Certification of a type inference tool for ML: Damas-Milner within Coq

Catherine Dubois *
Université d'Évry Val d'Essonne

Valérie Ménissier-Morain † ‡
Université Paris 6

Abstract. We develop a formal proof of the ML type inference algorithm, within the Coq proof assistant. We are much concerned with methodology and reusability of such a mechanization. This proof is an essential step towards the certification of a complete ML compiler.

In this paper we present the Coq formalization of the typing system and its inference algorithm. We establish formally the correctness and the completeness of the type inference algorithm with respect to the typing rules of the language. We describe and comment the mechanized proofs.

Key words: ML, type system, type inference, calculus of inductive constructions, formal proofs

1. Introduction

Our goal is to realize a verified formal proof of the ML type inference algorithm, within the Coq proof assistant. Though this algorithm was proved a long time ago, this proof had never been mechanized entirely up to now. Simultaneously and independently of our work, Nazareth and Nipkow have carried out such a formal verification in the theorem prover Isabelle/HOL for simply-typed λ-terms [13] and then W. Naraschewski and T. Nipkow have done it for a polymorphic type discipline [12].
The certification of an ML compiler done in [2] does not deal with the type inference problem. However it is a major step during the compilation process and it should be incorporated into any compiler certification.

The certification is done within the Coq system. This proof assistant suits well to prove properties on programming languages because any abstract syntax can easily be encoded as an inductive type and Coq

* E-mail address: *Catherine.Dubois@lami.univ-evry.fr*
† E-mail address: *Valerie.Menissier@lip6.fr*
‡ Valérie Ménissier-Morain worked on that subject when she was at Université d'Évry Val d'Essonne

provides specialized tactics to handle inductive definitions. We present the Coq system more precisely in the next section.

In this paper, we first briefly describe the term algebra and the type system within the logical language of the Coq system: our language is close to the ML core language and provides parametric polymorphism *à la ML*. Then, we present the modelisation of the type inference algorithm named W as usual in the literature. Our guideline is to stick as close as possible to an implementation of the typing process written in a functional language: we want to verify the true implementation of a compiler. It means for example that the type inference algorithm is given in a functional style, avoiding Prolog style definitions as much as possible.

The certification of the type inference algorithm W is done in two steps: we prove the correctness and the completeness of W with respect to the typing rules of the language. We have completely developed the correctness and completeness proofs within the Coq system (version 6.1). The paper describes in detail the mechanized proofs and shows the choices and the difficulties. The last section briefly compares our certification with the one proposed by Naraschewski and Nipkow [12].

Beyond the challenge of mechanizing this classical proof, we are motivated by developing a methodology and a framework to handle this kind of proofs and we are concerned with the reusability of such a mechanization. For example the Damas-Milner framework and the type system for extensional polymorphism [7] have the same typing rules and the same type inference algorithm. These two type systems only differ by the substitution notion and the unification mechanism which both handle two different kinds of type variables in the case of extensional polymorphism. Consequently we want to evaluate how much of the proof of Damas-Milner's algorithm can be reused. However this is a long term motivation that we enter upon in the section 10 by considering a slightly different typing framework, i.e. the restriction to value polymorphism [18] in order to type imperative features.

2. The Coq proof assistant

We give here a brief presentation of the Coq interactive proof assistant. A more detailed description can be found in [1]. The Coq system allows the development of verified formal proofs. The axiomatizations and specifications are written in the logical language Gallina whose foundation is the Calculus of Inductive Constructions [15], a version of higher order typed λ-calculus whose types are themselves typed terms of the language, extended with inductive types, very close to ML

datatypes and inductive relations related to Prolog predicates. From the definition of an inductive construction, the Coq system automatically generates the associated induction principle and provides proof tools to manipulate them (e.g. the `Induction` and `Inversion` tactics).

The user can write functions and also recursive functions: facilities are provided to define functions with structural recursion (`Fixpoint` and `Recursive` constructs). Coq allows also the definition of functions by using pattern-matching (`Cases`) with a syntax very close to ML.

Coq permits the extraction of ML programs from proofs of specification. However in our work, this functionality is not used because our purpose does not require it. The reverse approach is also possible: a specialized tactic, named `Program`, applied to a real program written in ML-style, generates the lemmas to be proved in order to ensure the correctness of the program with respect to its specification [14]. This methodology could be tried in our case and we plan to investigate it in the future.

We shall give information about Coq syntax all along the paper when it will be necessary to understand the formalization. However in some places, we have adopted a mathematical syntax rather than Coq syntax. These transformations are systematic and essentially limited to quantifiers. When general polymorphic functions (e.g. `length`, `if`) or constructors (e.g. `cons`) are applied, we have omitted the type argument. We intentionally write no complete script of proofs for sake of readability.

3. The language

In this section we present the definition of the abstract syntax of the language we consider, i.e. a model of the core ML-language. An elegant presentation of a similar language with the associated typing proofs can be read in the first chapter of Leroy's thesis [11].

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

The recursion is presented in our formalism as an extension of the λ-abstraction in order to show explicitly that the recursive construction is concerned only with functions: the expression `Rec` $f\ x.e$ defines the recursive function named f, its parameter is x, its body is e. The approach of the recursion as extension of the `let` construct (as in most of the implementations of ML) would require supplementary tests to unable recursive non-functional values and would obscure our proofs. The expression `Rec` $f\ x.e$ is equivalent to the classical

ML sentence `let rec` f x `=` e `in` f. On the contrary, the expression `let rec` f x `=` e `in` e' is translated in our formalism in the expression `let` f `= Rec` f $x.e$ `in` e'.

In Coq, the language is described via the definition of the inductive type `expr`, that is an object of type `Set` according to the Coq type system:

```
Inductive expr: Set :=
  Const_int: nat -> expr
| Variable: identifier -> expr
| Lambda: identifier -> expr -> expr
| Rec: identifier -> identifier -> expr -> expr
| Apply: expr -> expr -> expr
| Let_in: identifier -> expr -> expr -> expr
```

In our context, an identifier is encoded as a natural number. However the type `identifier` could be let abstract with the assumption that it is equipped with a decidable equality.

Note: it would be easy to add to the language the standard supplementary language constructs (conditional expressions, pairs, lists ...). It would raise no special difficulty in the certification.

4. The type language

Intuitively, a type is associated to each expression. The type information relative to the variables is saved in an environment. A polymorphic function is introduced only via a `let` definition, polymorphic means that it can be applied to objects of different types. So its type contains quantified variables. The information about these quantified variables is necessary in the environment in order to verify the different applications of the function. Below we develop the notion of *type scheme* corresponding approximatively to a quantified type. Afterwards we specify the relation of *type instance* between two types and also the relation of *generic type instance* between a type and a type scheme.

4.1. Types and type schemes

The considered types are the basic type *int*, type variables denoted as usual with Greek letters α, β ...and functional types $\tau \to \tau'$ (where τ and τ' are types too). It is encoded in Coq as:

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

A type variable (also named a *stamp*) is encoded by a term like `(Stamp n)` where `n` is a natural number. For sake of readability, we sometimes liken it to a natural number.

A type scheme σ is defined as a type universally quantified over a finite set (possibly empty) of type variables: $\forall\alpha_1 \ldots \alpha_n.\tau$ (where τ is a type expression). The quantified variables are called the *generic variables* of the type scheme. A type scheme may also contain some free variables, for example in the type scheme $\forall\alpha.\alpha \to \beta$, α is a generic type variable and β a free variable. A type scheme without generic variables is said a *trivial* type scheme and written as $\forall.\tau$.

The previous (and usual) definition accentuates the generic variables but complicates the application of a substitution on a type scheme a bit. Our encoding in Coq does not follow this line, we prefer to distinguish syntactically free and bound variables, 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 :=
  Int_ts: type_scheme
| Var_ts: stamp -> type_scheme
| Gen_var: stamp -> type_scheme
| Arrow_ts: type_scheme -> type_scheme -> type_scheme
```

According to this definition, the type scheme $\forall\alpha.\alpha \to \beta$ 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.

4.2. Type instance and substitution

A type instance is defined relatively to the notion of substitution: a type τ' is a *type instance* of the type τ if there exists a substitution s such that $s\tau = \tau'$.

We specify a substitution as a list of pairs *(stamp, type expression to substitute)*. The operations defined on substitutions are the usual ones: to get the type related to a given stamp (`assoc_stamp_in_subst`), to apply a substitution on a type variable, a type or a type scheme (`apply_substitution, apply_subst_type, apply_subst_tscheme`), to compose substitutions (`compose_subst`), to compute the domain, the range of a substitution etc. These operations are specified in Coq in a functional style and are very close to their ML implementation. Let us detail two of them below: the application of a substitution on a type scheme `apply_subst_tscheme` and the composition of substitutions `compose_subst`.

```
Recursive Definition apply_subst_tscheme [s: substitution]:
type_scheme -> type_scheme:=
 Int_ts => Int_ts
|(Var_ts v) => (type_to_type_scheme (apply_substitution s v))
|(Gen_var v) => (Gen_var v)
|(Arrow_ts ts1 ts2) => (Arrow_ts (apply_subst_tscheme s ts1)
                                 (apply_subst_tscheme s ts2))
```

Coq notation: `Definition id [x: t]: t':= e` defines a function named `id` whose parameter is `x` of type `t`, whose body is the expression `e` declared with the type `t'`. Here the body of `apply_subst_tscheme` is a function that matches its anonymous parameter against the different patterns `Int_ts`, `(Var_ts v)`, `(Gen_var v)` and `(Arrow_ts ts1 ts2)`.

Applying a substitution `s` on a type scheme `ts` consists in replacing only the free variables (labeled with `Var_ts`) appearing in `ts` by the associated type expression in `s`: a conversion from type to type scheme is necessary and done by the `type_to_type_scheme` function.

```
Definition compose_subst:= [s1, s2: substitution]
(append  (subst_diff s2 s1) (apply_subst_list s1 s2))
```

Intuitively, the substitution `(compose_subst s1 s2)` (`s1` and then `s2`) is computed by applying `s2` to each type expression in `s1` (done by `(apply_subst_list s1 s2)`). We add to the resulting list those pairs whose first component is in the domain of `s2` but not in the domain of `s1` (`append` and `subst_diff`).

Many lemmas are required about these definitions. We detail some of them, particularly those dealing with the composition of substitutions.

```
Lemma composition_of_substitutions_stamp:
∀ s1, s2: substitution, ∀ st: stamp
  (apply_substitution (compose_subst s1 s2) st)
= (apply_subst_type s2 (apply_substitution s1 st))
```

The lemma `composition_of_substitutions_stamp` shows that the `compose_subst` operation performs on stamps what its name means: to apply the substitution denoted by `(compose_subst s1 s2)` on the stamp `st` produces the same term as to apply `s2` to the result of the application of `s1` on `st`. The proof of this lemma requires some supplementary technical lemmas. This part counts approximatively 600 lines detailed. We need to establish the relations between `assoc_stamp_in_subst` and the operations `append`, `subst_diff`, `apply_subst_list` and `compose_subst`. Unfortunately, these operations have very few good algebraic properties. Elimination on inductive structures, one of the most common way to start a proof

in Coq, performs almost no simplification of the problem and the proof deals with a complete combinatoric exploration of all conditions in these definitions.

As last example of properties required for substitutions, here is a proposition which concerns relations between the variables appearing in a type term τ (also called its free variables and computed by `FV_type`) and those of the type $(s\ \tau)$ where s is a substitution:

$$\alpha \in \texttt{FV\_type}(\tau) \Rightarrow \texttt{FV\_type}(s\ \alpha) \subset \texttt{FV\_type}(s\ \tau)$$

Remark: Let us notice that a substitution could also be encoded as a Coq abstraction of type `stamp -> type`. This encoding is interesting for the operations consisting in applying a substitution or composing substitutions, for example, we have for free the lemma `composition_of_substitutions`. We have investigated simultaneously this second encoding for a monomorphic restriction. It is pretty convenient until we have to exhibit the greatest stamp of the range of the substitution for example (and we need to compute it, as we shall see later). The other reason concerns our guideline i.e. to stay as near as possible to a real implementation of the typing tool.

4.3. Generic instance and generic substitution

A type τ' is a *generic instance* of a type scheme $\forall\alpha_1 \ldots \alpha_n.\tau$ if and only if there is a substitution s_g for $\alpha_1 \ldots \alpha_n$ such that $s_g(\tau) = \tau'$. The substitution s_g is called a *generic substitution*. For example, the type $int \to \beta$ is a generic instance of the type scheme $\forall\alpha.\alpha \to \beta$.

Consequently, the Coq specification has to provide the operation `apply_subst_gen` that applies a generic substitution on a type scheme in order to produce a generic instance: it only modifies the bound variables of the type scheme but keeps unchanged the free ones.

In our framework, a generic substitution is implemented as a list of type expressions without any reference to the names of the variables it applies on. In the following, a generic substitution is given the type `gen_subst`.

The type to be associated to the n-th generic variable can be found at the n-th list position. It means that the length of a generic substitution s_g applied to a type scheme σ must be at least[1] the greatest stamp p denoting a generic variable found in σ. Consequently (`apply_subst_gen` $s_g\ \sigma$) is partially defined. Then our specification simulates the exception mechanism. For that purpose, we use an inductive

[1] The generic substitutions built in our framework contain usually exactly p types, however the *at least* p condition is sufficient for us and also required in some lemmas.

type `gen_check` defined with two constructors, `Error_gen` applies when s_g cannot be considered as a valid generic substitution for σ, else `Some` introduces the resulting type expression[2].
The function `apply_subst_gen` is written in Coq as follows:

```
Fixpoint apply_subst_gen [s: gen_subst; ts: type_scheme]:
gen_check:=
  Cases ts of
    Int_ts => (Some_gen Int)
  | (Var_ts v) => (Some_gen (Var v))
  | (Gen_var (Stamp x)) => Cases (nth x s) of
                             Error_nth => Error_gen
                           | (Some_nth t) => (Some_gen t)
                           end
  | (Arrow_ts ts1 ts2) =>
      Cases (apply_subst_gen s ts1) of
        Error_gen => Error_gen
      | (Some_gen t1) =>
          Cases (apply_subst_gen s ts2) of
            Error_gen => Error_gen
          | (Some_gen t2) => (Some_gen (Arrow t1 t2))
          end
      end
  end.
```

According to the previous informal definitions, the predicate `is_gen_instance` that specifies when a type is a generic instance of a type scheme, is defined as:

```
Definition is_gen_instance:= [t: type] [ts: type_scheme]
  ∃ sg: gen_subst | (apply_subst_gen sg ts)=(Some_gen t)
```

5. The typing rules

This section presents the typing rules for our language. They all refer to a type environment, consequently we first specify this notion.

5.1. Environment

We have mentioned before the notion of environment as a list of pairs *(identifier, type scheme)*. We have retained this view in Coq and defined the type `environment` as `list (identifier * type_scheme)`.

[2] Another solution to encode the partial function `apply_subst_gen` would consist in introducing the definedness condition as a supplementary argument whose type can be derived as in [8]

The notation "$\Gamma \oplus y : \sigma$" denotes the environment Γ extended with the new information $y : \sigma$, that is the environment where the type informations found in Γ are kept unchanged, except for the identifier y which is bound to σ. The list representation of the environments allows a very simple implementation of the $\oplus$ function (`add_env` in Coq), it is simply the "cons" operator. Combined with an implementation of the operation $\Gamma(x)$ (`assoc_ident_in_env` in Coq) that returns the type associated to the *first* occurence of x in Γ, this naive representation ensures that the visibility rules of the language are respected.

Other operations like the application of a substitution to an environment are required in the following. This last operation `apply_subst_env` is easily defined as extension of `apply_subst_tscheme`.

The certification of the type inference algorithm needs several propositions about environments, for example the following one that we mention here for sake of clarity:

```
Lemma Ident_in_apply_subst_env:
∀ env: environment, ∀ i: identifier, ∀ ts: type_scheme,
(assoc_ident_in_env i env)=(Some_in_env ts) ->
   ∀ s: substitution,
    (assoc_ident_in_env i (apply_subst_env env s))
   =(Some_in_env (apply_subst_tscheme s ts))
```

This lemma states that $(s\Gamma)(i) = s(\Gamma(i))$, if i is one of the identifiers of the environment Γ. The Coq formulation looks clumsy because the function `assoc_ident_in_env` may fail. The proof is done by induction on Γ. If Γ is a non-empty list, we distinguish two cases: the first identifier of the environment is exactly i or different from i. This discrimination is performed via the elimination of the `identifier_dec` decision lemma for the identifiers.

```
Lemma identifier_dec:
  ∀ v1, v2: identifier, {v1=v2} ∨ {¬ (v1=v2)}
```

5.2. Type generalization

The `let` construct is the only one that introduces in the environment identifiers with polymorphic types, i.e. non trivial type schemes. 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).

This definition suggests the very simple and natural following implementation in Coq for the `gen_type` function:

```
Recursive Definition gen_type:
type -> environment -> type_scheme:=
     Int  env => Int_ts
| (Var v) env => (if_ (in_list_stamp v (FV_env env))
                   (Var_ts v)
                   (Gen_var v))
| (Arrow tau1 tau2) env => (Arrow_ts (gen_type tau1 env)
                                     (gen_type tau2 env))
```

However problems arise when two different types are turned into equivalent type schemes, in the sense that they denote the same set of types. For example, if α and β denote two distinct stamps not free in the considered environment, the types $\alpha \to \alpha$ and $\beta \to \beta$ produce the respective type schemes $\forall\alpha.\alpha \to \alpha$ and $\forall\beta.\beta \to \beta$. These two type schemes are encoded with two syntactically different terms in Coq. The type system and the inference algorithm do not require to decide whether two type schemes are equivalent or not, but unfortunately the proof of the correctness does! In fact, the only type schemes we need to test for equivalence are those produced by generalization. Consequently, in order to avoid to deal with alpha-conversion in the entire certification, our Coq implementation introduces a particular encoding for type schemes produced by the generalization process. We handle a type scheme linearly: any occurrence of the generic variable α in a type scheme σ is written as `(Gen_var n)` if α is the `n`-th generic variable discovered when σ is read from left to right. For example, the generalization of the type $(\alpha \to \beta) \to (\beta \to \alpha)$ when α and β are not free in the environment produces the type scheme $\forall\alpha\beta.(\alpha \to \beta) \to (\beta \to \alpha)$ represented by the Coq term `(Arrow_ts (Arrow_ts (Gen_var 0) (Gen_var 1)) (Arrow_ts (Gen_var 1) (Gen_var 0)))`.

The proposed encoding ensures that two type schemes produced by generalization that are equivalent with respect to alpha-conversion are syntactically equal. For example, (`gen_type` $\alpha\ \Gamma$) = `(Gen_var 0)` = (`gen_type` $\beta\ \Gamma$) if α and β are not free in Γ.

To define `gen_type` in Coq, we use an auxiliary recursive function `gen_type_aux`: `(gen_type_aux t env l)` computes the pair `(ts, l')` where `ts` is the generalization of `t` with respect to the environment `env` when `l` indicates the variables already discovered as generic and `l'` is

the list `l` at the end of which have been added the new generic variables produced by the computation. The index attributed to a generic variable relates to its position in `l'`. It is implemented in Coq via the function **index** that may fail when used to search a missing variable.

```
Fixpoint gen_type_aux [t: type]:
environment -> (list stamp) -> type_scheme*(list stamp):=
[env:environment] [l: (list stamp)]
  Cases t of
    Int => (Int_ts,l)
  | (Var v) =>
    (if_ (in_list_stamp v (FV_env env))
      ((Var_ts v), l)
      (Cases (index l v) of
         Stamp_not_in =>  ((Gen_var (Stamp (length l))),
                           (append l (cons v nil )))
       | (Index_in k) => ((Gen_var (Stamp k)), l)
       end))
  | (Arrow t1 t2) =>
     Cases (gen_type_aux t1 env l) of
     (ts1, l1) =>
       Cases (gen_type_aux t2 env l1) of
        (ts2, l2) => ((Arrow_ts ts1 ts2), l2)
       end
     end
  end

Definition gen_type :=
  [t: type] [env: environment] (Fst (gen_type_aux t env nil))
```

Remark: the **gen_type** operation turns a type into a type scheme in which each bound variable is mapped to an index computed by a bijective function with results in an integer interval starting from 0. However we do not impose to follow this condition upon the numbering of the generic variables in every type scheme handled in the specification or the proof. For example, we can accept the type scheme `(Arrow_ts (Gen_var 2) (Gen_var 0))` in the initial typing environment, it has no incidence on the type inference problem. In fact the initial environment is usually empty and all the type schemes added to it during the type inference phase are produced by **gen_type**.

5.3. The inference rules

The typing rules are described in the Natural Semantics style [10] (see figure 1). The sequent $\Gamma \vdash e : \tau$ means that the expression e has type τ under the environment Γ.
We have chosen to use the syntax-directed presentation of the rules in the style of [3]. The most important reason for that choice is that it

$$(\text{CST})\ \Gamma \vdash n : int$$

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

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

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

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

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

Figure 1. The typing rules

makes our proof easier: the shape of the λ-expression determines the unique applicable rule and the premises of the rules are only concerned with subexpressions of their subject. Structural induction is thus a powerful proof tool well suited for the manipulation of the typing rules. However, Dubois has proved formally in Coq the equivalence with the non-deterministic type system given by Damas and Milner (see [5] for details).
The typing rules are encoded in Coq as clauses of the inductive relation `type_of`, the translation is quite obvious here. A more general framework to automatically translate Natural Semantics of programming languages to Coq is proposed in [17].
Here is a fragment of the Coq encoding:

```
 Inductive type_of: environment -> expr -> type -> Prop :=
  type_of_int_const: ∀ env: environment, ∀ n: nat,
   (type_of env (Const_int n) Int)
| type_of_var: ∀ env: environment,
  ∀ x: identifier, ∀ t: type, ∀ ts: type_scheme,
  (assoc_Some_in_env x env)=(Some_in_env ts) ->
    (is_gen_instance t ts) ->
      (type_of env (Variable x) t)
| type_of_lambda: ∀ env: environment,
  ∀ x: identifier, ∀ e: expr, ∀ t, t': type,
  (type_of (add_env env x (type_to_type_scheme t)) e t') ->
    (type_of env (Lambda x e) (Arrow t t'))
...
```

Let us remark that this predicate is the only one defined *à la Prolog* in the Coq modelisation of the type system.

5.4. Principal type

This type system allows for example to derive that the sequent $\Gamma \vdash \lambda x.x : int \to int$ holds and also the two following ones: $\Gamma \vdash \lambda x.x : (int \to int) \to (int \to int)$ and $\Gamma \vdash \lambda x.x : \alpha \to \alpha$ (with α not free in Γ). The last type is the principal type of the expression λ`x.x`: it corresponds to the most general type that can be attached to the expression. The other types e.g. $int \to int$, $(int \to int) \to (int \to int)$ are instances of the principal type.

We inductively define the predicate `is_principal_type` below: (`is_principal_type` $e\ \Gamma\ \tau$) holds when τ is the principal type of the expression e with respect to Γ.

```
Definition is_principal_type:=
[e: expr] [tau: type] [env: environment]
  (∀ t: type, ∀ s: substitution,
  (type_of (apply_subst_env env s) e t) ->
    ∃ s1, s2: substitution | t = (apply_substitution s1 tau)
                           ∧ s = (compose_subst s1 s2))
```

6. The type inference algorithm

We consider an adaptation to our formalism of the well-known Damas-Milner type inference algorithm for ML (algorithm W of [4]) that computes the principal type of an expression, described below in a functional style: it uses the classical mechanism of unification: `unify`(τ_1, τ_2) computes the most general unifier of two types τ_1 and τ_2 if they are unifiable, fails otherwise.

6.1. Unification

The function `unify` has only been assumed in our modelisation. We have borrowed this methodology from Nazareth and Nipkow [13].

```
Inductive unify_check: Set :=
  Error_unify: unify_check
| Some_unify: substitution -> unify_check

Parameter unify : type*type -> unify_check
```

In fact the unification algorithm has no incidence on the certification of the inference tool since the type inference algorithm does not depend on its implementation. Thus, we have just introduced the properties relative to the correctness and the completeness of the `unify` function as axioms. Lastly, a third more technical axiom ensures that the unification algorithm computes a unifier whose free variables are only those contained in the initial problem.

Another study orthogonal to ours could be to encode a unification algorithm, for example the Robinson's algorithm and then prove the three previous axioms. Furthermore, in the Coq community, some work has already been done around unification [16] [9].

6.2. The type inference algorithm

Let e be a term and Γ a typing environment. We define $W(\Gamma, e)$ as the pair (τ, s), where τ is a type expression, s a substitution: τ is the most general type of e and s contains the instantiations performed during the computation (on free variables of Γ). The algorithm W is described in figure 2.

The encoding in Coq is very close to that definition and uses the `Fixpoint` construction. However, we have to introduce two supplementary features in the previous algorithm: the failure of the algorithm in the case of non-typable expressions and the management of the fresh variables. To solve the last point, the function takes as a supplementary argument the stamp of the last new type variable created. So, when the algorithm succeeds, it computes a triplet (τ, s, st) where τ and s are defined as previously and st is the stamp of the last new variable created. To take into account the possible failure of the algorithm, we have to simulate the exception mechanism with the inductive type `infer_check` very similar to `unify_check`.

```
Inductive infer_check: Set :=
  Error_infer: infer_check
| Some_infer: type -> substitution -> stamp -> infer_check
```

$W(\Gamma, n) = (int, \emptyset)$

$W(\Gamma, x) =$
  let $(\forall\ \alpha_1 \ldots \alpha_n.\tau) = \Gamma(x)$
  let $\beta_1, \ldots, \beta_n$ be fresh variables
  let $s = \{\alpha_i \mapsto \beta_i\}$
  in $(s\tau, \emptyset)$

$W(\Gamma, \lambda x.e) =$
  let α be a fresh variable
  let $\tau, s = W(\Gamma \oplus x : \forall.\alpha, e)$
  in $(s\alpha \to \tau, s)$

$W(\Gamma, \texttt{Rec}\ f\ x.e) =$
  let α, β be two fresh variables
  let $\tau, s =$
  $W(\Gamma \oplus f : \forall.\alpha \to \beta \oplus x : \forall.\alpha, e)$
  let $\mu = \texttt{unify}(\tau, s\ \beta)$
  in $(\mu\ o\ s(\alpha \to \beta), \mu\ o\ s)$

$W(\Gamma, e_1\ e_2) =$
  let $\tau_1, s_1 = W(\Gamma, e_1)$
  let $\tau_2, s_2 = W(s_1\Gamma, e_2)$
  let α be a fresh variable
  let $\mu = \texttt{unify}(s_2\tau_1, \tau_2 \to \alpha)$
  in $(\mu\alpha, \mu \circ s_2 \circ s_1)$

$W(\Gamma, \texttt{let}\ x = e_1 \texttt{in}\ e_2) =$
  let $\tau_1, s_1 = W(\Gamma, e_1)$
  let $\sigma = \texttt{gen\_type}\ \tau_1\ s_1\Gamma$
  let $\tau_2, s_2 = W(s_1\Gamma \oplus x : \sigma, e_2)$
  in $(\tau_2, s_2 \circ s_1)$

Figure 2. The W algorithm

We give here only the beginning of the Coq specification:

```
Fixpoint W [st: stamp; env: environment; e: expr]:
infer_check :=
  Cases e of
    (Const_int n) => (Some_infer Int ∅ st)
  | (Variable x) =>
      Cases (assoc_Some_in_env x env) of
        Error_in_env => Error_infer
      | (Some_in_env ts) =>
          Cases (compute_gen_subst st (max_gen_vars ts)) of
           (l, st') =>
            Cases (apply_subst_gen l ts) of
               Error_gen => Error_infer
             | (Some_gen t) => (Some_infer t ∅ st')
            end
          end
      end ...
```

In this fragment of code, `(compute_gen_subst st n)` computes the list of type variables `[st; st+1; ...; st+n-1]` and returns the current value of the stamp (here `st+n`).

7. Certification of the type inference algorithm

After having modeled the type system and the type inference tool, we are ready now to certify this tool, that is it computes the principal type of each typable expression or fails otherwise. This certification is done in two steps. First we prove that W is correct with respect to the typing rules. Then we show that W is complete, more precisely, if an expression admits a type, then W succeeds and computes the principal type of the expression.

Another way to realize this certification in Coq consists in using the tactic `Program`: then W is proposed as a realizer for the proof of the following lemma:

```
Lemma specification_W: ∀ e: expr, ∀ env: environment,
  {∃ tau: type | (is_principal_type e env tau)}
∨ { ∀ s: substitution, ∀ t: type, ¬ (type_of (s env) e t)}
```

We expect that one of the lemmas generated by the tactic `Program` will be for example the correctness lemma or instantiated forms of it. In the future, we plan to investigate also this possibility.

8. Correctness of the type inference algorithm

The correctness statement establishes that if $W(\Gamma, e)$ succeeds and computes the type τ and the substitution s then the sequent $s\Gamma \vdash e : \tau$ holds. This is encoded in Coq as:

```
Lemma correctness:
∀ e: expr, ∀ st, st': stamp, ∀ env, env': environment,
∀ t: type, ∀ s: substitution,
(W st env e) = (Some_infer t s st') ->
  (type_of (apply_subst_env env s) e t)
```

The correctness proof requires some fundamental lemmas used several times for example the main lemma
`typing_is_stable_under_substitution` that we explain in the next subsection. Then we'll come back to the correctness proof.

8.1. Typing is stable under substitution

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

This lemma corresponds to a classical property of the typing rules that states that if $\Gamma \vdash e : \tau$ holds then for any substitution s, the typing sequent $s\Gamma \vdash e : s\tau$ holds too. It means that if τ is a possible type for e with respect to Γ, we can obtain another type for e by applying a substitution on Γ and τ.

The proof of the lemma `typing_is_stable_under_substitution` arises no problem in the monomorphic case but becomes tedious in the polymorphic case, especially because of the generalization process. It requires large developments on substitutions that perform stamps renaming for example. This proof is done by induction on the expression e, most cases do not require much effort and follow from the induction hypothesis. The `identifier` and `let` cases require much more! In the remainder of this subsection, we point to the difficulties relative to both cases and we enumerate the different necessary lemmas.

8.1.1. *Let e be the identifier i*
By hypothesis, τ is a generic instance of the type scheme associated to i in Γ, that is $\sigma = \Gamma(i)$. Consequently, there exists a generic substitution s_g that maps σ to τ. We have to show that $s\tau$ is a generic instance of $s\sigma$. For that purpose, we show that the generic substitution obtained by applying the substitution s to each type appearing in s_g (this is done via the Coq function `map_apply_subst_type`) allows to transform the type scheme $s\sigma$ into the type $s\tau$.

This computation requires the next property which indicates how `apply_subst_type` distributes over `apply_subst_gen` : applying the substitution s to the type computed from the generic substitution s_g and the type scheme σ and applying the generic substitution (`map_apply_subst_type` s_g s) to the type scheme $s\sigma$ give the same result.

```
Lemma subst_gen_subst_type:
∀ ts: type_scheme, ∀ t: type, ∀ sg: gen_subst,
(apply_subst_gen sg ts) = (Some_gen t) ->
  ∀ s: substitution,
    (apply_subst_gen (map_apply_subst_type sg s)
                     (apply_subst_tscheme s ts))
   =(Some_gen (apply_subst_type s t))
```

8.1.2. *Let e be the expression* `let` x `=` e_1 `in` e_2
According to the (LET) typing rule, the hypothesis on e allows to deduce that the two following sequents hold:

$$\Gamma \vdash e_1 : \tau_1 \text{ and } \Gamma \oplus x : (\texttt{gen\_type}\ \tau_1\ \Gamma) \vdash e_2 : \tau$$

We have to find a type τ_1' such that the sequents below are satisfied:

$$s\Gamma \vdash e_1 : \ \tau_1' \text{ and } s\Gamma \oplus x : (\texttt{gen\_type}\ \tau_1'\ s\Gamma) \vdash e_2 : \ s\tau.$$

At a first glance, we are tempted to say that it is a trivial case following from the induction assumptions on e_1 and e_2 with $\tau_1' = s\tau_1$. Not at all! The difficulty comes from the relation between applying a substitution ϕ and making a generalization: more precisely, the two operations commute only on condition that the substitution ϕ is not concerned with the type variables quantified during the generalization. The following lemma lays it down:

```
Lemma gen_in_subst_env:
∀ env: environment, ∀ s: substitution, ∀ t: type,
(are_disjoint (FV_subst s) (gen_vars t env)) ->
  (apply_subst_tscheme s (gen_type t env))
 =(gen_type (apply_subst_type s t) (apply_subst_env env s))
```

Let us take a counter-example to illustrate this pre-condition on `s`. Let `t` be the type $\alpha \to int$ with α not free in `env`, `s` is the substitution that maps α to int. The generalization of `t` with respect to `env` is the type scheme $\forall\alpha.\alpha \to int$ which remains unchanged when `s` is applied; the other generalization results in the type scheme $\forall.int \to int$.

As for all the propositions manipulating the operations `gen_type`, this lemma is proved via an auxiliary lemma `gen_in_subst_env_aux` proved by induction on `t`. We write it below. We can think of it as an invariant property verified at each step of the computation of `(gen_type t env)`.

```
Lemma gen_in_subst_env_aux:
∀ env: environment, ∀ s: substitution, ∀ t: type,
(are_disjoint (domain_of_subst s) (gen_vars t env)) ->
(are_disjoint (range_of_subst s) (gen_vars t env)) ->
  ∀ l: (list stamp),
    (gen_type_aux (apply_subst_type s t)
                  (apply_subst_env env s) l)
  = ((apply_subst_tscheme s (Fst (gen_type_aux t env l))),
     (Snd (gen_type_aux t env l)))
```

Thus, the approach to solve the current goal, that is the `let` case in the stability proof, consists in computing a substitution ρ that renames the stamps in τ_1 that may interact with s (that is (`gen_vars` $\tau_1\ \Gamma$)) into stamps not appearing in s and free for Γ. We take as new stamps numbers from the greatest stamp found in (`gen_vars` $\tau_1\ \Gamma$), (`FV_subst` s) and (`FV_env` Γ). The applicability condition is now satisfied and then we can apply the lemma `gen_in_subst_env` with the substitution s and the type $\rho\tau_1$. The following equality (E) is verified:

$$s(\texttt{gen\_type}\ \rho\tau_1\ \rho\Gamma) = (\texttt{gen\_type}\ s\rho\tau_1\ s\rho\Gamma)$$

We can rewrite the expression $\rho\Gamma$ over the expression Γ because of the construction of ρ and the equality becomes:

$$s(\texttt{gen\_type}\ \rho\tau_1\ \Gamma) = (\texttt{gen\_type}\ s\rho\tau_1\ s\Gamma)$$

We prove then that the type schemes (**gen_type** $\rho\tau_1$ Γ) and (**gen_type** τ_1 Γ) are *syntactically* equal because of our canonical encoding (see the lemma **gen_renaming** in the next subsection. And then, the previous equality (E) becomes:

$$s(\texttt{gen\_type}\ \tau_1\ \Gamma) = (\texttt{gen\_type}\ s\rho\tau_1\ s\Gamma).$$

We can now easily conclude the initial proof, let us take $\tau_1' = s\rho\tau_1$. The first sequent $s\Gamma \vdash e_1 : s\rho\tau_1$ follows from the inductive assumption relative to e_1 with the substitution $s\rho$ and the second one $s\Gamma \oplus x : (\texttt{gen\_type}\ s\rho\tau_1\ s\Gamma) \vdash e_2 : s\tau$ follows from the inductive assumption relative to e_2 with the substitution s and the equality (E).

This part of the proof needs a lot of definitions and theorems, in particular material on the renaming substitutions on which we focus in the next subsection.

8.2. Renaming substitutions

A *renaming substitution* is a substitution with specific features:

- each variable in the domain is associated to another type variable
- the domain and the range of the substitution are disjoint
- two distinct variables in the domain have different images.

Although renaming substitutions are also substitutions, we have for example specific operations for computing the domain and the range and mapping a renaming substitution on a list. For that reason, we prefer to define in Coq a specific type **ren_subst** for the renaming substitutions, they are implemented as lists of pairs of stamps (`list stamp*stamp`). To reuse operations on ordinary substitutions, we have to coerce any renaming substitution into a substitution (done by **rename_to_subst**).

Here is the Coq definition of the predicate **is_rename_subst**:

```
Definition is_rename_subst:= [rho: ren_subst]
(are_disjoint (domain_of_ren rho) (range_of_ren rho)) ->
  (∀ x, y: stamp,
   (in_list_stamp x (domain_of_ren rho)) = true ->
   (in_list_stamp y (domain_of_ren rho)) = true ->
   ¬ x=y ->
     ¬ (apply_ren_subst rho x)=(apply_ren_subst rho y))
```

The following lemma determines under which circumstances the type `t` and the type obtained by renaming its generalized variables give identical type scheme under generalization.

```
Lemma gen_renaming: ∀ env: environment,
∀ rho: ren_subst, ∀ t: type, ∀ s: substitution,
(is_rename_subst rho) ->
(domain_of_ren rho)=(gen_vars t env) ->
(are_disjoint (range_of_ren rho) (FV_env env) ->
(are_disjoint (range_of_ren rho) (FV_subst s)) ->
   (gen_type t env)
  =(gen_type (apply_subst_type (rename_to_subst rho) t) env)
```

As previously, this lemma is proved via an auxiliary lemma proved by induction on `t`. A consistent theory about disjoint lists and subsets is a by-product of this part of the work.

8.3. Come back to the correctness proof

The proof is done by induction on the expression e. In each case, we first introduce and save in the context the constraints on variables that derive from the fact that the computation $W(\Gamma, e)$ does not fail. This method can be compared with the predefined `Inversion` tactics useful for inferring facts from inductive predicates. This kind of manual inversion is performed by doing the elimination of the related inversion lemma proved independently.
We exemplify here the inversion lemma for the case of the abstraction: if W succeeds for $\lambda x.e$ with the type τ and the substitution s, necessarily τ is a functional type of the form $\alpha \rightarrow \tau'$, where α is a new type variable and W succeeds when applied on e and $\Gamma \oplus x : \forall.\alpha$. It is written in Coq as:

```
Lemma W_lambda_inversion: ∀ n: nat, ∀ st': stamp,
∀ env: environment, ∀ e: expr, ∀ x: identifier,
∀ t: type, ∀ s: substitution,
(W (Stamp n) env (Lambda x e))=(Some_infer t s st') ->
  ∃ t1: type |
     (W  (Stamp (S n)) (add_env env x (Var_ts (Stamp n))) e)
    =(Some_infer  t1 s st')
    ∧ t=(Arrow (apply_subst_type s (Var (Stamp n))) t1)
```

Our proof of this lemma consists in unrolling the computation of `(W (Stamp n) env (Lambda x e))` and eliminating every `Cases` expression. It requires several decision lemmas such as for example the trivial one about the `infer_check` type:

```
Lemma infer_check_inv: ∀ i: infer_check,
  ∃ t: type, ∃ s: substitution, ∃ st: stamp |
  {i=(Some_infer t s st)}∨ {i=Error_infer}
```

As for the proof of the stability lemma, both `identifier` and `let` cases are the most interesting ones in the correctness proof. We detail them below.

8.3.1. *Let e be the identifier i*
The present step leads to demonstrate that the type obtained by mapping each generic variable of a type scheme σ to a fresh type variable is a generic instance of σ. This proposition is formulated in Coq below and proved without any difficulty.

```
Lemma compute_gen_subst_create_generic_instance:
∀ ts: type_scheme, ∀ st: stamp, ∀ t: type,
 (apply_subst_gen (Fst (compute_gen_subst st
                                          (max_gen_vars ts)))
                  ts) = (Some_gen t) ->
   (is_gen_instance t ts)
```

8.3.2. *Let e be the expression* `let` x `=` e_1 `in` e_2
The fact that W succeeds and yields τ and s implies that `W` succeeds also for e_1 and computes a type τ_1 and a substitution s_1. It implies also that W returns a type τ_2 and a substitution s_2 for e_2 with the environment $s_1\Gamma \oplus x : (\textbf{gen\_type}\ \tau_1\ s_1\Gamma)$. Then the type τ is equal to τ_2 and s is exactly $s_2 \circ s_1$. The two following sequents follow from induction hypothesis on e_1 and e_2:

$$s_1\Gamma \vdash e_1 :\ \tau_1 \quad \text{and} \quad s_2(s_1\Gamma \oplus x :\ (\textbf{gen\_type}\ \tau_1\ s_1\Gamma)) \vdash e_2 :\ \tau_2$$

Here we use the same techniques as in the `let` case in the proof of the stability lemma. We introduce a renaming substitution ρ for the generalized variables of τ_1 with respect to $s_1\Gamma$, not concerned with the free variables in $s_1\Gamma$ and s_2.
The application of the lemma `typing_is_stable_under_substitution` (with $s_2\rho$ as substitution) allows to write the following sequent:

$$s_2\rho s_1\Gamma \vdash e_1 :\ s_2\rho\tau_1$$

The theorem `gen_renaming` can be applied and then

$$(\textbf{gen\_type}\ \tau_1\ s_1\Gamma) = (\textbf{gen\_type}\ \rho\tau_1\ s_1\Gamma)$$

By the lemma `gen_in_subst_env`, we can deduce that:

$$s_2(\textbf{gen\_type}\ \rho\tau_1\ s_1\Gamma) = (\textbf{gen\_type}\ s_2\rho\tau_1\ s_2s_1\Gamma)$$

Rewriting $\rho S_1\Gamma$ to $s_1\Gamma$ gives:

$$s_2 s_1 \Gamma \vdash e_1 : \ s_2 \rho \tau_1 \quad \text{and}$$
$$s_2 s_1 \Gamma \oplus x : \ (\texttt{gen\_type}\ s_2 \rho \tau_1 \ s_2 s_1 \Gamma) \vdash e_2 : \ \tau_2,$$

from which we establish that

$$s_2 s_1 \Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \ \tau_2$$

holds, QED.

Remark: another possible way to verify this case might be to characterize precisely the substitutions computed by W in order to state that s_2 has nothing to do with the generalized variables of τ_1 with respect to $s_1\Gamma$. Thus the renaming substitution ρ would not be necessary and we could use directly the lemma `gen_in_subst_env` with $s_2\tau_1$ and $s_1\Gamma$. However such a characterization cannot be expressed easily. Furthermore the stability of typing under substitutions is an intrinsic property of the type system, consequently it is interesting to prove it in a formal way, independently of the type inference tool. It is also required to establish the *subject reduction theorem*, formally proved within Coq by Dubois [5].

9. Completeness of the type inference algorithm

9.1. Formulation of the completeness statement

Roughly speaking, the completeness of W means that *if one can propose a solution to the typing problem of the expression e with respect to the environment* Γ*, then the type inference process will succeed and yield the most general solution which implies the proposed solution.* More formally, the completeness statement establishes that if a type τ' can be associated to e with respect to Γ where some free variables have been instantiated (via a substitution ϕ), then for a "correct" stamp st, (`W` $st\ \Gamma\ e$) succeeds and computes a type τ and a substitution s, furthermore τ' and ϕ can be deduced from τ and s. We refine this last property as follows: τ' is an instance of τ and if s' is the substitution such that $s'\ \tau = \tau'$ then applying ϕ means to apply s and then s'. Of course the equality between ϕ and $s' \circ s$ makes sense only for stamps appearing in the derivation of $\phi\Gamma \vdash e : \ \tau'$.

Upwards, we put a pre-condition on st which was supposed to be "correct", the stamp st is expected here to be a *a new type variable*, i.e. not used elsewhere (more precisely in the free type variables of Γ) and the same for the stamps greater than st. This is expressed by the predicate `new_tv_env` that we define and study in the subsection 9.2.

Let us write now the completeness lemma in Coq:

```
Lemma completeness: ∀ e: expr, ∀ env: environment,
 ∀ t': type, ∀ phi: substitution, ∀ st: stamp,
(type_of (apply_subst_env env phi) e t') ->
(new_tv_env env st) ->
  ∃ s, t, st', s' |
  (W st env e) = (Some_infer t s st')) ∧
   t'=(apply_subst_type s' t) ∧
   (∀ x : stamp, x < st ->
       (apply_substitution phi x) =
       (apply_substitution (compose_subst s s')  x))
```

At this point of the certification, we need to define and manipulate a relation over type schemes that expresses that a type scheme is more general than another. We introduce it in the subsection 9.3 before coming back to the proof of the completeness of W.

9.2. New type variables

The notion of new type variable is defined relatively to a type, a type scheme, an environment or a substitution. Then a type variable st, implemented in our context as a natural number, is considered as a new type variable for a given structure r if st is greater than any free variable occurring in r. This view imposes itself because W increments the counter argument every time a new type variable is needed. Nazareth and Nipkow [13] characterize the notion of new type variables in the same way.

For example the inductive predicate **new_tv_tscheme** implements this definition in the case of type schemes. Similar predicates are defined also for types **new_tv_type**), environments (**new_tv_env**) and substitutions (**new_tv_subst**).

```
Inductive new_tv_tscheme: type_scheme -> stamp -> Prop:=
  new_tv_ts_Int: ∀ st: stamp, (new_tv_tscheme Int_ts st)
| new_tv_ts_Var: ∀ st,st': stamp, st < st' ->
    (new_tv_tscheme (Var_ts st) st')
| new_tv_ts_Gen: ∀ st,st': stamp
    (new_tv_tscheme (Gen_var st) st')
| new_tv_ts_Arrow: ∀ ts1, ts2 : type_scheme, ∀ st: stamp,
  (new_tv_tscheme ts1 st) -> (new_tv_tscheme ts2 st) ->
    (new_tv_tscheme (Arrow_ts ts1 ts2)  st).
```

Besides these definitions, we establish some technical lemmas, for example **new_tv_compose_subst**:

```
Lemma new_tv_compose_subst:
∀ s1,s2: substitution, ∀ st: stamp,
(new_tv_subst s1 st) ->
  (new_tv_subst s2 st) ->
    (new_tv_subst (compose_subst s1 s2) st)
```

Some theorems as for example **new_tv_W** and **stamp_increases_in_W** characterize the behaviour of W with respect to new type variables. The lemma **new_tv_W** emphasizes that the stamp computed by W is a new type variable for the computed type and substitution (according to the validity of the input stamp). Both assertions are proved by induction on the typed expression. They require a lot of supplementary properties (around 500 lines).

```
Lemma stamp_increases_in_W: ∀ e: expr, ∀ env: environment,
∀ st,st': stamp, ∀ t: type, ∀ s: substitution,
(W st env e) = (Some_infer t s st') -> st ≤ st'

Lemma new_tv_W: ∀ e: expr, ∀ env: environment,
∀ st,st': stamp, ∀ t: type, ∀ s: substitution,
(new_tv_env env st) ->
(W st env e) = (Some_infer t s st') ->
  (new_tv_type t st') ∧ (new_tv_subst s st')
```

9.3. The *more general* relation over type schemes

A type scheme σ_1 is said *more general* that a type scheme σ_2 and written $\sigma_1 \succ \sigma_2$ or (**more_general** σ_1 σ_2) in Coq, if and only if every generic type instance of σ_2 is also a generic type instance of σ_1. For example, $\forall\alpha\beta.\alpha \to \beta$ is more general than $\forall\alpha.\alpha \to \alpha$.
The translation is straightforward in Coq:

```
Definition more_general
: type_scheme -> type_scheme -> Prop :=
[ts1, ts2 : type_scheme]
(∀ t : type,
 (is_gen_instance t ts2) -> (is_gen_instance t ts1))
```

This definition is extended classically upon the environments: an environment Γ_1 is said more general that the environment Γ_2 ($\Gamma_1 \succ \Gamma_2$) if and only if Γ_1 and Γ_2 are relative to the same identifiers $x_1, x_2 \ldots x_n$ and $\forall i \in [1, n],\ \Gamma_1(x_i) \succ \Gamma_2(x_i)$.
The definition we have encoded in Coq is more restrictive than the previous one, however it is sufficient in our context: we enforce that the identifiers $x_1, x_2 \ldots x_n$ occur at the same position in Γ_1 and Γ_2.

```
Inductive more_general_env:
environment -> environment -> Prop:=
  more_general_nil: (more_general_env nil nil)
| more_general_cons: ∀ env1, env2: environment,
  ∀ i: identifier, ∀ ts1, ts2 : type_scheme),
  (more_general_env env1 env2) ->
  (more_general ts1 ts2) ->
    (more_general_env (cons (i,ts1) env1)
                      (cons (i,ts2) env2))
```

The proof of the completeness of W uses still another property on the typing rules: if we can prove that the type of the expression e is τ under the environment Γ_2, then we can also prove it under a more general environment Γ_1.

```
Lemma typing_in_a_more_general_env:
∀ e: expr, ∀ env2, env1: environment, ∀ t: type,
(more_general_env env1 env2) ->
(type_of env2 e t) ->
   (type_of env1 e t)
```

This lemma is easily proved by induction on `e`. The `let` case relies on the following property:

```
Lemma more_general_gen_type:
∀ env2, env1: environment, ∀ t: type,
(more_general_env env1 env2) ->
  (more_general (gen_type t env1) (gen_type t env2))
```

It is interesting to have a look on the proof of this last theorem because this kind of proof is repeated several times. Thus for a type instance τ of `(gen_type t env2)`, let us call $s_g = [t_1; \ldots; t_n]$ the corresponding generic substitution, we build a generic substitution s'_g that transforms `(gen_type t env1)` into τ as follows: let $\alpha_1 \ldots \alpha_n$ be the generalized variables in `t` with respect to `env2` (discovered in this order). We compute the substitution $\phi = \{(\alpha_1, t_1) \ldots (\alpha_n, t_n)\}$ (by the `product_list` function) that is *equivalent* to s_g (in the sense that s_g `(gen_type t env2)` $= \phi$ `t`) then the generic substitution s'_g is $[\phi(\beta_1); \ldots; \phi(\beta_k)]$ where $\beta_1 \ldots \beta_k$ are the generalized type variables of `t` with respect to `env1`. We have to check that s'_g `(gen_type t env1)` $= \tau$. In fact, the proved property has a more complex and less natural formulation because of the intermediate function `gen_type_aux`.

9.4. Proof of the completeness

The proof we have mechanized follows the plan detailed in [11]. It is done by an induction on the expression e. The verification of every induction case can be sketched as follows: from the hypothesis we deduce that the computation of W succeeds and we exhibit its results, essentially a substitution s and a type τ. This first step may be heavy, particularly for the application and the `Rec` construct (essentially because we have to take into account most general unifiers). Then we have to imagine the substitution s' that allows to recreate the proposed solution from s and τ and then verify that it fits. In general, this phase requires an important computational development.

As for the correctness part, the difficulty comes from `let`, more precisely from the generalization. It is again the only case where a direct use of the inductive hypothesis is forbidden. We detail this case below in order to show exactly where the order $\succ$ is needed.
From the hypothesis $\phi\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau'$, we deduce the sequents

$$\phi\Gamma \vdash e_1 : \tau_1' \quad \text{and} \quad \phi\Gamma \oplus x : (\texttt{gen\_type}\ \tau_1'\ \phi\Gamma) \vdash e_2 : \tau'.$$

By applying the inductive hypothesis relative to e_1, it comes:

$$\begin{array}{c} W\ e_1\ \Gamma\ st = (\tau_1, s_1, st_1) \text{ and} \\ \exists s_1',\ s_1'\tau_1 = \tau_1' \text{ and } \phi\Gamma = s_1'(s_1\Gamma). \end{array}$$

The success of W for the `let` expression requires the successful computation of $(W\ (s_1\Gamma \oplus x : (\texttt{gen\_type}\ \tau_1\ s_1\Gamma))\ e_2\ st_1)$. To obtain that result from the inductive hypothesis on e_2, we need to show for example that the sequent

$$s_1'(s_1\Gamma \oplus x : (\texttt{gen\_type}\ \tau_1\ s_1\Gamma)) \vdash e_2 : \tau' \quad (H)$$

holds. At this point of the demonstration, the solution comes from the fundamental relationship between applying a substitution and generalizing a type (we have already shown in this paper that these two operations do not commute): *generalizing a type τ and then applying to it a substitution ϕ produces a type scheme more general than doing this in the reverse order.*

$$\phi(\texttt{gen\_type}\ \tau\ \Gamma) \succ (\texttt{gen\_type}\ \phi\tau\ \phi\Gamma).$$

The proof of this property follows exactly the same scheme than for the lemma `more_general_gen_type`, the generic substitution to find is just a little bit more complex!
Thus, in the present case, it follows that

$$s_1'(\texttt{gen\_type}\ \tau_1\ s_1\Gamma) \succ (\texttt{gen\_type}\ s_1'\tau_1\ s_1'(s_1\Gamma)) \text{ rewritten as}$$
$$s_1'(\texttt{gen\_type}\ \tau_1\ s_1\Gamma) \succ (\texttt{gen\_type}\ \tau_1'\ \phi\Gamma).$$

We deduce

$$\phi\Gamma \oplus x : s_1'(\texttt{gen\_type}\ \tau_1\ s_1\Gamma) \succ \phi\Gamma \oplus x : (\texttt{gen\_type}\ \tau_1'\ \phi\Gamma).$$

The lemma `typing_in_a_more_general_env` applies to this last relation and the sequent $\phi\Gamma \oplus x : (\texttt{gen\_type}\ \tau_1'\ \phi\Gamma) \vdash e_2 : \tau'$ to assert the typing sequent (H).

That's it. We reach the end of the proof and the end of the certification of W!

10. A first step to reuse

In this section we address the certification of the type inference tool when references are incorporated in the language. Several solutions to integrate imperative features with Milner polymorphism have been devised, we retain in our context the simpler one which becomes in fact the more usual one now in the ML community, proposed by Wright [18]. It consists in limiting polymorphism to values, i.e. to `let`-expressions where the binding is a syntactic value (for us, an integer constant, an identifier or an abstraction, recursive or not).

The main interest of this variant in our context is the abstract syntax for this extended language and its encoding in Coq. We have chosen to fix the value polymorphism in the syntax itself. For example, an expression like `let` x `=` e_1 e_2 `in` e_3 is forbidden, nevertheless it is equivalent here to $(\lambda x.e_3)(e_1\ e_2)$ dynamically and statically. Then, instead of having one syntactical sort and then one inductive type `expr` in Coq, we introduce two sorts: one for syntactic values `Left_Let_expr` and `expr` for all the λ-expressions. Of course, any syntactic expression is also a λ-expression, then the sort `Left_Let_expr` is a subsort of the other one.

The `let` construct is described as follows: `let` x `=` *Left_Let_expr* `in` *expr*.

We propose to encode this new abstract syntax within Coq via mutual inductive definitions:

```
Mutual Inductive Left_Let_expr: Set :=
  Const_int: nat -> Left_Let_expr
| Variable: identifier -> Left_Let_expr
| Lambda: identifier -> expr -> Left_Let_expr
| Rec: identifier -> identifier -> expr -> Left_Let_expr
with expr : Set :=
  Coerce : Left_Let_expr -> expr
| Apply: expr -> expr -> expr
| Let_in: identifier -> Left_Let_expr -> expr -> expr.
| Ref: expr -> expr
| Deref: expr -> expr
| Assign: expr -> expr -> expr
```

The terms of the language involve the new constructors `Ref`, `Deref` and `Assign` which are associated to respectively the allocation of a reference cell, the extraction of the cell's contents and the assignment of a value to a reference cell. The constructor `Coerce` is introduced to include elements of the inductive set `Left_Let_expr`.

The type system and the inference algorithm need to be updated and extended according to this new syntax. However there is no change for the old expressions. We detail below the typing rule for the assignment,

we write $\tau\ ref$ the type of a cell whose contents has the type τ:

$$\frac{\Gamma \vdash e_1 : \tau\ ref, \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{Assign}\ e_1\ e_2) : \tau}$$

The certification of W replays the same proof, we only have to take into account the expressions relative to references but it raises no difficulty at all, the associated goals follow usually from the inductive hypothesis.

11. Comparison with Naraschewski and Nipkow's proof

In this section we briefly compare our approach to the one taken by Naraschewski and Nipkow [12]. Their specification of the ML language, the type system and the inference algorithm is similar to ours (except for the constants and the `Rec` expressions they do not take into account). However we notice two significant differences: the encoding of the substitution and the generalization function. They implement a substitution as a function from variables to types. Amazingly their proof requires no restriction to finite functions. And their generalization function is the simpler and the more elegant one, without any particular encoding. This is essentially what we proposed at the beginning of our experience reported in [6]. The reason why they can retain this solution (and consequently the reason why we choose another one) can be found in their proof of the stability statement, in the `let` case: at the point of the demonstration where one needs to compare two type schemes identical up to alpha-conversion, they use the $\succ$ order and the associated lemmas, e.g. `typing_in_a_more_general_env`.

Last of all, we mention that the correctness property proved by Naraschewski and Nipkow requires that the stamp given as a parameter to W is a new type variable with respect to the considered environment in the typing sequent. Our certification does not require such a hypothesis: in that sense, our correctness property is very close to the one found in the informal proofs.

We can say that both proofs use more or less the same set of lemmas.

12. Conclusion

We can summarize the work described in this paper as follows:

- we have formalized within the Coq proof assistant a specification of the polymorphic type discipline of ML as described in [4] [3],

- we have implemented in a functional way the type inference algorithm W within Coq,
- we have proved that W is correct and complete with respect to the typing rules of the specification.

An important by-product affects the notion of substitution. We have isolated and specified three kinds of substitutions: the (common) substitution that binds free variables to terms, the (renaming) substitution that renames some variables and the (generic) substitution that binds the bound variables of a term.
We have demonstrated that the certification of W is tractable within the Coq system. It is a heavy proof requiring to handle sophisticated theories (substitutions, renaming, unification, fresh variables and so on). The specification of the problem and the proofs count 7371 lines detailed, more precisely 91 definitions and 322 lemmas, and also a very large piece of work with much backtracking.
The certification of the type inference tool needed to be familiar with the very details of the paper proof and furthermore to go beyond it, for example the part about the management of the new type variables is always ignored in a mathematical proof but it cannot be ignored in any formal proof.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliâtre, E. Giménez, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi and B. Werner. The Coq Proof Assistant, Reference Manual, Version 6.1. INRIA, Rocquencourt, December 1996, also available at `http://pauillac.inria.fr/coq/doc/main.html`.
2. S. 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. 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.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.
5. C. Dubois. Sûreté du typage de ML : Spécification et Preuve en Coq. 9èmes Journées Francophones des Langages Applicatifs, Côme, Italie, 1998.
6. C. Dubois and V. Ménissier-Morain. A proved type inference tool for ML: Damas-Milner within Coq (work in progress). In *Supplementary Proceedings of Theorem Proving in Higher Order Logics*, J. von Wright, J. Grundy and J. Harrison, editors, pages 15-30, Turku Centre for Computer Science, 1996.
7. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, pages 118–129, January 1995.

8. C. Dubois and V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. CADE-15, Workshop on Mechanization of Partial Functions, Lindau, 1998.
9. M. Jaume. Unification : a Case Study in Transposition of Formal Properties. In *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: Poster session TPHOLs'97*, E.L. Gunter and A. Felty editors, pages 79-93, Murray Hill, N.J., 1997.
10. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, 1987.
11. X. Leroy. Polymorphic typing of an algorithmic language. research report (english version of his PhD thesis at université Paris 7) RR-1778, INRIA, Rocquencourt, 1992.
12. W. Naraschewski and T. Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. In *Journal of Automated reasoning*, same issue.
13. D. Nazareth and T. Nipkow. Formal Verification of Algorithm W: The Monomorphic case In *Proceedings of Theorem Proving in Higher Order Logics*, LNCS 1125, Springer-Verlag, 331-345, 1996.
14. C. Parent. Developing certified programs in Coq - The Program Tactic. In H. Barendregt and T. Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, LNCS 806, pages 291–312. Springer-Verlag, 1993.
15. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, LNCS 442. Springer-Verlag, 1990.
16. J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. INRIA-Lorraine, Research report 1795, november 1992.
17. D. Terrasse. Encoding Natural Semantics in Coq. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology (AMAST'95)*, LNCS 936. Springer-Verlag, July 1995.
18. A. K. Wright. Simple Imperative Polymorphism. *Lisp and Symbolic Computation*, 1994.

Address for correspondence:
Catherine Dubois, LaMI, CNRS EP738, Université d'Évry Val d'Essonne – 4, Bd. des Coquibus – F-91025 Évry Cedex FRANCE
and
Valérie Ménissier-Morain, LIP6, CNRS 7606, Université Pierre et Marie Curie – 4, Place Jussieu – F-75252 Paris 05 FRANCE