \mathit{list} \\
\mathsf{let}\ f_1 = \mathsf{function}\ \mathit{Number}(n) \to \mathit{string\_of\_int}\ n \\
\qquad\qquad\qquad \mid \mathit{Face}(\mathit{name}) \to \mathit{name} \\
f_1 \ : \ \alpha :: (\{\mathit{Number} : \mathit{int}, \mathit{Face} : \mathit{string}\}, \emptyset, \{\mathit{Number}, \mathit{Face}\}) \triangleright \alpha \to \mathit{string}
\end{array}$$

Our two sets of labels have now different meanings. The first one is the set of present constructors, or required cases, which must be handled by case-analysis; the second one is the set of handled constructors, or available cases, which is accepted by all case-analyses.

An advantage of not using predefined sum types, is that we can make case analysis modular. Consider the following function f_2, which uses a special syntax for dispatch. g (respectively h) will only receive A or B (respectively C or D). This can be reflected at the type level by requiring them to handle only relevant cases.

$$\begin{array}{l}
\mathsf{let}\ f_2 = \mathsf{fun}\ g \to \mathsf{fun}\ h \to \mathsf{function}\ (A|B)\ \mathit{as}\ x \to g\ x \mid (C|D)\ \mathit{as}\ x \to h\ x \\
f_2 \ : \ \alpha_1 :: (\{A : \alpha_A, B : \alpha_B\}, \{A, B\}, \mathscr{L}), \alpha_2 :: (\{C : \alpha_C, D : \alpha_D\}, \{C, D\}, \mathscr{L}), \\
\qquad \alpha_3 :: (\{A : \alpha_A, B : \alpha_B, C : \alpha_C, D : \alpha_D\}, \emptyset, \{A, B, C, D\}) \\
\qquad \triangleright (\alpha_1 \to \alpha) \to (\alpha_2 \to \alpha) \to \alpha_3 \to \alpha
\end{array}$$

Our choice of keeping all constraints local (constraining only one variable) makes our system slightly weaker than Rémy's, which has row and presence variables. For instance we have no way to relate the set of required cases in α_1 and α_3, which forces us to make the safe assumption $\{A, B\}$ in α_1. Rémy's system handles requirement for each constructor as an independent presence variable, which may be shared between two different variant types. This allows more precise typing, but at the cost of harder to understand types, with lots of variables, of different sorts. Experience suggests that kinded variables, by reducing the number of variables to one by record or variant type, and keeping only one sort of type variables, make reading types much easier.

2.3 Discarding masked types

One may wonder about why one should keep all field types in the type of l_1, a list of records with some masked fields. If fields *age* and *weight* are actually unavailable, why should their types matter? Clearly, this is not the case with subtyping, which would allow to discard not only the fields, but also their types. However, our requirement of principal type inference makes impossible to simply forget the type.

For instance, let us consider the list:

$$\mathsf{let}\ l_3 = [\{\mathit{name} = \texttt{"Jacques"}, \mathit{age} = 40\}, \{\mathit{name} = \texttt{"Marie"}, \mathit{age} = 4.5\}]$$

The two records disagree on the type for *age*, either *int* or *float*. This should trigger a type error. Yet, if we choose to discard the type for *age* in l_1, we would be able to type $l_1 @ [\{name = \texttt{"Marie"}, age = 4.5\}]$, which contains the untypable l_3 as a sublist.

A solution to this problem is to only trigger an error when a field with conflicting types is used. This amounts to allowing conjunctive types in the types of non-required fields, only forcing them to be equal when they are required. Then we can give the following type to l_3.

$$l_3 \ : \ \alpha :: (\{name : string, age : int \wedge float\}, \emptyset, \{name, age\}) \triangleright \alpha$$

Since the two types for the field *age* are incompatible, it cannot be accessed. And if it disappears from the list of available fields, we can just drop these types as useless information.

While the absence of error may seem strange in the above case (even though it is necessary for coherence), it is a natural way to allow the construction of heterogeneous collections of objects in a system without subtyping.

The same mechanism also applies to polymorphic variants, and is used in Objective Caml since version 3. This allows principal type inference of f_5 for the following case, solving the long lasting problem of “masked but not discarded” types.

$$\begin{array}{l}
\mathsf{let}\ f_3 = \mathsf{function}\ Number(n) \rightarrow n \mid Face() \rightarrow 15 \\
f_3 \ : \ \alpha :: (\{Number : int, Face : unit\}, \emptyset, \{Number, Face\}) \triangleright \alpha \rightarrow int \\
\mathsf{let}\ f_4 = \mathsf{function}\ Number(n) \rightarrow n/2 \\
f_4 \ : \ \alpha :: (\{Number : int\}, \emptyset, \{Number\}) \triangleright \alpha \rightarrow int \\
\mathsf{let}\ f_5 = \mathsf{fun}\ x \rightarrow (f_1(x), f_3(x), f_4(x)) \\
f_5 \ : \ \alpha :: (\{Number : int\}, \emptyset, \{Number\}) \triangleright \alpha \rightarrow (string \times int \times int)
\end{array}$$

Here is the corresponding Objective Caml syntax for the above examples.

```
let f1 = function ‘Number n -> string_of_int n | ‘Face nm -> nm;;
val f1 : [< ‘Face of string | ‘Number of int ] -> string
let f2 g h = function ‘A|‘B as x -> g x | ‘C|‘D as x -> h x;;
val f2 : ([> ‘A | ‘B ] -> ’a) ->
         ([> ‘C | ‘D ] -> ’a) -> [< ‘A | ‘B | ‘C | ‘D ] -> ’a
let f3 = function ‘Number n -> n | ‘Face -> 15;;
val f3 : [< ‘Face | ‘Number of int ] -> int
let f4 = function ‘Number n -> n/2
val f4 : [< ‘Number of int ] -> int
let f5 x = (f1 x, f3 x, f4 x);;
val f5 : [< ‘Number of int ] -> string * int * int
```

2.4 Recursive types

While structural polymorphism in itself does not require recursive types, many applications do. For instance, one might want to define a map function on lists defined as polymorphic variants.

$$\begin{array}{l}
\mathsf{let}\ map\ f = \mathsf{function}\ Nil() \rightarrow Nil() \mid Cons(a,l) \rightarrow Cons(f\ a, map\ f\ l) \\
map \ : \ \gamma :: (\{Nil : unit, Cons : \alpha \times \gamma\}, \emptyset, \{Nil, Cons\}), \\
\qquad \delta :: (\{Nil : unit, Cons : \beta \times \delta\}, \{Nil, Cons\}, \mathscr{L}) \triangleright (\alpha \rightarrow \beta) \rightarrow \gamma \rightarrow \delta
\end{array}$$

Note that, thanks to the kinding environment, we don't need to explicitly introduce recursive types to obtain recursion. It is sufficient to make the kinding environment implicitly recursive.

Another application of recursion is to have methods returning the object itself.

$$\begin{array}{l}
\mathsf{let\ rec}\ \mathit{point}\, x = \{\mathit{current} = x, \mathit{move} = \mathsf{fun}\ d \rightarrow \mathit{point}(x+d)\} \\
\mathit{point}\ :\ \alpha :: (\{\mathit{current} : \mathit{int}, \mathit{move} : \mathit{int} \rightarrow \alpha\}, \{\mathit{current}, \mathit{point}\}, \{\mathit{current}, \mathit{point}\}) \\
\qquad\quad \triangleright\, \mathit{int} \rightarrow \alpha
\end{array}$$

In the Objective Caml syntax, the keyword "as" is used to express kind sharing, which is not limited to recursive types.

```
let rec map f =
  function ‘Nil -> ‘Nil | ‘Cons(a,l) -> ‘Cons(f a, map f l);;
val map : ('a -> 'b) ->
        ([< ‘Cons of 'a * 'c | ‘Nil ] as 'c) ->
        ([> ‘Cons of 'b * 'd | ‘Nil ] as 'd)
let rec point x =
  object method current = x method move d = point (x+d) end;;
val point : int -> (< current : int; move : int -> 'a > as 'a)
```

3 Constraint domains

The examples of the previous section should have demonstrated that a number of type system based on structural polymorphism share the same basic formulation, with some technical differences. An important point, which becomes apparent in examples with masking, is that one can split the kind information into two parts: a constraint part C, which indicates which labels (fields or constructors) the constrained type variable should contain, and a relational part R, which associate types with these labels. The notion of constraint domain, central to our formalization of structural polymorphism, attempts to abstract from those differences in the most general possible way.

Contrary to usual constraint-based type systems such as HM(X) [16], our local constraints are grafted on top of the Hindley-Milner type system, rather than mixing constraints and types at the same level. This means that we can reduce requirements on constraints to a minimum: their interaction with the rest of the type system will be minimal anyway. In particular, we do not introduce any syntax for constraints: they are black boxes, and need just be able to answer some questions. While not essential, we think this freedom is important, as the choice of how to represent constraints is relevant to their understanding. Except for this extra freedom, nothing would prevent one to formalize our constraint domains as special kinds of cylindric algebras, which can be plugged into HM(X).

A constraint domain describes a class of constraints, and how they interact with the type system. A particular instance of the type system may contain several constraint domains, as long as all their operations and values are clearly distinguished. For simplicity, we will only consider type systems operating on a single constraint domain.

Definition 1. *A constraint domain $\mathscr{C}$ is composed of the following items.*

1. *A theory $\mathcal{T}_\mathcal{C}$ with an entailment relation $\models$ satisfying the following properties*
   (a) *There is a constraint $\bot$, such that for any C we have $\bot \models C$.*
   (b) *A constraint C is said to be* invalid *when $C \models \bot$. Validity is decidable.*
   (c) *Entailment is reflexive and transitive: $C \models C$; if $C \models C'$ and $C' \models C''$ then $C \models C''$.*
   (d) *For any two constraints C and C', there is a constraint $C \wedge C'$ such that $C \wedge C' \models C$, $C \wedge C' \models C'$, and for all C'' such that $C'' \models C$ and $C'' \models C'$, we have $C'' \models C \wedge C'$.*
2. *A set $\mathcal{L}$ of labels, and an observation relation $\vdash$ checking some atomic properties of a constraint: $C \vdash p(a)$ where p and a are respectively a predicate and a label for the domain. Observation should be compatible with entailment:*

   $$\textit{If } C \models C' \textit{ and } C' \vdash p(a) \textit{ then } C \vdash p(a).$$

3. *A specific predicate* unique *which requires* coherence*, i.e. the type associated to a label must be unique. Any kind (C,R) must satisfy the rule:*

   $$\forall x \tau_1 \tau_2,\ x : \tau_1 \in R \wedge x : \tau_2 \in R \wedge C \vdash \mathsf{unique}(x) \Rightarrow \tau_1 = \tau_2$$

   i.e. if a label x exhibits unique(x)*, then R can only associate a single type to x.*

Our requirements on the entailment relation $\models$ are as free as one can get. Basically, we only need a way to distinguish valid constraints from invalid ones, and build the intersection of two constraints.

The observation relation $\vdash$ is a consequence of the representation independence of our constraints: we need an explicit way to relate them to the rest of the world. In particular unique interfaces constraints, which are semantic, with syntactic type equality.

To form a kind, each constraint will be coupled with a relation R associating types to observable labels. R need not always describe a function: coherence is only enforced on labels exhibiting the unique property.

We say that a constraint is *exact* if it cannot be further refined, *i.e.* if it is only entailed by itself and $\bot$ (modulo equivalence with respect to entailment). This is equivalent to a ground type in the world of constraints. Whether a constraint is exact or not does not impact anything in the theory, but knowing it may be helpful when reading types.

Note that the definition of constraint domain we give here is simpler but weaker than the one in [9]. The extra power of the original definition was needed to handle a form of dependent typing of pattern-matching, which we have omitted here since it is not used in Objective Caml.

We now consider the constraint domains associated to examples in the previous sections.

Records à la Ohori The original formulation did mix required fields and their types together. We have to distinguish the two, to adapt to our framework. Moreover, since we do not extend monomorphic types themselves, monomorphic records should also be described by exact constraints.

A constraint is a pair (L,x) of a finite set of labels L, together with a mark x distinguishing exact types (1) from refinable ones (0). Entailment on refinable types is containment: $(L,x) \models (L',0)$ iff $L \supset L'$, which makes set union the conjunction operation. For exact types entailment is only reflexive. All such constraints are valid, so we must add a $\bot$ element. We do not need to observe anything particular, so we just use $C \vdash \mathit{true}(l)$ for any C and l, requiring coherence for all labels.

Records with maskable fields Constraints are represented by a pair of sets (L,U), L a finite set of accessed labels, and U either the set of all labels $\mathscr{L}$, or a finite set of available labels. Entailment is defined by $(L,U) \models (L',U')$ iff $L \supset L'$ and $U \subset U'$. We can choose $(\mathscr{L},\emptyset)$ as $\bot$. The validity check is $(L,U) \models \bot$ when $L \not\subset U$. One can observe required labels: $(L,U) \vdash \mathit{req}(l)$ iff $l \in L$. We can either require coherence for all labels, and obtain records without discarding, or only for required ones, which allows conjunctive typing, by delaying the equality constraint until the field is accessed.

Since the constraint domain for records with maskable fields also applies to polymorphic variants, and subsumes Ohori-style records, one may wonder why we do not choose it as canonical constraint domain, avoiding the extra complexity of abstraction. Independently of the advantage of clearer reasoning provided by abstraction, experience tells us that we practically need the freedom it provides. Indeed, when adding polymorphic variants to Objective Caml, we have experimented with different variations. A first one is that the constraint $(\emptyset,\emptyset)$, while meaningful for records (it denotes a record with no fields), should be invalid for polymorphic variants, as it denotes the empty type, and allowing it would delay the detection of some errors. A second variation, on top of the first one, was to add an extra predicate for *lone* labels, *i.e.* polymorphic variants with only one possible case. Since this label cannot disappear (as it would result in an empty type), it is principal to require it to be coherent. This second variation was eventually dropped in Objective Caml for other reasons (the first one is kept), but it demonstrates how hard it is to know in advance what exact specification of the constraint domain will be right in practice.

4 Type system

We now present the type system we will work with. While this description is intended to be self-contained, the original formalization of local constraints [9] may provide more explanations.

Terms are the usual ones: variables, constants, functions, application and let-binding. We intend to provide all other constructs through constants and δ-rules.

$$e ::= x \mid c \mid \lambda x.e \mid e\, e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

Types are less usual.

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

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

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

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

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

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

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

Fig. 1. Typing rules (original)

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

Constant
$$\frac{\mathrm{K} \vdash \bar{\tau} :: \bar{\kappa}^{\bar{\tau}} \quad \mathrm{Tconst}(c) = \bar{\kappa} \triangleright \tau_1}{\mathrm{K};\Gamma \vdash c : \tau_1^{\bar{\tau}}}$$

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

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

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

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

Fig. 2. Typing rules using cofinite quantification

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, as a *kinding environment* associates them with their respective *kinds*. When they are associated with a concrete kind, they actually denote structural types, like records or variants. Such types are described by a pair (C,R) of a local constraint C and a relation R between labels and 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 are allowed to be recursive, we can use them to define recursive types (where the recursion must necessarily go through kinds.) Since type variables only make sense in presence of a kinding environment, *polytypes* have to include a kinding environment for the variables they quantify; *i.e.*, in $\forall\bar{\alpha}.\mathrm{K} \triangleright \tau$, K is such that $\mathsf{dom}(\mathrm{K}) = \{\bar{\alpha}\}$, and the variables of $\bar{\alpha}$ may appear both inside the kinds of K and in τ. A good way to understand these definitions 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, as described in sections 2 and 3. Note that the type system only needs to know about entailment, and whether a kind is valid or not, *i.e.* whether the constraint is valid and coherence is satisfied. By extension we also use the notation $\kappa' \models \kappa$ for kinds, *i.e.* $(C',R') \models (C,R)$ iff $C' \models C$ and $R \subset R'$.

$$
\begin{array}{lll}
\text{R-ABS} & \text{R-DELTA} & \text{R-APP}_1 \\
(\lambda e_1)\, v_2 \longrightarrow e_1^{v_2} & \dfrac{e = \mathsf{Delta.reduce}\ c\ [v_1;\ldots;v_n]}{c\, v_1\ \ldots\ v_n \longrightarrow e} & \dfrac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2} \\
\text{R-LET} & \text{R-LET}_1 & \text{R-APP}_2 \\
\mathsf{let}\ v_1\ \mathsf{in}\ e_2 \longrightarrow e_2^{v_1} & \dfrac{e_1 \longrightarrow e_1'}{\mathsf{let}\ e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ e_1'\ \mathsf{in}\ e_2} & \dfrac{e_2 \longrightarrow e_2'}{v_1\, e_2 \longrightarrow v_1\, e_2'}
\end{array}
$$

Fig. 3. Reduction rules

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 $\mathrm{K};\Gamma \vdash e : \tau$, where the variables kinded in K may appear in both Γ and τ.

The typing rules are given in Fig. 1. $\mathrm{K} \vdash \theta : \mathrm{K}'$ means that the substitution θ (defined as usual) 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 kind in K′ ($\alpha' :: \kappa' \in \mathrm{K}'$ and $\kappa' \models \theta(\kappa)$). Tconst assigns closed type schemes to constants.

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 (denoted by $\mathrm{K}|_B$), and a non-generalized part for the rest (denoted by $\mathrm{K}|_{\mathrm{K}\setminus B}$). When determining which type variables can be generalized, we must be careful that for any type variable accessible from Γ, the type variables appearing in its kind (inside K) are also accessible. For this reason FV takes K as parameter.

$$
\begin{array}{rcll}
\mathsf{FV}_\mathrm{K}(\forall \alpha_1 \ldots \alpha_n.\mathrm{K}' \triangleright \tau) &=& \mathsf{FV}_{\mathrm{K},\mathrm{K}'}(\tau) \setminus \{\alpha_1,\ldots,\alpha_n\} & \\
\mathsf{FV}_\mathrm{K}(\alpha) &=& \{\alpha\} \cup \mathsf{FV}_\mathrm{K}(R) & \alpha :: (C,R) \in \mathrm{K} \\
\mathsf{FV}_\mathrm{K}(\alpha) &=& \{\alpha\} & \alpha :: \bullet \in \mathrm{K} \\
\mathsf{FV}_\mathrm{K}(\tau_1 \to \tau_2) &=& \mathsf{FV}_\mathrm{K}(\tau_1) \cup \mathsf{FV}_\mathrm{K}(\tau_2) &
\end{array}
$$

It may be difficult to understand this type system in abstract form. We have already given examples of concrete constraint domains in sections 2 and 3. Types for specific constants corresponding to usual syntactic constructs appear in Fig. 7.

5 Type soundness

The first step of our mechanical proof, using Coq [22], was to prove type soundness for the system described in the previous section, 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.

Fig. 2 contains the typing rules adapted to locally nameless cofinite quantification, and the reduction rules are in Fig. 3. They both use locally nameless terms and types.

$$
\begin{array}{lcll}
e &::=& n \mid x \mid c \mid \lambda e \mid e\ e \mid \mathsf{let}\ e\ \mathsf{in}\ e & \text{term} \\
\tau &::=& n \mid \alpha \mid \tau_1 \to \tau_2 & \text{type} \\
\kappa &::=& \bullet \mid (C, \{l_1 : \tau_1, \ldots, l_n : \tau_n\}) & \text{kind} \\
\sigma &::=& \bar{\kappa} \triangleright \tau & \text{polytype}
\end{array}
$$

```
Module Type CstrIntf.
 Parameter cstr attr : Set.              (* types for abstract constraints and labels *)
 Parameter valid : cstr → Prop.                            (* validity of a constraint *)
 Parameter valid_dec : ∀c, {valid c} + {¬valid c}.           (* validity is decidable *)
 Parameter eq_dec : ∀xy : attr, {x = y} + {x ≠ y}.      (* label equality is decidable *)
 Parameter unique : cstr → attr → bool.                     (* uniqueness of a label *)
 Parameter ⊓ : cstr → cstr → cstr.                                    (* conjunction *)
 Parameter ⊨ : cstr → cstr → Prop.                (* entailment between constraints *)
 Parameter entails_refl : ∀c, c ⊨ c.                      (* properties of entailment *)
 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.
Module Type CstIntf.
 Parameter const : Set.                                                 (* constants *)
 Parameter arity : const → nat.                                       (* their arity *)
```

Fig. 4. Interfaces for constraints and constants

$\bar{\tau}$ 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[2]. $\tau_1^{\bar{\tau}}$ 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, with scope the hypotheses on the right of the quantifier. Each L appearing in a derivation is existentially quantified (*i.e.* one chooses a concrete L when building the derivation), but has to be finite, to allow an infinite number of variables outside of L. At first, the rules may look very different from those in Fig. 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 $\mathsf{dom}(\mathrm{K}) \cup \mathsf{FV}_{\mathrm{K}}(\Gamma)$ 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

[2] The implementation has indices starting from 0, but we will start from 1 in this explanation.

```
Module MkDefs (Cstr : CstrIntf) (Const : CstIntf).
 Inductive typ : Set := ...                                        (* our types *)
 Inductive type : typ → Prop := ...                        (* well-formed types *)
 Definition sch := typ * list kind.                           (* type schemes *)
 Inductive trm : Set := ...                                        (* our terms *)
 ...
 Module Type DeltaIntf.
  Parameter Tconst : Const.const → sch.                 (* types of constants *)
  Parameter reduce : ∀c el, (list_for_n value (1 + Const.arity c) el) → trm.   (* δ-rules *)
  ...                                                   (* 3 more properties *)
 Module MkJudge (Delta : DeltaIntf).
  Inductive ⊢ : kenv → env → trm → typ → Prop := ...   (* the typing judgment *)
  Inductive ⟶ : trm → trm → Prop := ...              (* the reduction relation *)
  Inductive value : trm → Prop := ...                                 (* values *)
  ...
  Module Type SndHypIntf.
   Parameter delta_typed : ∀c el vl K Γ τ,
     (K;Γ ⊢ const_app c el : τ) → (K;Γ ⊢ Delta.reduce c el vl : τ).
  Module MkSound (SH : SndHypIntf).
   Theorem preservation : ∀KΓee′τ, (K;Γ ⊢ e : τ) → (e ⟶ e′) → (K;Γ ⊢ e′ : τ).
   Theorem progress : ∀Keτ, (K;∅ ⊢ e : τ) → (value e ∨ ∃e′, e ⟶ e′).
```

Fig. 5. Module structure

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.

We also added a formalism for constants and δ-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 of the system. This was formalized through the use of functors. The signature for constraints and constants is in Fig. 4, and an outline of the module structure of the soundness proof (including the statements proved) is in Fig. 5. We omit here the definitions of terms, types, typing derivations, and reduction, as they just implement the locally nameless definitions we described above. A value is either a λ-abstraction, or a constant applied to a list of values of length less than its arity.

This approach worked well, but there are some drawbacks. One is that since some definitions depend on parameters of the framework, and some of the proofs required by the framework depend on those definitions, we need nested functors, and the instantiation of the framework with a *constraint domain* looks like a "dialogue": we repeatedly alternate domain-specific definitions, and applications of framework functors to those definitions, each new definition using the result of the previous functor application. The

```
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).
 Definition unique c v := set_mem eq_nat_dec v (low c).
 ...
```

Fig. 6. Constraint domain for polymorphic variants and records

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

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

Fig. 7. Types and δ-rules for constants

problem appears not so much with constraints themselves, but rather with constants and δ-rules. 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 δ-rules. We enforce the completeness of δ-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. The same problem is known to occur in programs using heavily ML functors, so this is not specific to Coq. But the level of stratification of definitions we see in this proof rarely occurs in programs.

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 Fig. 6. A constraint combines a sort — whether the constraint denotes a variant (Ksum), a record (Kprod), or is invalid (Kbot) —, a lower bound (low), and an upper bound (high) of the set of used labels. We write $\langle\rangle$ for None, which denotes here the set of

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

Co-finite Kind GC
$$\frac{\forall \bar{\alpha} \notin L \quad \mathsf{K},\bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}};\Gamma \vdash e : \tau}{\mathsf{K};\Gamma \vdash e : \tau}$$

Fig. 8. Kind discarding

all possible labels. Validity requires the sort not to be Kbot, and the lower bound to be included in the upper bound. unique is true for all the labels in the lower bound. Constants and δ-rules are in Fig. 7, using the nameful syntax for types. You can see the duality between variants and records, at least for tag and get.

Proofs for the constraint domain were somewhat lengthy. As indicated in Fig. 5, one must prove delta_typed for each δ-rule, and the constants involved are n-ary, with rather complex types. This requires very boring lemmas, to handle all cases. The clear separation between the type system and the constraint domain avoided polluting the soundness proof with those details.

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. Fig. 8 shows both the normal and cofinite forms. Again one can see the elegance of the cofinite version, where there is no need to specify which kinds are irrelevant: just the ones whose names have no impact on typability. Proofs went on smoothly, until I realized that I needed the following inversion lemma, relating derivations using it ($\vdash_{GC}$ is $\vdash$ extended with the rule Kind GC), and those without it.

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

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 in the cofinite version. And this is just impossible without a true renaming lemma. I got stuck there for a while, unable to see what was going wrong[3]. 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.

Lemma typing_rename : $\forall \mathsf{K}\,\Gamma\, x\, y\, \sigma\, \Gamma'\, e\, \tau$,
$\mathsf{K};\Gamma, x{:}\sigma, \Gamma' \vdash e : \tau \rightarrow y \notin \mathsf{dom}(\Gamma,\Gamma') \cup \{x\} \cup \mathsf{FV}(e) \rightarrow \mathsf{K};\Gamma, y{:}\sigma, \Gamma' \vdash [y/x]e : \tau.$

[3] Thanks to Arthur Charguéraud for opening my eyes.

Lemma typing_rename_typ : $\forall \mathrm{K}\Gamma\,\bar{\kappa}\,\tau\,\bar{\alpha}\,\bar{\alpha}'\,e,$
$\bar{\alpha} \notin \mathsf{FV}(\Gamma) \cup \mathsf{FV}(\bar{\kappa} \triangleright \tau) \cup \mathsf{dom}(\mathrm{K}) \cup \mathsf{FV}(\mathrm{K}) \to \bar{\alpha}' \notin \mathsf{dom}(\mathrm{K}) \cup \{\bar{\alpha}\} \to$
$\mathrm{K}, \bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}}; \Gamma \vdash e : \tau^{\bar{\alpha}} \to \mathrm{K}, \bar{\alpha}' :: \bar{\kappa}^{\bar{\alpha}'}; \Gamma \vdash e : \tau^{\bar{\alpha}'}.$

The 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 by a direct induction, and they ended up being of the same length. On the other hand, one could argue that the direct proof was easy precisely thanks to cofinite quantification, which eschews the need for extra machinery.

Once the essence of the problem (*i.e.* commutation of quantifiers) becomes clear, one can see a much simpler solution: in most situations, it is actually sufficient to have KIND GC occur only just above ABSTRACTION and GENERALIZE, and the canonization lemma is just 100 lines, as it doesn't change the quantifier structure of the proof. 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, which are used by the definitions themselves, allow to manipulate this state transparently.

6 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.

6.1 Unification

Unification has been a target of formal verification for a long time, with formal proofs as early as 1985 [20]. Here we just wrote down the algorithm in Coq, and proved both correctness and completeness. A rule-based version of the algorithm can be found in [9]. The following statements were proved:

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

Theorem unify_types : $\forall h l \mathrm{K} \theta$, unify $h\ l\ \mathrm{K}\ \theta = \langle \mathrm{K}', \theta' \rangle \to$ unifies $\theta'\ l.$

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

Theorem unify_mgu : $\forall h l \mathrm{K}_0 \mathrm{K} \theta,$
unify $h\ l\ \mathrm{K}_0$ id $= \langle \mathrm{K}, \theta \rangle \to$ unifies $\theta'\ l \to \mathrm{K}_0 \vdash \theta' : \mathrm{K}' \to \theta' \sqsupseteq \theta \wedge \mathrm{K} \vdash \theta : \mathrm{K}'.$

Theorem unify_complete : $\forall \mathrm{K}\,\theta\,\mathrm{K}_0\,l\,h,$
unifies $\theta\ l \to \mathrm{K}_0 \vdash \theta : \mathrm{K} \to$ size_pairs id $\mathrm{K}_0\ l < h \to$ unify $h\ l\ \mathrm{K}_0$ id $\neq \langle\rangle.$

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

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

Fig. 9. Type inference algorithm

The first argument to unify is the number of type variables, which is used to enforce termination. Then comes a list of type pairs to unify and the original kinding 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 there are two useful tricks. One concerns substitutions. Rather than using the relation "θ is more general than θ'" ($\exists \theta_1,\ \theta' = \theta_1 \circ \theta$), we used the more direct "θ' extends θ" ($\forall \alpha,\ \theta'(\theta(\alpha)) = \theta'(\alpha)$). In the above theorem it is noted $\theta' \sqsupseteq \theta$. 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. 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.

6.2 Inference

The next step is type inference itself. Again, correctness has been proved before for Core ML [15, 6, 24], but to our 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 Fig. 9. $[\bar{\alpha}]\tau$ 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.

$$\begin{array}{ll}
\mathsf{Theorem\ soundness}: \forall \mathrm{K}\Gamma e\,\tau\,\theta L \mathrm{K}'\,\theta' L', & \mathsf{Theorem\ principality}: \forall \mathrm{K}\Gamma e\,\tau\,\theta\,\mathrm{K}_1\,\theta_1\,L, \\
\quad \mathsf{typinf}(\mathrm{K},\Gamma,e,\tau,\theta,L) = \langle \mathrm{K}',\theta',L'\rangle \rightarrow & \quad \mathrm{K};\theta(\Gamma) \vdash e : \theta(\tau) \rightarrow \mathrm{K}_1 \vdash \theta : \mathrm{K} \rightarrow \\
\quad \mathsf{dom}(\theta) \cap \mathsf{dom}(\mathrm{K}) = \emptyset \rightarrow & \quad \theta \sqsupseteq \theta_1 \rightarrow \mathsf{dom}(\theta_1) \cap \mathsf{dom}(\mathrm{K}_1) = \emptyset \rightarrow \\
\quad \mathsf{FV}(\theta,\mathrm{K},\Gamma,\tau) \subset L \rightarrow & \quad \mathsf{dom}(\theta) \cup \mathsf{FV}(\theta_1,\mathrm{K}_1,\Gamma,\tau) \subset L \rightarrow \\
\quad \theta'(\mathrm{K}');\theta'(\Gamma) \vdash e : \theta'(\tau) \wedge & \quad \exists \mathrm{K}'\theta' L', \\
\quad \mathrm{K} \vdash \theta' : \theta'(\mathrm{K}') \wedge \theta' \sqsupseteq \theta \wedge & \quad\ \mathsf{typinf}(\mathrm{K}_1,\Gamma,e,\tau,\theta_1,L) = \langle \mathrm{K}',\theta',L'\rangle \wedge \\
\quad \mathsf{FV}(\theta',\mathrm{K}',\Gamma) \cup L \subset L' \wedge & \quad\ \exists \theta'', \mathrm{K}' \vdash \theta\theta'' : \mathrm{K} \wedge \theta\theta'' \sqsupseteq \theta' \wedge \\
\quad \mathsf{dom}(\theta') \cap \mathsf{dom}(\mathrm{K}') = \emptyset. & \qquad\quad \mathsf{dom}(\theta'') \subset L' \setminus L.
\end{array}$$

Fig. 10. Properties of type inference

Due to the large number of side-conditions required, the statements for the inductive versions of soundness of principality become very long. In Fig. 10 we show slightly simplified versions, omitting well-formedness properties. These statements can be proved directly by induction. From those, we can derive the following corollaries for a simplified version of typinf, taking only a term and a closed environment as arguments.

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

While principality is clearly more difficult, both proofs are somewhat comparable in length. As already pointed out by other authors, a painful aspect is the need to track fresh type variables. Here it is exacerbated by the complexity of the algorithm and of the data structures involved. We end up needing lots of lemmas about inclusion of free variable sets, such as the following two, for soundness and principality.

$$\begin{array}{l}
\mathsf{Lemma\ fv\_in\_compose} : \forall \theta\theta', \mathsf{FV}(\theta \circ \theta') \subset \mathsf{FV}(\theta') \setminus \mathsf{dom}(\theta) \cup \mathsf{FV}(\theta). \\
\mathsf{Lemma\ close\_fvk\_ok} : \forall \mathrm{K}\Gamma\alpha\kappa, \alpha \in \mathsf{FV}_{\mathrm{K}}(\Gamma) \rightarrow \alpha :: \kappa \in \mathrm{K} \rightarrow \mathsf{FV}(\kappa) \subset \mathsf{FV}_{\mathrm{K}}(\Gamma).
\end{array}$$

It is relatively difficult to pinpoint the presence of equi-recursive types as a direct cause of extra complexity, since the ability to infer equi-recursive types is just a side-effect of using a kinding environment. However, the need to compute a closure for free variables, which is a consequence of the kinding environment, is certainly a source of complexity. Another direct consequence of kinds is the need for proofs of admissibility of substitutions, but those are relatively short.

A more ironical problem is that I had to prove again the substitution lemma for types. The version proved for type soundness was easier but too specialized, and ended up not being sufficient here. The general version is just a little harder to prove.

As usual, the proof of principality requires the following lemma, which states that if a term e has a type τ under an environment Γ, then we can give it the same type under a more general environment Γ_1.

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

```
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 e′ →
    ∃h, ∃cl, eval h nil nil t nil = Result 0 cl ∧ e′ = clos2trm cl.
```

Fig. 11. Definitions and theorems for stack-based evaluation

$K \vdash \Gamma_1 \leq \Gamma$ means that the polytypes of Γ are instances of those in Γ_1. Due to the presence of kinds, the definition of the instantiation order gets a bit complicated.

$$K \vdash \bar{\kappa}_1 \triangleright \tau_1 \leq \bar{\kappa} \triangleright \tau \overset{\text{def}}{=} \forall \bar{\alpha}, \mathsf{dom}(K) \cap \bar{\alpha} = \emptyset \to \exists \bar{\tau},\ K, \bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}} \vdash \bar{\tau} :: \bar{\kappa}_1^{\bar{\tau}} \wedge \tau_1^{\bar{\tau}} = \tau^{\bar{\alpha}}.$$

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

$$K \vdash \forall \bar{\alpha}_1.K_1 \triangleright \tau_1 \leq \forall \bar{\alpha}_2.K_2 \triangleright \tau_2 \overset{\text{def}}{=} \exists \theta,\ \mathsf{dom}(\theta) \subset \bar{\alpha}_1\ \wedge K, K_1 \vdash \theta : K, K_2 \wedge \theta(\tau_1) = \tau_2.$$

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.

7 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. We defined a stack-based abstract machine, and first proved that at every step the state of the abstract machine

```
Parameter reduce_clos : Const.const → list clos → clos × list clos.

Variable fenv : env clos.

Definition trm2clos (benv : list clos) e :=
 match e with
 |  n  ⇒ nth n benv clos_def
 |  x  ⇒ match get x fenv with⟨v⟩ ⇒ v | ⟨⟩ ⇒ clos_def end
 |  c  ⇒ clos_const c nil
 | λe₁ ⇒ clos_abs e₁ benv
 |  _  ⇒ clos_def
 end.

Definition trm2app e :=
 match e with
 | (e₁ e₂) ⇒ ⟨e₁,e₂⟩
 | let e₂ in e₁ ⇒ ⟨λe₁,e₂⟩
 | _ ⇒ ⟨⟩
 end.

Fixpoint eval (h : nat) (benv : list clos) (app : list clos) (e : trm) (stack : list frame)
   {struct h} : eval_res :=
 match h with
 | 0 ⇒ Inter(Frame benv app e :: stack)
 | S h ⇒
   let result cl := match stack with
                    | nil ⇒ Result h cl
                    | Frame benv′ app′ e :: rem ⇒ eval h benv′ (cl :: app′) e rem
                    end in
   match trm2app e with
   | ⟨e₁,e₂⟩ ⇒ eval h benv nil e₂ (Frame benv app e₁ :: stack)
   | ⟨⟩  ⇒
   let cl := trm2clos benv e in
   match app with
   | nil ⇒ result cl
   | cl₁ :: rem ⇒
     match cl with
     | clos_abs e₁ benv ⇒ eval h (cl₁ :: benv) rem e₁ stack
     | clos_const c₁ app₁ ⇒
       let nargs := length app₁ + length app in
       let nred := S (Const.arity c₁) in
       if nred ≤ nargs then
         let (args,app′) := cut nred (app₁ ++ app) in
         match reduce_clos c₁ args with
         | (clos_const c₂ app₂,app₃) ⇒ eval h nil (app₂ ++ app₃ ++ app′) c₁ stack
         | (clos_abs e₂ benv,app₃)    ⇒ eval h benv (app₃ ++ app′) (λe₂) stack
         end
       else result (clos_const c₁ (app₁ ++ app))
     end
   end end
 end.
```

Fig. 12. Stack-based evaluation

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 (described above) and evaluation completeness are in Fig. 11. The concrete definition of eval is in Fig. 12. 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, trm_inst instantiating all de Bruijn indices at once with a list of terms, and const_app building the curried application of *c* to a list of terms. 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 indices, 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.

Completeness was proved 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:

$$\begin{array}{l}\mathsf{Definition\ inst}\ t\ benv := \mathsf{trm\_inst}\ t\ (\mathsf{map\ clos2trm}\ benv).\\ \mathsf{Lemma\ complete\_rec} : \forall args\, args'\, fl\, fl'\, e\, e'\, benv\, benv'\, \tau,\\ \quad args \equiv args' \to fl \equiv fl' \to (\mathsf{inst}\ e\ benv \longrightarrow \mathsf{inst}\ e'\ benv') \to\\ \quad \mathrm{K};\Gamma \vdash \mathsf{stack2trm}\ (\mathsf{app2trm}\ (\mathsf{inst}\ e\ benv)\ args)\ fl\ : \tau \to\\ \quad \exists h, \exists h', \mathsf{eval}\ h\ benv\ args\ e\ fl \equiv \mathsf{eval}\ h'\ benv'\ args'\ e'\ fl'.\end{array}$$

where $\equiv$ denotes the equality of closures after substitution by their environment, *i.e.* $\mathsf{clos\_abs}\ e\ benv \equiv \mathsf{clos\_abs}\ e'\ benv'$ iff $\mathsf{inst}\ (\lambda e)\ benv = \mathsf{inst}\ (\lambda e')\ benv'$. What it basically says is that if inst *e benv* reduces to inst e' $benv'$ in one step, and converting back the whole state of the stack machine with code *e*, variable environment *benv*, application stack *args*, and contral stack *fl*, gives a well-typed term, then the evaluation of this stack machine and that of the machine with respective state e', $benv'$, $args'$ and fl' (for any application stack $args'$ equivalent to *args* and control stack fl' equivalent to *fl*), will eventually either reach the same result, or an identical state if they do not terminate. This ensures that evaluation properly simulates reduction. Combined with eval_value —evaluation of a value produces a closure representing that value—, this allows to prove eval_complete.

$$\begin{array}{l}\mathsf{Lemma\ eval\_value} : \forall benv\ t,\ \mathsf{value}\ (\mathsf{inst}\ t\ benv) \to\\ \quad \exists h, \exists cl, \mathsf{eval}\ h\ benv\ nil\ t\ nil = \mathsf{Result}\ 0\ cl \wedge \mathsf{clos2trm}\ cl = \mathsf{inst}\ t\ benv.\end{array}$$

Both eval_value and eval_complete are easy, but proving complete_rec by case analysis on *e* and e' ended up being very time consuming. The proofs being rather repetitive, they may profit from better lemmas.

8 Dependent types

As we pointed in section 6, 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 [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 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 to streamline the proofs. This is due to the abstract nature of constraint domains, which limits interactions between kinds and other features. The definition of kinds becomes:

$$\begin{array}{l}\textsf{Definition coherent } kc\ kr := \forall x\,(\tau\,\tau' : \textsf{typ}),\\ \quad \textsf{Cstr.unique } kc\ x = \textsf{true} \to \textsf{In } (x,\tau)\ kr \to \textsf{In } (x,\tau')\ kr \to \tau = \tau'.\end{array}$$

$$\begin{array}{l}\textsf{Record ckind : Set := Kind\{}\\ \quad \textsf{kind\_cstr : Cstr.cstr;}\\ \quad \textsf{kind\_valid : Cstr.valid kind\_cstr;}\\ \quad \textsf{kind\_rel : list (Cstr.attr} \times \textsf{typ);}\\ \quad \textsf{kind\_coherent : coherent kind\_cstr kind\_rel\}.}\\ \textsf{Definition kind := option ckind.}\end{array}$$

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.

$$\begin{array}{l}\textsf{Definition ckind\_map\_spec} : \forall (f : \textsf{typ} \to \textsf{typ})(k : \textsf{ckind}),\\ \quad \{k' : \textsf{ckind} \mid \textsf{kind\_cstr } k = \textsf{kind\_cstr } k' \wedge \textsf{kind\_rel } k' = \textsf{map\_snd } f\ (\textsf{kind\_rel } k)\}.\end{array}$$

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[4].

$$\begin{array}{l}\textsf{Lemma ckind\_pi} : \forall k\,k' : \textsf{ckind},\\ \quad \textsf{kind\_cstr } k = \textsf{kind\_cstr } k' \to \textsf{kind\_rel } k = \textsf{kind\_rel } k' \to k = k'.\end{array}$$

Another application of dependent types is ensuring termination for the unification and type inference algorithms. In Coq all functions must be total. 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 6.1. 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

[4] Since both validity and coherence are decidable, proof irrelevance could be avoided here by slightly changing definitions.

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.

9 Program extraction

Both the type checker and interpreter can be extracted to Objective Caml code. This lets us build a fully certified[5] 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. Well-founded recursion cannot be used here, as our language is Turing-complete. (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 is going 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

[5] The validity of our certification relies on the correctness of Coq and Objective Caml, which are rather strong assumptions.

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
```

10 Related works

The mechanization of type safety proofs for programming languages has been extensively studied. Existing works include Core ML using Coq [5], Java using Isabelle/HOL [17], and more recently full specification of OCaml light using HOL-4 [19] and Standard ML using Twelf [13, 4]. The main difference in our system is the presence of structural polymorphism and recursion. In particular, among the above works, only [13] handles inclusion problems for iso-recursive types (in a simpler setting than ours, since when checking signature subtyping no structural polymorphism is allowed). 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 [23] 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 [20], Dubois and Ménissier-Morain in Coq [6], and Naraschewski and Nipkow in Isabelle [15], and the more recent Isabelle/Nominal proof by Urban and Nipkow [24]. 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 we are aware of no proof of principality including them. It might be interesting to compare these different proofs of W in more detail, as the first two use de Bruijn indices [6, 15], the latter nominal datatypes [24], and ours cofinite quantification. However, as Urban and Nipkow already observed, while there are clear differences between the different approaches, in the case of type inference lots of low-level handling of type variables has to be done, and as a result clever encodings do not seem to be that helpful.

More generally, all the literature concerning the PoplMark challenge [2] can be seen as relevant here, at least for the type soundness part. In particular, one could argue that structural polymorphism being related to structural subtyping, challenges 1B and 2B (transitivity of subtyping with records, and type safety with records and pattern matching) should be relevant. However, in the case of structural polymorphism, the presence of recursive types requires the use of a graph structure to represent types, which does not seem to be necessary for those challenges, where trees are sufficient. We believe that this changes the complexity of the proof.

File	Lines	Contents
Lib_∗	1706	Auxiliary lemmas and tactics from [1]
Metatheory	1376	Metatheory lemmas and tactics from [1]
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. Components of the proof

11 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 relaxed 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 now, we do not expect it to be an easy task.

References

1. B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 3–15, 2008.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Proc. of the International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Springer LNCS*, pages 50–65, 2005.
3. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.
4. K. Crary and B. Harper. Mechanized definition of Standard ML alpha release. Twelf proof scripts, Aug. 2009.
5. C. Dubois. Proving ML type soundness within Coq. In *Proc. of the International Conference on Theorem Proving in Higher Order Logics*, volume 1869 of *Springer LNCS*, pages 126–144, 2000.
6. C. Dubois and V. Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3):319–346, Nov. 1999.

7. J. P. Furuse and J. Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, Oct. 1995.
8. J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, Sept. 1998.
9. J. Garrigue. Simple type inference for structural polymorphism. In *The Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, 2002.
10. J. Garrigue. Relaxing the value restriction. In *Proc. International Symposium on Functional and Logic Programming*, volume 2998 of *Springer LNCS*, Nara, Apr. 2004.
11. J. Garrigue. A certified implementation of ML with structural polymorphism. In *Proc. Asian Symposium on Programming Languages and Systems*, volume 6461 of *Springer-Verlag LNCS*, pages 360–375, Shanghai, Nov. 2010.
12. J. Garrigue and D. Rémy. Extending ML with semi-explicit higher order polymorphism. *Information and Computation*, 155:134–171, Dec. 1999.
13. D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 173–184, Jan. 2007.
14. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.11, Documentation and user's manual*. Projet Gallium, INRIA, Nov. 2008.
15. W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.
16. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
17. D. v. Oheimb and T. Nipkow. Machine-checking the Java specification : Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Springer LNCS*, pages 119–156. Springer, 1999.
18. A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Prog. Lang. Syst.*, 17(6):844–895, Nov. 1995.
19. S. Owens. A sound semantics for OCaml light. In *Proc. European Symposium on Programming*, volume 4960 of *Springer LNCS*, pages 1–15, Apr. 2008.
20. L. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–169, 1985.
21. D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 77–87, 1989.
22. The Coq Team. *The Coq Proof Assistant, Version 8.2*. INRIA, 2009.
23. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, 2008.
24. C. Urban and T. Nipkow. Nominal verification of algorithm W. In G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*, pages 363–382. Cambridge University Press, 2009.