Proving ML Type Soundness Within Coq

Catherine Dubois

Université d'Évry Val d'Essonne, CNRS EP738, LaMI. F-91025 Évry Cedex, France
dubois@lami.univ-evry.fr

Abstract. We verify within the Coq proof assistant that ML typing is sound with respect to the dynamic semantics. We prove this property in the framework of a big step semantics and also in the framework of a reduction semantics. For that purpose, we use a syntax-directed version of the typing rules: we prove mechanically its equivalence with the initial type system provided by Damas and Milner. This work is complementary to the certification of the ML type inference algorithm done previously by the author and Valérie Ménissier-Morain.

1 Introduction

The piece of work presented in this paper supplements the certification laid out in [6] whose purpose was to verify in Coq the soundness and the completeness of the ML type inference algorithm with respect to the typing rules. We now connect the typing rules with the dynamic semantics and verify that the type system ensures a strong typing: a well-typed program cannot then produce type errors during its execution or, according to Milner's slogan [13], *Well-typed programs do not go wrong.* Thus the whole formal development presented in both this paper and [6] constitutes a machine-checked certification of the different aspects related to the ML typing discipline. More precisely we provide for a functional kernel of the ML language a formalization of the type system in the Calculus of Inductive Constructions and also a formalization of the type inference algorithm well-known in the literature as the algorithm $\mathcal{W}$. We prove within the Coq tool that $\mathcal{W}$ is correct and complete with respect to the typing rules. Completeness means here that if an expression is well-typed according to the typing rules then $\mathcal{W}$ succeeds and computes the principal type of the expression. The formal development contains also the definition of the dynamic semantics and establishes the soundness property. In this study, we consider a syntax-directed version of the typing rules. This version is often used in the ML community but it is not the one proposed initially by Damas and Milner [13]. Thus we formalize within Coq the initial version and prove mechanically the equivalence of both type systems. As far as we know, no publication does mention such a complete mechanized certification of ML typing aspects ([16, 8, 22] are only related to the type soundness, [15] is another certification of $\mathcal{W}$ in Isabelle/HOL).

The formulation and the proof of the type soundness property are intimately bound to the formulation of the dynamic semantics of the language. For example, for ML, Milner used a denotational semantics, Tofte used a big step semantic-s, Wright and Felleisen a reduction semantics. Machine-checked proofs of type soundness are often based upon a big step semantics or a reduction semantics. For instance, Syme [19] considers a reduction semantics to prove Java type soundness whereas Nipkow and von Oheimb [14] refer to a big step semantics. About ML, Terrasse in [21] uses a big step semantics but deals with the monomorphic case (Coq), Michaylov and Pfenning in [16] uses also a big step semantics but takes into account the polymorphic typing by substituting expressions (Elf). Lastly, in [3], A. Bove uses a reduction semantics but in the restricted monomorphic case (ALF). As far as we are aware, our machine-checked proof of ML type soundness is the first published one that deals with the notion of type scheme.

The rest of the paper is organized as follows. Section 2 presents the formalization of the ML kernel we consider together with its type system (and the involved notions e.g. substitutions). This part is another presentation (less detailed and less technical) of the sections 3, 4 and 5 of [6]. Consequently, the choices done for verifying $\mathcal{W}$, particularly the fact to stick to a functional implementation of $\mathcal{W}$, impact on the type soundness part. Then our paper deals with type soundness, first in the framework of an evaluation semantics (big step) and then in the context of a reduction semantics (small step). The last section connects our formalization with the type system provided by Damas and Milner.

We assume here familiarity with the Calculus of Inductive Constructions. We use version 6.1 of the Coq proof assistant [1]. In order to make this paper more readable, we adopt sometimes a pseudo-Coq syntax which differs slightly from the usual Coq syntax. Our paper provides the definitions of most concepts, the key lemmas but almost no proofs. The complete development is accessible on the Internet via `http://www.univ-evry.fr/labos/lami/specif/dubois`.

2 The type system

2.1 The kernel of the ML language

The expressions we consider are natural number constants, identifiers (x), λ-abstraction ($\lambda x.e$), application ($e\ e'$), `let` binding (`let` x `=` e `in` e') and recursive functions (`Rec` $f\ x.e$).

These expressions are described in Coq as an inductive data type (`expr`) with constructors for each kind of expressions. The type `ident` is the type of the identifiers. It does not matter what it is exactly provided that the equality of two identifiers is decidable.

```
Inductive expr: Set :=
  Const: nat -> expr         | Variable: ident -> expr
| Lam: ident -> expr -> expr | Rec: ident -> ident -> expr -> expr
| App: expr -> expr -> expr  | Let_in: ident -> expr -> expr -> expr
```

2.2 Types and type schemes

Types consist only of the basic type *nat*, type variables denoted as usual with Greek letters α, β ... and functional types $\tau \rightarrow \tau'$ (where τ and τ' are types too). It is encoded in Coq as:

```
Inductive type: Set :=
  Nat: type | Var: stamp -> type | Arrow: type -> type -> type
```

The type variables, whose type is `stamp`, are essentially natural numbers. It is also a choice close to implementations. In the following, Coq terms like `(Arrow t1 t2)` are sometimes written `t1` $\rightarrow$ `t2`.

In order to express parametric polymorphism, it is necessary to specify type schemes : a type scheme, of the form $\forall \alpha, \alpha_1, \dots \alpha_n.\tau$, is a type with some quantified type variables. The quantified variables are called the *generic variables* of the type scheme. A type scheme without generic variables is called a *trivial* type scheme and written as $\forall.\tau$.

In order to simplify the manipulation of free and bound variables, we distinguish syntactically free and bound variables in a type scheme. Consequently we define inductively the type `type_scheme` with two different constructors for variables, `Gen_var` for bound ones and `Var_ts` for the free ones.

```
Inductive type_scheme: Set :=  Nat_ts: type_scheme
| Gen_var: stamp -> type_scheme | Var_ts: stamp -> type_scheme
| Arrow_ts: type_scheme -> type_scheme -> type_scheme
```

According to this definition, the type scheme $\forall \alpha.\alpha \rightarrow \beta$ is represented by the Coq term `(Arrow_ts (Gen_var alpha) (Var_ts beta))` where `alpha` and `beta` are the stamps associated to α and β respectively. We may also write this Coq term as `(Gen_var alpha)` $\rightarrow^{\sigma}$`(Var_ts beta)` in a pseudo-Coq syntax.

The choice of this representation for type schemes has an important impact on the formal development. It allows to define with case analysis many operations on type schemes and to proceed by induction in a lot of proofs. However this choice gives no help when α-conversion is concerned. In order to smooth away this difficulty, it would be interesting to use higher order abstract syntax [5] to represent variable bindings in type schemes. But this representation does not admit induction (for the moment): this is a damning drawback for us.

2.3 Type environments

The type information about the free identifiers of an expression is contained in a type environment (environment for short when there is no ambiguity) denoted in the rest of the paper by Γ or `env`. Because of polymorphism, environments contain type schemes. Thus an environment can be considered as a partial function from identifiers to type schemes. In Coq we represent an environment as a list of associations between identifiers and type schemes. Thus the type of environments, `type_env`, is defined as `list (ident * type_scheme)`.

The operation `assoc_ident_in_env` that finds the type scheme associated to an identifier (it is also written informally as $\Gamma(x)$) may fail. Thus this operation has the type `ident -> type_env -> (option type_scheme)` where the type `option` defined below allows to simulate the exception mechanism.

```
Inductive option [A : Set] : Set :=
      None : (option A) | Some : A -> (option A)
```

The extension of an environment is done by the operation `add_env` implemented as a simple list addition (a classical *cons*). We use also the informal notation $\Gamma \oplus x : \sigma$ (where x is an identifier and σ a type scheme).

2.4 Substitutions and instances

The literature provides different definitions for the notion of substitution, which are not all equivalent (see [11] for a survey). We consider a substitution to be a function s from the set of type variables to the set of types, such that the domain, that is $\{x : \texttt{stamp} \mid s(x) \neq x\}$, is finite : then s behaves like the identity anywhere else.

Substitutions are undeniably fundamental objects in our mechanized verification, but in fact they are brought indirectly by two instance relations: the instance relation between two types (type instance) and the instance relation between a type and a type scheme (generic instance).

> The type τ is a type instance of the type τ' if there exists a substitution s such that $s\tau' = \tau$.
>
> The type τ is a generic instance of the type scheme $\forall\alpha_1, \dots, \alpha_n.\tau'$ if there exists a substitution s whose domain is $\{\alpha_1, \dots, \alpha_n\}$ such that $s\tau' = \tau$.

Consequently we distinguish two kinds of substitutions:

- the so-called *free substitutions* (or substitutions), that can work only on the free variables of a type, a type scheme or an environment.
- the so-called *generic substitutions* that can work only on the generic variables of a type scheme.

The first ones are represented in Coq as association (between type variable and type) lists. The generic substitutions are represented by type vectors, without any reference to the names of the variables they are concerned with. They are used with the requirement that the type of the ith generic variable is located at the ith position in the vector.

Many operations come with the definition of substitutions: application of a substitution on a type variable, a type, a type scheme, composition of substitutions, domain, range, free variables of a substitution ... These operations are specified in Coq in a functional style and are very close to their ML implementation.

The choice of representing substitutions as association lists make some operations (e.g. the composition) complex. The proof that the composition does really what it is expected is quite clumsy (about 600 lines).

We could also represent substitutions (both kinds) by Coq abstractions of type `stamp -> type`. The application of a substitution to a variable, the composition of two substitutions are then operations got for free. This kind of representation is very attractive and often chosen in proof assistants based on λ-calculus. It is essentially the representation chosen by Naraschewski and Nipkow [15]. However the functional representation makes the implementation of some operations (e.g. the computation of the domain) impossible. And we need such an operation !

The notion of generic instances induces an ordering between type schemes: a type scheme σ_1 is said to be *more general* than a type scheme σ_2 and written $\sigma_1 \succ \sigma_2$ or in Coq (`more_general` σ_1 σ_2), if and only if any arbitrary generic instance of σ_2 is also a generic instance of σ_1. For example, $\forall\alpha\beta.\alpha \rightarrow \beta$ is more general than $\forall\alpha.\alpha \rightarrow \alpha$.
The translation in Coq is straightforward:

```
Definition more_general: type_scheme -> type_scheme -> Prop :=
[ts1, ts2 : type_scheme]
 (∀ t: type, (is_gen_instance t ts2) -> (is_gen_instance t ts1))
The Coq notation [x: T]e binds the identifier x of type T in e
```

This ordering induces in turn a partial order between environments: Γ_1 is said to be more general than the environment Γ_2 ($\Gamma_1 \succ \Gamma_2$) if and only if Γ_1 and Γ_2 are relative to the same identifiers[1] $x_1, x_2 \ldots x_n$ and $\forall i \in [1,n],\ \Gamma_1(x_i) \succ \Gamma_2(x_i)$.

2.5 Type generalization

The `let` construct is the only one that may introduce true polymorphic types[2] in the environment. This is done by the operation of generalization `gen_type` which builds a type scheme from a type τ and an environment Γ: it turns into generic variables those variables appearing free in τ but not in Γ.

$$\texttt{gen\_type}\ \tau\ \Gamma = \forall\alpha_1 \ldots \alpha_n.\tau$$

with $\alpha_i \in (\texttt{FV\_type}\ \tau) - (\texttt{FV\_env}\ \Gamma)$ (as indicated by its name, `FV_env` computes the list of free variables of an environment).

The most natural Coq implementation that follows from the representations of types and type schemes is a function (see below) defined by case analysis according to the type to be generalized.

1 we impose without any loss of generality the same order for the identifiers in both environments

2 that is, non trivial type schemes

```
 Fixpoint  gen_type:= [t: type] [env: type_env]
   Cases t of
   Nat -> Nat_ts
 | (Var v) ->  if v ∈ (FV_env env) then (Var_ts v) else (Gen_var v)
 | (Arrow t1 t2) -> (Arrow_ts (gen_type t1 env) (gen_type t2 env))
   end.
```

However we do not implement the generalization exactly in this way. The implemented algorithm shares the same structure but incorporates a linear encoding of generic variables: any occurrence of the generic variable α is encoded as `(Gen n)` if α is the `n`th generic variable discovered during the generalization. Thus the generalization of the type $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha)$ with respect to the empty environment is the type scheme `((Gen_var 0)` $\rightarrow^{\sigma}$`(Gen_var 1))` $\rightarrow^{\sigma}$`((Gen_var 1)` $\rightarrow^{\sigma}$`(Gen_var 0))`.

This encoding provides us with the following property: two type schemes obtained by generalization, identical up to the renaming of the generic variables, are represented by two terms (of type `type_scheme`) *syntactically equal.* Nevertheless the price for this is high: because of the encoding, the lemmas involving generalization have an inductive step not very natural, close to an invariant (see [6] for more details).

2.6 The typing rules and some of their properties

The typing rules are given in the Natural Semantics style [10] (see figure 1). They are described as inference rules expressing how to derive typing sequents of the form $\Gamma \vdash e : \tau$. Such a sequent is read *the expression e has type τ under the environment* Γ.

The typing rules are encoded in Coq as clauses of the inductive relation `type_of`, the translation is quite obvious here. Here is a fragment of the Coq specification:

```
 Inductive type_of: type_env -> expr -> type -> Prop :=
  type_of_const: ∀ env: type_env, ∀ n: nat, (type_of env (Const n) Nat)
| type_of_var: ∀ env: type_env, ∀ x: ident,
  ∀ t: type, ∀ ts: type_scheme,
  (assoc_ident_in_env x env)=(Some ts)  ->
    (is_gen_instance t ts) -> (type_of env (Variable x) t)
| type_of_lam: ∀ env: type_env, ∀ x: ident, ∀ e: expr, ∀ t, t': type,
  (type_of (add_env env x (type_to_type_scheme t)) e t') ->
    (type_of env (Lam x e) (Arrow t t'))
...
```

An important property, that appears as a key property for many other properties, states that the relation `type_of` is stable under substitution.

$$(\textsc{cst})\ \Gamma \vdash n : nat$$

$$(\textsc{id})\ \frac{\Gamma(x) = \sigma, \quad \tau \text{ is a generic instance of } \sigma}{\Gamma \vdash x : \tau}$$

$$(\textsc{abs})\frac{\Gamma \oplus x : \forall.\tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x.e : \tau\ \ \rightarrow \tau'}$$

$$(\textsc{rec})\frac{\Gamma \oplus x : \forall.\tau \oplus f : \forall.\tau \rightarrow \tau' \vdash e : \tau'}{\Gamma \vdash \texttt{Rec}\ f\ x.e : \tau\ \ \rightarrow \tau'}$$

$$(\textsc{app})\frac{\Gamma \vdash e : \tau \rightarrow \tau', \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

$$(\textsc{let})\frac{\Gamma \vdash e : \tau, \quad \Gamma \oplus x : (\texttt{gen\_type}\ \tau\ \Gamma) \vdash e' : \tau'}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \tau'}$$

Fig. 1. The typing rules

```
Theorem typing_is_stable_under_substitution:
∀ e: expr, ∀ t: type, ∀ env: type_env, ∀ s: substitution,
(type_of env e t) ->
  (type_of (apply_subst_env env s) e (apply_subst_type s t))
```

Our Coq verification of this theorem deals explicitly with α-conversion. It requires for example to formalize the notion of renaming substitution and to verify mechanically under what conditions a type and a renamed version of it are generalized in the same type scheme. In that sense, our proof is fundamental because we really formalize and verify informal proofs that often become a little bit nebulous as soon as they deal with renaming. For details, see [6].

Some other properties about the typing rules are also required in different points in our whole development, for example the following property connecting typing sequents together with the ordering $\succ$ between environments:

```
Theorem typing_in_a_more_general_env:
∀ e: expr, ∀ Γ1, Γ2: type_env, ∀ τ: type,
Γ1 ≻ Γ2 -> (type_of Γ2 e τ) -> (type_of Γ1 e τ)
```

3 Big step dynamic semantics

The big step dynamic semantics gives a meaning to the expressions of the language by defining their evaluation. Again we chose to specify it in the style of Natural Semantics. Let us first describe the possible values and then the inference rules.

3.1 Semantic values and evaluation environments

The values we consider here are numbers and functional values also called closures. The value of an expression depends on the values of its free variables. These values are recorded in an evaluation environment (environment for short when there is no ambiguity), written Δ or `c` in the following.

Values and environments are mutually recursive notions. In effect, a closure is a pair composed of a functional expression and an environment. Two kinds of closures are distinguished: recursive closures $<<_{@}$ `Rec` $f\ x.e, \Delta >>$ and non recursive closures $<< \lambda\ x.e, \Delta >>$. This discrimination is also introduced by Boutin in his ML compiler certification [2]. We could also consider opaque closures (closures whose contents cannot be inspected): these are the values associated to predefined operations. They can be ignored without any loss of generality.

Within Coq, values and environments are specified by two mutually inductive types `val` and `eval_env` (isomorphic to lists of pairs *(identifier, value)*). [3]

```
Mutual Inductive val: Set := Num: nat -> val
   | Clos: ident -> expr -> eval_env -> val
   | Rec_clos: ident -> ident -> expr -> eval_env -> val
with  eval_env : Set := Cnil: eval_env
                      | Ccons: ident*val -> eval_env -> eval_env
```

Structurally the evaluation environments are very similar to the typing environments. Consequently we use similar notations: $\Delta(x)$ (in Coq, `assoc_ident_in-_eval`), $\Delta \oplus y : v$.

3.2 The dynamic semantics

The inference rules that describe the dynamic semantics are detailed in the figure 2. The evaluation sequent $\Delta \vdash_{eval} e \hookrightarrow v$ is read *the expression e evaluates to the value v in the environment* Δ.

The inference rules implements a call by value semantics. The distinction between the recursive and non recursive closures implies the definition of two inference rules for the application: (APP1) when a non recursive function is applied and (APP2) when a recursive function is applied.

The inference rules in figure 2 are translated into the inductive predicate `val_of` (a constructor per inference rule):

[3] We use here a separate type for environments and not the predefined lists : this choice is due to a limitation of the Coq version V6.1 we used.

```
Inductive val_of: eval_env -> expr -> val -> Prop:=
 Val_of_num: ∀ n: nat,∀ c: eval_env, (val_of c (Const n) (Num n))
|Val_of_ident: ∀ c: eval_env, ∀ i: ident, ∀ v: val,
  (assoc_ident_in_eval i c) = (Some v) -> (val_of c (Variable i) v)
|Val_of_lambda: ∀ c: eval_env, ∀ i: ident, ∀ e: expr,
  (val_of c (Lam i e) (Clos i e c))
|Val_of_app1: ∀ c,c1: eval_env, ∀ e1,e2,e: expr, ∀ i: ident,
    ∀ u,v: val, (val_of c e1 (Clos i e c1)) -> (val_of c e2 u) ->
      (val_of (Ccons (i,u) c1) e v)-> (val_of c (App e1 e2) v)
....
```

$$(\textsc{cst})\quad \Delta \vdash_{eval} n \hookrightarrow n$$

$$(\textsc{id})\quad \Delta \vdash_{eval} x \hookrightarrow \Delta(x)$$

$$(\textsc{abs})\quad \Delta \vdash_{eval} \lambda\ x.e \hookrightarrow << \lambda\ x.e, \Delta >>$$

$$(\textsc{rec})\quad \Delta \vdash_{eval} \mathtt{Rec}\ f\ x.e \hookrightarrow <<_{@} \mathtt{Rec}\ f\ x.e, \Delta >>$$

$$(\textsc{app1})\quad \frac{\begin{array}{c}\Delta \vdash_{eval} e \hookrightarrow << \lambda\ x.e_f, \Delta_f >>\ , \quad \Delta \vdash_{eval} e' \hookrightarrow v, \\ \Delta_f \oplus x : v \vdash_{eval} e_f \hookrightarrow v'\end{array}}{\Delta \vdash_{eval} e\ e' \hookrightarrow v'}$$

$$(\textsc{app2})\quad \frac{\begin{array}{c}\Delta \vdash_{eval} e \hookrightarrow <<_{@} \mathtt{Rec}\ f\ x.e_f, \Delta_f >>\ , \quad \Delta \vdash_{eval} e' \hookrightarrow v, \\ \Delta_f \oplus x : v \oplus f : <<_{@} \mathtt{Rec}\ f\ x.e_f, \Delta_f >> \vdash_{eval} e_f \hookrightarrow v'\end{array}}{\Delta \vdash_{eval} e\ e' \hookrightarrow v'}$$

$$(\textsc{let})\quad \frac{\Delta \vdash_{eval} e \hookrightarrow v, \quad \Delta \oplus x : v \vdash_{eval} e' \hookrightarrow v'}{\Delta \vdash_{eval} \mathtt{let}\ x\ \mathtt{=}\ e\ \mathtt{in}\ e' \hookrightarrow v'}$$

Fig. 2. Big step dynamic semantics

3.3 Typing soundness or the *subject reduction* theorem

The proved property is that any well-typed expression of type τ whose evaluation terminates has a value of type τ. This formulation shows that we need to formalize the notion of type for a value. It is immediate for a natural number constant but not for a closure because it is not an object of the language. Consequently we specify an inductive predicate called "semantic typing" (`type_of_val` in Coq) that links a value to its type: we write $\Vdash v : \tau$ to indicate that *the value* v *has the type* τ.

Furthermore the typing/evaluation connection can only be done if the typing environment and the evaluation environment agree (we write $\Gamma \vdash \Delta$ to denote that property, the corresponding Coq predicate is `eval_type_env_match`). It means the value associated to an identifier in Δ has the type (more precisely the type scheme) indicated in the typing environment Γ.

A value v is assigned the type scheme σ ((`sem_gen` $v\ \sigma$) in Coq) if v has some type obtained as a generic instance of σ.

To define the semantic typing predicate, we follow Tofte's approach reformulated by Leroy in [12]. Thus we use the typing rules to type a closure. Informally, it means that the value $<< \lambda\ x.e, \Delta >>$ has the type $\tau_1 \rightarrow \tau_2$ if there exists a typing environment Γ that agrees with the evaluation environment Δ ($\Gamma \vdash \Delta$) and such that the typing sequent $\Gamma \vdash \lambda\ x.e : \tau_1 \rightarrow \tau_2$ can be derived.

The Coq formalization (given below) of the previous predicates raises no particular problem. Their definitions are mutually inductive. Let us notice that our Coq definition for $\Gamma \vdash \Delta$ adds a constraint about the order of the identifiers in both Δ and Γ : it must be the same. This constraint simplifies the formulation and the proof but is not restrictive at all.

```
Mutual Inductive type_of_val: val -> type -> Prop :=
 type_num: ∀ n: nat, (type_of_val (Num n) Nat)
|type_closure: ∀ i: ident, ∀ e: expr, ∀ c: eval_env,
   ∀ env: type_env, ∀ t1, t2 : type,
   (eval_type_env_match c env) ->
    (type_of env (Lam i e) (Arrow t1 t2)) ->
     (type_of_val (Clos i e c) (Arrow t1 t2))
|type_rec_closure:  similar to the previous clause

with  eval_type_env_match: eval_env -> type_env -> Prop :=
      match_nil: (eval_type_env_match Cnil nil)
     |match_cons: ∀ c: eval_env, ∀ env: type_env, ∀ i: ident,
      ∀ ts: type_scheme, ∀ v: val,
      (sem_gen v ts) -> (eval_type_env_match c env) ->
         (eval_type_env_match (Ccons (i,v) c)  (cons (i,ts) env))

with sem_gen: val -> type_scheme -> Prop  :=
     sem_gen_def: ∀ v: val, ∀ ts: type_scheme,
        (∀ t: type,  (is_gen_instance t ts) -> (type_of_val v t)) ->
            (sem_gen v ts)
```

The formulation of the typing soundness theorem in pseudo-Coq is as follows:

```
Theorem subject reduction:
∀ $e$: expr, ∀ $v$: val, ∀ $\Delta$: eval_env, ∀ $\Gamma$ : type_env, ∀ $\tau$ : type,
   $\Delta\ \vdash_{eval}\ e\ \hookrightarrow\ v$  ->  $\Gamma\ \vdash\ e\ :\ \tau$  ->
         $\Gamma\ \vdash\ \Delta$  ->  $\Vdash v\ :\ \tau$
```

The proof proceeds by induction on $\Delta \vdash_{eval} e \hookrightarrow v$. Not surprisingly, the `let` step is the most difficult one. It requires the following lemma: if a value u has the type τ, then it also has the type scheme resulting from the generalization of τ.

```
Lemma sem_gen_gen_type: ∀ u: val, ∀ τ: type, ∀ Γ: type_env,
(type_of_val u τ)  ->  (sem_gen u (gen_type τ Γ)).,
```

Proving this lemma consists in establishing that for any generic instance τ' of `(gen_type` τ Γ`)`, the value u has the type τ'. The generic instance τ' can be written as a type instance of τ (according to a lemma required in the certification of $\mathcal{W}$). The end of the proof rests upon the property `type_val_stable_subst` close to the stability of typing by substitutions: if u has the type τ then u has also the type $s\tau$ for any substitution s.

```
Lemma type_val_stable_subst: ∀ v: val, ∀ τ: type, ∀ s: substitution,
(type_of_val v τ)  ->  (type_of_val v (extend_subst_type s τ))
```

To verify this last property, we need to use a mutual induction scheme (between evaluation environments and values) generated automatically by Coq. In fact we prove simultaneously a similar property about the typing/evaluation environments connection: if $\Gamma \vdash \Delta$ then $s\Gamma \vdash \Delta$ for any substitution s.
Here again we use the property of preservation of the typing sequents by substitution.

The formalization of the big step semantics together with the proof of the type soundness require about thirty supplementary definitions and a hundred new lemmas with respect to the certification of $\mathcal{W}$. It is very little compared with the 7500 lines (91 definitions and 322 lemmas) for verifying $\mathcal{W}$.

4 Reduction dynamic semantics

The dynamic semantics presented in the previous section is called big step semantics because it gives no information about the computation. It only considers the possible values resulting from the evaluation of an expression. It follows that the big step semantics cannot deal with non terminating programs. The reduction semantics, also called small step semantics, specifies the elementary steps of the computation and consists of a bunch of rewriting rules. Consequently we can observe the reduction of an expression step by step (through a derivation) either for ever (if it is a non terminating expression) or until an expression in normal form is obtained (if the initial expression terminates). In our case, an expression in normal form also called a value is a constant or an abstraction (recursive or not).

Using a reduction semantics to establish type soundness for languages *à la ML* has been popularized by Wright and Felleisen in [23] and again afterwards by other researchers, for example Rémy and Vouillon when they specified the semantics of Objective ML [18]. With such an approach a type error is modelled as a locked reduction, that is the impossibility to further reduce a non value expression. In this context, establishing type soundness consists in verifying two properties: the preservation of the type by reduction (also called the subject reduction theorem) and the non-locking of well typed programs.

In this section we present the Coq formalization of the reduction semantics and prove the preservation of the type by reduction. We have also proved the non-locking property by establishing that any well-typed program that cannot reduce anymore is a value. This last part, not developed here by lack of space, does not raise any specific difficulty.

The Coq theories relative to this section add a dozen definitions and about thirty five lemmas.

4.1 The Coq specification of the reduction semantics

The formalization of the reduction semantics is modular, it consists of three steps:

– the definition of the subset of the expressions that are values,

– the definition of the evaluation contexts that indicate where the reductions are allowed. A context is an expression that contains a hole written •. The notation $C[e]$ denotes the expression obtained by placing an expression e in the hole of C.

  The contexts are described by the following grammar

$$C ::= \bullet \mid C\ e \mid v\ C \mid \texttt{let}\ x\ \texttt{=}\ C\ \texttt{in}\ \ e$$

  where v and e are respectively a value and an expression.

  Defining such contexts amounts to imposing a reduction strategy. For instance, the right hand side of an application can be reduced only if the left hand side is a value.

– the definition of the reduction relation $\longrightarrow_r$ that specifies the elementary reductions.

These different steps are modelled within Coq as follows:

values The subset of the expressions which are values is described as the set of expressions such that the predicate `is_value` is proved to be satisfied:

```
Inductive is_value: expr -> Prop :=
  Cst_val : ∀ n: nat,  (is_value (Const n))
 |fun_val : ∀ i: ident, ∀ e: expr, (is_value (Lam i e))
```

contexts One originality of our work is that we explicitly formalize the notion of evaluation context. In fact, if many Coq contributions about λ-calculus use a reduction relation, few of them formalize the notion of context.

We have chosen to represent a context (of type `context`) as a function on expressions. Then `context` is synonymous with `expr -> expr`. Consequently $C[e]$ is translated to the application (`c` e) where `c` is the functional representation of C. The following inductive definition `is_MLcontext` describes the allowed contexts.

```
Inductive is_MLcontext: context -> Prop :=
  hole : (is_MLcontext ([x:expr]x))
 |app_left : ∀ e: expr, c: context,
               (is_MLcontext c) ->
                (is_MLcontext ([x: expr] (App c x e)))
 |app_right : ∀ v: expr, (is_value v) ->
                 ∀ c: context, (is_MLcontext c) ->
                   (is_MLcontext ([x: expr] (App v c x )))
 |let_left : ∀ e: expr, ∀ i: identifier,
               ∀ c: context, (is_MLcontext c) ->
                  (is_MLcontext ([x: expr] (Let_in i c x e)))
```

the reduction relation $\longrightarrow_r$ It contains 3 rules that express β-reduction and a fourth one which specifies the relation $\longrightarrow_r$ is context compatible.

(β_1) $(\lambda x.e)v \longrightarrow_r e[v/x]$

(β_2) $(\texttt{Rec}\ fx.e)v \longrightarrow_r e[v/x, \texttt{Rec}\ fx.e/f]$

(β_3) $\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \longrightarrow_r e[v/x]$

$$(\text{CONTEXT})\frac{e_1 \longrightarrow_r e_2}{C[e_1] \longrightarrow_r C[e_2]}$$

The formalization of the relation $\longrightarrow_r$ requires the preliminary specification of the following notions:

– free/bound identifier (inductive predicate `free_ident`)

– substituting an expression e' for x in an expression e: this operation is not allowed when free identifiers may be captured (we do not implement automatic renaming). The easiest way to implement a partial operation in Coq [4] is to switch to a relational version defined inductively, `subst_expr`. Thus (`subst_expr` $e\ x\ e'\ e''$) means e'' is the expression obtained by replacing in the expression e all the free occurrences of the identifier x by the expression e' ($e'' = e[e'/x]$).

[4] The Coq functions are incurably total functions (see [7] and [17] about encoding partial functions).

The relation $\longrightarrow_r$ is written in Coq as an inductive predicate `red` with four constructors corresponding to the rules (β_1), (β_2), (β_3) and (CONTEXT).

```
Inductive red: expr -> expr -> Prop :=
  beta1: ∀ i: ident, ∀ e1,v,er: expr,
     (is_value v) -> (subst_expr e i v er) ->
       (red  (App (Lam i e) v) er)
| beta2, beta3  follow a similar construction
| context: ∀ e1,e2: expr, ∀ ctx: context,
     (red e1 e2) -> (is_MLcontext ctx) -> (red (ctx e1) (ctx e2))
```

4.2 Subject reduction theorem

The subject reduction property states that reductions preserve the type of expressions. It is given below in the case of an elementary reduction step (it can be easily extended to the closure of $\longrightarrow_r$).

```
Theorem reduction_preserves_types: ∀ e₁,e₂: expr,
    (e₁  ⟶ᵣ  e₂) ->
      (∀ τ: type, ∀ Γ: type_env, Γ ⊢  e₁ : τ  -> Γ ⊢  e₂ : τ)
```

The proof proceeds by case analysis according to the reduction $e_1 \longrightarrow_r e_2$. In the step corresponding to the context compatibility, we finish the proof by case analysis on the form of the context.
For reductions involving β-reduction a substitution lemma is the key to showing type preservation.

```
Lemma substitution:
∀ e,e₁,e₂: expr, ∀ τ,τ₁: type, ∀ Γ: type_env, ∀ i: ident,
   Γ ⊢  e : τ  ->
     Γ ⊕ i : (gen_type τ Γ) ⊢  e₁ : τ₁  ->
       (subst_term e₁ i e e₂)  ->  Γ ⊢  e₂ : τ₁
```

First of all, let us notice the formulation of the `substitution` lemma. Usually in the literature the typing hypothesis about e_1 assigns to the identifier i the type scheme $\forall\alpha, \alpha_1, \dots \alpha_n.\tau$ where $\alpha, \alpha_1, \dots \alpha_n$ are type variables not free in Γ. Thus the lemma we prove in our formalization can be seen as a specialization of the usual one. However it suffices to establish the type soundness and furthermore a lot of technical lemmas about `gen_type` were already available.

To verify this lemma, we proceed by induction on the expression e. The heaviest induction step concerns the abstraction because renaming of type variables is required. However numerous required properties have been established for verifying the property of preservation of typing sequents by substitution.

More generally, the proof makes an intensive use of the relation $\succ$ and the connected properties as for example the lemma `typing_in_a_more_general_env` (displayed in section 2.6). It remains to establish several supplementary lemmas about the type system as for example the extension lemma (see below). This lemma states that adding (or removing) in the environment a type information about an identifier non free in the expression has no impact on the conclusion of a typing sequent.

```
Lemma env_extension :
∀ e: expr, ∀ x: ident, ∀ τ: type, σ: type_scheme, ∀ Γ: type_env,
  ¬(is_free x e)  ->  ((Γ ⊕ x : σ) ⊢  e  :  τ  <->  Γ ⊢  e  :  τ)
```

The proof of this lemma is based on a very specific equivalence relation between environments, $\Gamma \approx \Gamma'$, used nowhere else. Mainly this equivalence is built on the two following clauses:

$\Gamma \approx \Gamma'$ -> $(i:\sigma)(i:\sigma')\Gamma \approx (i:\sigma)\Gamma'$

$\Gamma \approx \Gamma'$ -> $(i:\sigma)(j:\sigma')\Gamma \approx (j:\sigma')(i:\sigma)\Gamma'$ when $i \neq i'$

The first clause removes a useless information, the second one swaps the information for two distinct identifiers. An important lemma about that notion states that if the typing sequent $\Gamma \vdash e : \tau$ is valid then we can also derive the sequent $\Gamma' \vdash e : \tau$ when $\Gamma \approx \Gamma'$.

5 Equivalence of type systems

All the work presented until now uses a syntax-directed presentation (called DM' in the paper) of the type system. Although this version is commonly used, it is not the initial version (called DM in the paper) given by Damas and Milner. This last one is not syntax-directed. We have chosen to use the presentation DM' because it is closer to $\mathcal{W}$ than DM and it is deterministic. This feature makes many proofs easier.

The systems DM and DM' are equivalent. We can find a paper proof in [9] but according to us, no formal and mechanized proof exists. We prove this equivalence with Coq, more precisely the soundness and completeness of DM' with respect to DM. It requires seventeen supplementary definitions and seventy lemmas.

5.1 Formalization of the Damas-Milner type system DM

The DM typing rules are given in the figure 3: the typing sequent is now $\Gamma \vdash_{DM} e : \sigma$ (a type scheme is used instead of a type). We follow here the presentation given in [4] by Clément et al. We use also our favorite notations (e.g. a trivial type scheme (a type) is written $\forall.\tau$).

The rules (CST), (ABS), (REC), (APP) are identical in both systems: in DM, they handle trivial type schemes, likened to types. The rule (TAUT) related to identifiers is very simple : it only extracts the type information from the environment. The (LET) rule does no generalization at all, but on the other hand the rule (GEN) may introduce some supplementary quantified variables in a type scheme. The rule (INST) may weaken the type scheme of an expression.

$$(\textsc{cst})\quad \Gamma \vdash_{DM} n : \forall.nat$$

$$(\textsc{taut})\quad \Gamma \vdash_{DM} x : \Gamma(x)$$

$$(\textsc{inst})\quad \frac{\Gamma \vdash_{DM} x : \sigma, \quad \sigma \succ \sigma'}{\Gamma \vdash_{DM} x : \sigma'}$$

$$(\textsc{gen})\quad \frac{\Gamma \vdash_{DM} x : \sigma, \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash_{DM} x : \forall\alpha.\sigma}$$

$$(\textsc{abs})\quad \frac{\Gamma \oplus x : \forall.\tau \vdash_{DM} e : \forall.\tau'}{\Gamma \vdash_{DM} \lambda\ x.e : \forall.\tau \ \to \tau'}$$

$$(\textsc{rec})\quad \frac{\Gamma \oplus x : \forall.\tau \oplus f : \forall.\tau \to \tau' \vdash_{DM} e : \forall.\tau'}{\Gamma \vdash_{DM} \texttt{Rec}\ f\ x.e : \forall.\tau \ \to \tau'}$$

$$(\textsc{app})\quad \frac{\Gamma \vdash_{DM} e : \forall.\tau \to \tau', \quad \Gamma \vdash_{DM} e' : \forall.\tau}{\Gamma \vdash_{DM} e\ e' : \forall.\tau'}$$

$$(\textsc{let})\quad \frac{\Gamma \vdash_{DM} e : \sigma, \quad \Gamma \oplus x : \sigma \vdash_{DM} e' : \sigma'}{\Gamma \vdash_{DM} \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \sigma'}$$

Fig. 3. The *DM* type system

The inductive predicate `type_of_DM`, illustrated partly below, formalizes the *DM* system in Coq. The relationship with the inference rules is again straightforward, except for the constructor `type_of_DM_gen` that quantifies a list of variables instead of a unique variable in the rule (GEN). The function `bind_list` implements the binding of variables in a type scheme. Its definition is very close to the definition of the function `gen_type`: it introduces a similar linear encoding of the generic variables.

```
Inductive type_of_DM: type_env -> expr -> type_scheme -> Prop :=
  type_DM_taut: ∀ env: type_env, ∀ x: ident, ∀ ts : type_scheme,
     (assoc_ident_in_env x env)=(Some ts) ->
       (type_of_DM env (Variable x) ts)
| type_DM_inst: ∀ env: type_env, ∀ e: expr, ∀ ts,ts’: type_scheme,
     (type_of_DM env e ts) -> ts ≻ ts’ -> (type_of_DM env e ts’)
| type_DM_gen: ∀ env: type_env, ∀ e: expr, ∀ ts: type_scheme,
     (type_of_DM env e ts) ->
        ∀ l : (list stamp)(are_disjoints l (FV_env env)) ->
           (type_of_DM env e (bind_list l ts))
 ...
| type_DM_let_in: ∀ env: type_env, ∀ e,e': expr,
    ∀ x: ident, ∀ ts,ts': type_scheme
     (type_of_DM env e ts) ->
       (type_of_DM (add_env env x ts) e' ts') ->
         (type_of_DM env (Let_in x e e') ts')
```

5.2 Soundness of DM' with respect to DM

We demonstrate the soundness of DM' with respect to DM by showing that if we can prove with the DM' typing rules that e has type τ then we can prove with the DM typing rules that e has also the type τ (or the trivial type scheme $\forall.\tau$). All that is done with the same environment.

Lemma soundness_*DM'*_wrt_*DM*: ∀ e: expr, ∀ Γ: type_env, ∀τ: type,
$\Gamma \vdash e : \tau$ -> $\Gamma \vdash_{DM} e : \forall.\tau$

The proof is done by induction on the expression e. Most cases are established by applying the corresponding rule in *DM* and the induction hypothesis. The case where e is an identifier requires to successively apply the rules (TAUT) and (INST). The `let` case needs more effort, in particular we have to rewrite (bind_list (gen_vars τ Γ) $\forall.\tau$) as (gen_type τ Γ). Intuitively, this rewriting means that binding (in one time) all the variables of τ not free in Γ (computed by (gen_vars τ Γ)) gives exactly the same result as generalizing τ with respect to Γ. We have here a syntactic equality because of the encoding encapsulated in both bind_list and gen_type.

5.3 Completeness of DM' with respect to DM

The completeness theorem states that if we can prove with the DM typing rules that e has the type scheme σ then we can establish with the DM' typing rules that e has a type τ whose generalization provides a type scheme at least as general as σ.

Lemma completeness_*DM'*_wrt_*DM*:
∀ e: expr, ∀ Γ: type_env, ∀ σ: type_scheme,
$\Gamma \vdash_{DM} e : \sigma$ -> ∃ τ. $\Gamma \vdash e : \tau$ ∧ (gen_type τ Γ) $\succ$ σ

The proof requires an induction on $\Gamma \vdash_{DM} e : \sigma$ and is based on numerous lemmas about the $\succ$ relation. For example:

$\sigma_1 \succ \sigma_2$ -> $s\sigma_1 \succ s\sigma_2$ where s is a substitution

(are_disjoints l (FV_env Γ)) ->
  (gen_type τ Γ) $\succ$ σ -> (gen_type τ Γ) $\succ$ (bind_list l σ)

(bind_list l σ) $\succ$ σ and (gen_type τ Γ) $\succ$ $\forall.\tau$

The proofs of these lemmas are clumsy and technical: most require to exhibit generic substitutions.

6 Conclusion

In this paper, we have specified in the Calculus of Inductive Constructions the abstract syntax, the type system and the dynamic semantics of a polymorphic functional fragment of the core ML language. We have verified one of the more fundamental properties, that is ML typing is sound. We have experimented two kinds of semantics: evaluation and reduction.

The ML language and its type system were often extended, mainly with the aim of offering more flexibility to the programmer: extensible records, mutable values, objects, overloading . . . Thus in order to validate these modifications, our formal development may be considered as a basis to investigate the properties of the new language (does it preserve or violate the properties established initially?). Beyond the necessary checking step by step, our objective is to develop a formal framework (based on the Calculus of Inductive Constructions and Coq) that allows to define type systems *à la ML* and to reason about them. Intuitively, it requires at the same time the construction of a library of formal components and a methodology for composing and re-using formal pieces.

Acknowledgements We thank Karim Berkani for his collaboration in the earlier verification of the `reductions_preserve_types` lemma in the monomorphic case. The polymorphic case presented in this paper has built on his work. Furthermore we thank the anonymous referees for their helpful comments.

References

1. B. Barras et al. The Coq Proof Assistant, Reference Manual, Version 6.1. INRIA, Rocquencourt, December 1996.
2. Samuel Boutin. Proving Correctness of the Translation from Mini-ML to the CAM with the Coq Proof Development System. Research report RR-2536, INRIA, Rocquencourt, April 1995.
3. Ana Bove. A Machine-assisted Proof that Well Typed Expressions Cannot Go Wrong. Chalmers University of Technology and Göteborg University, May 1998.
4. D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple Applicative Language: Mini-ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, August 1986. also available as research report RR-529, INRIA, Sophia-Antipolis, May 1986.
5. J. Despeyroux, F. Pfenning, C. Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. In *Proc. of TLCA'97*, LNCS 1210, Springer Verlag, 1997.
6. C. Dubois, V. Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. Journal of Automated Reasoning, Vol 23, nos 3-4, 319-346, 1999.

7. C. Dubois, V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions : domains as inductive predicates. Workshop *Mechanization of partial functions*, CADE'15, Lindau, Germany, July 1998.
8. J. Frost. A Case Study of Co-induction in Isabelle. Technical Report 359, University of Cambridge, Computer Laboratory, February 1995.
9. R. Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.
10. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Passau, Germany, 1987. Also available as a Research Report RR-601, Inria, Sophia-Antipolis, February 187.
11. H.P. Ko, M.E. Nadel. Substitution and refutation revisited. In K. Furukawa, editor, *Proc. 8th International Conference on Logic Programming*, 679-692, the MIT Press, 1991.
12. X. Leroy. Polymorphic typing of an algorithmic language. Research report 1778, INRIA, 1992 (French original also available).
13. L. Damas, R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages*, pages 207-212. ACM, 1982.
14. T. Nipkow, D. von Oheimb. Java$_{light}$ is Type-Safe - Definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, ACM Press, 1998, 161-170.
15. W. Naraschewski, T. Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. Journal of Automated Reasoning, Vol 23, nos 3-4, 299-318, 1999.
16. S. Michaylov, F. Pfenning. Natural Semantics and some of its meta-theory in Elf. In Lars Halln, editor, *Proceedings of the Second Workshop on Extensions of Logic Programming*, Springer-Verlag LNCS, 1991. Also available as a Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrucken, Germany, August 1991.
17. O. Müller, K. Slind. Treating Partiality in a Logic of Total Functions. *The Computer Journal*, 40(10), 1997.
18. D. Rémy, J. Vouillon. An effective object-oriented extension to ML. Theory And Practice of Object Systems, 4(1):27-50, 1998.
19. D. Syme. Proving Java type soundness. Technical Report 427, University of Cambridge, Computer Laboratory, 1997.
20. D. Terrasse. Encoding Natural Semantics in Coq. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology (A-MAST'95)*, LNCS 936. Springer-Verlag, July 1995.
21. D. Terrasse. Vers un Environnement d'Aide au Développement de Preuves en Sémantique Naturelle Thèse de Doctorat, Ecole Nationale des Ponts et Chaussées, 1995.
22. M. VanInwegen. Towards type preservation for core SML. University of Cambridge Computer Laboratory, 1997.
23. A. Wright, M. Felleisen. A Syntactic Approach to Type Soundness. Information and Computation, 115(1), pp.38-94, 1994.

This article was processed using the LaTeX macro package with LLNCS style