Proving Type Soundness of a Simply Typed ML-Like Language with References

Olivier Boite and Catherine Dubois

CEDRIC-IIE (CNAM), 18 allée Jean Rostand F-91025 EVRY, France
{boite,dubois}@iie.cnam.fr

Abstract. In this paper, we formalize in the Coq proof assistant an ML-like language with imperative features, a monomorphic type system, a reduction semantics, and the type soundness. We consider this language as an extension of Mini-ML, and emphasize the consequences on the definitions and the proofs due to this extension.

1 Introduction

This paper reports on the machine-checked proof of the type soundness for a kernel of the ML language incorporating imperative features (references, assignments, sequences). Type soundness is a safety property that relates typing and evaluation : the evaluation of a well-typed program either loops, or terminates (by computing a value). Thus it never fails because of a type error or in other words, according to Milner, *Well-typed programs do not go wrong.*

Different formal methods exist to describe the semantics of a language, we have chosen to use a reduction operational semantics (also called small-step semantics) as it is exemplified in [16]. Such a method allows to separate the different concepts and consequently provides more abstraction and modularity. Furthermore it describes the computation a step at a time, and so allows to express very fine properties about the computation. Some other works about the mechanical verification of language properties such as [12, 5] use also a reduction semantics.

The formal development illustrated in this paper is done within the Coq proof assistant [6] and consequently can be seen as another formal piece that supplements the certification laid out in [5, 4] even if we consider here a monomorphic type system.

As far as we are aware, among the different computer-verified proofs (e.g. [5, 4, 3, 12, 9, 15, 11, 13]), no publication mentions the computer-verified formalization and proof of type soundness for a fragment of ML mixing imperative and functional primitives. In [14], VanInwegen presents a formalization of Core Standard ML within HOL and by the way incorporates references. She has proved the type preservation property (it means that when the evaluation of a well-typed program terminates, it computes a value of the same type than the type of the program) for a good portion of the language. However this work is done with a big step semantics (based on an evaluation semantic relation).

Beyond this important and useful purpose, our work is the starting-point of the development of 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. Such a project requires at the same time the construction of formal components, a methodology and tools for composing and re-using formal pieces. At the present time, in order to reach this long term objective, we compare the formal development of the type soundness for Mini-ML, a language without imperative aspects and the formal development of the same property for Reference-ML, the same language with imperative features. Along this paper, we'll emphasize what have to be added in the proof or re-defined.

We assume here familiarity with the Calculus of Inductive Constructions. We use version 6.3 of the Coq proof assistant. In order to make this paper more readable, we adopt 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.iie.cnam.fr/~boite/monomorphic_soundness.tgz`.

The next section introduces a generic structure used to represent type environment and store. Section 3 describes the language. The semantics is exposed and formalized in the fourth section. Then, we present the typing in section 5. In the last section, we deal with the type soundness.

2 A Generic Table

Static and dynamic semantics use structures to store information, for instance type environments that map free identifiers to types, or memories that map locations to values. So we present in this section the formalization of a generic structure `table`, which has two implicit parameters A and B, where A is the type of the keys, and B the type of the stored values. We define it as an inductive type with only one constructor `intro_table` : its parameters are the domain of the table represented by a list of elements of type A, and a function from A to B. The type A is required to verify the decidability of the equality (predicate *eq_A_dec*).

```
Inductive table : Set :=
 intro_table : (list A) → (A → B) → table.
```

An object of type `(table A B)` is in fact a function whose domain is explicitly given, which allows us to use it as a partial function, or an association table. We define the operations of application, domain computing, adding of a new information in the table. We see a lot of advantages compared to a list of pairs : adding a new association doesn't create any redundancy, the computation of the domain and the application are got for "free", as we can see below.

```
Inductive apply_table : (table A B)→ A → B → Prop :=
 intro_apply_table : ∀l:A,∀dom:(list A), ∀f:A→B,
   l∈dom→(apply_table (intro_table dom f) l (f l)).
```

The predicate `(apply_table t a b)` stands for two properties : $a \in dom(t)$ and $t(a) = b$, so the partiality of the function isn't a problem. The operation of adding is less simple, as we avoid redundancy.

```
Definition add_table [a:A; b:B; t:(table A B)] : table :=
 Cases t of (intro_table dom f) ⇒ Cases a∈dom of
  _ ⇒(intro_table dom
   (λa':A.(Cases (eq_A_dec a a') of _ ⇒ b | _ ⇒ (f a')))
| _⇒(intro_table (Cons a dom)
   (λa':A.(Cases (eq_A_dec a a') of _ ⇒ b | _ ⇒ (f a')))
```

The concrete definition of a table is never used in the formalization. We manipulate a table as an abstract data via several lemmas. The two most important of them follow, and are used to define a equivalence relation between tables.

```
Lemma swap_table :  ∀A,B:Set,∀a,a',e:A,∀b,b',r:B,∀t:table,
  ~a=a' → (apply_table (add_table a' b' (add_table a b t)) e r)→
    (apply_table (add_table a b (add_table a' b' t)) e r).

Lemma add_add_table :  ∀A,B:Set,∀t:table,∀a,e:A,∀b,b',r:B,
  (apply_table (add_table a b t) e r) →
   (apply_table (add_table a b (add_table a b' t)) e r).
```

The total Coq script for this structure with its lemmas overtakes 1000 lines.

3 The Language Reference-ML

The first step to study a language is to formally define its syntax. However, it is not independent from the semantics. Actually, the semantics imposes to define two notions in addition to the language: memory locations and values. We first detail the syntax of the language. Then, in the second subsection, we discuss semantic extensions, and lastly we give the Coq translation.

3.1 Expressions

We consider the following grammar of the expressions, in which we emphasize in a bold font the constructions of Reference-ML added to Mini-ML :

e ::= c	constant	| ***e* ; *e***	**sequence**
| x	identifier	| **while *e* do *e***	**loop**
| $e\ e$	application	| **ref *e***	**reference creation**
| fun x e	abstraction	| **!*e***	**look-up**
| let x=e in e	local definition	| ***e*:=*e***	**assignment**
| if e then e else e	conditional		
c ::= n	integer		
| b	boolean		
| ***unit***	**side-effect result**		

So Reference-ML is an extended λ-calculus which incorporates references and constructions to manipulate them. A reference is a memory cell containing a value. Its contains may be modified all along the program by using assignments.

3.2 Semantic Extensions

We've made the choice to work with a reduction semantics, also called Small-Step semantics. It focuses on each elementary step calculus, by rewriting expressions. With references, we have to take care of the memory state, which can change during the reduction process. For instance, the evaluation of an assignment $e_1 := e_2$ needs to evaluate e_1 to compute the location that is the address of the cell, whose contents in the memory has to be modified, and needs to evaluate e_2 to know which value to put in the memory. So, we have to explicitly manipulate locations in the reduction steps. The notion of location doesn't belong to the user's abstract syntax, but as we need it in the reduction process, we've introduced it in the Coq definition of the language.

Among the expressions, we have to distinguish the values. This notion is syntactic : a value is either a constant, either an abstraction (because the reduction strategy we use is a head reduction), or a location.

3.3 Coq Formalization

In order to name the identifiers and the locations, we define two abstract types, respectively `identifier` and `location`. We assume the decidability of equality on both types, and we assume that one can always find a fresh location *wrt* a memory, that is to say one can exhibit a location l so that l denotes an address that does not appear in the memory.

The language specification in Coq is obviously an inductive type :

```
Inductive Constant : Set :=
   Const_int : nat  → Constant
 | Const_bool: bool → Constant
 | unit      : Constant.

Inductive expr : Set :=
   Const : Constant → expr
 | Var   : identifier → expr
 | Loc   : location → expr
 | Fun   : identifier → expr → expr
 | Apply : expr → expr → expr
 | Let_in: identifier → expr → expr → expr
 | If    : expr → expr → expr → expr
 | While : expr → expr → expr
 | Ref   : expr → expr
 | Deref : expr → expr
 | Sequ  : expr → expr → expr
 | Assign: expr → expr → expr.
```

We define an object of type `value` as a pair composed of an expression e and a proof that e is a value.

```
Inductive is_value : expr → Prop :=
  Cst_val : ∀c:Constant,(is_value (Const c))
| Loc_val : ∀l:location,(is_value (Loc l))
| Fun_val : ∀i:identifier, ∀e:expr,(is_value (Fun i e)).

Inductive value : Set :=
 value_intro : ∀e:expr,(is_value e)→value.
```

3.4 From Mini-ML to Reference ML

The extension of the language consists simply in adding new constructors in the inductive types `expr` and `Constant` : `Loc, Sequ, While, Ref, Deref, Assign`, and `unit.` So going from Mini-ML to Reference-ML produces supertypes (in the sense of object-oriented programming) : any Mini-ML expression is also a Reference-ML expression.

The predicate `is_value` is extended with the `Loc_val` clause, but the definition of the type `value` is unchanged.

4 Reduction Semantics

4.1 Elementary Reduction and Reduction Strategy

As a reduction semantics takes care of elementary steps of the computation, it's straightforward to explain the memory evolution during the reduction. A memory state is a finite mapping between locations and values. We require three operations on a memory m :

- $dom(m)$ which computes the set of locations in the memory.
- $m(l)$ which gives the value mapped to l in m.
- $m \oplus (l, v)$ which maps l to v, and maps l' to $m(l')$ if $l' \in dom(m)$ and $l' \neq l$.

A configuration is an (expression/memory state)-pair, and the reduction relation is a transition relation between configurations. A reduction of the expression a_1 in the memory state m_1, into a_2 in the memory state m_2 will be noted $a_1/m_1 \longrightarrow a_2/m_2$.

To define the reduction strategy, we've adopted the technique which consists in defining an elementary reduction, also called ϵ-reduction, and the notion of reduction context, as proposed in [16]. We define the ϵ-reduction noted $a_1/m_1 \longrightarrow_\epsilon a_2/m_2$ in Fig.1. In all the rules, v denotes a value. The two first rules corresponds to β-reductions, and $a[v/x]$ denotes the substitution of x by v in a.

$$
\begin{aligned}
(fun\ x\ a)\ v\ /\ m &\longrightarrow_\epsilon a[v/x]\ /\ m \\
let\ x = v\ in\ a\ /\ m &\longrightarrow_\epsilon a[v/x]\ /\ m \\
if\ true\ then\ e_1\ else\ e_2\ /\ m &\longrightarrow_\epsilon e_1\ /\ m \\
if\ false\ then\ e_1\ else\ e_2\ /\ m &\longrightarrow_\epsilon e_2\ /\ m \\
while\ e_1\ do\ e_2\ /\ m &\longrightarrow_\epsilon if\ e_1\ then\ (e_2\ ;\ while\ e_1\ do\ e_2)\ else\ unit\ /\ m \\
ref\ v\ /\ m &\longrightarrow_\epsilon l\ /\ m \oplus (l, v) \quad \text{if } l \notin dom(m) \\
l := v\ /\ m &\longrightarrow_\epsilon unit\ /\ m \oplus (l, v) \quad \text{if } l \in dom(m) \\
!l\ /m &\longrightarrow_\epsilon m(l)\ /\ m \quad \text{if } l \in dom(m) \\
v; e/\ m &\longrightarrow_\epsilon e/\ m
\end{aligned}
$$

Fig. 1. ϵ-reduction rules

A reduction context E can be seen as an expression with a unique hole, and is defined as follows :

$$
\begin{aligned}
E ::= {} & \bullet \mid E\ e \mid v\ E \mid let\ x = E\ in\ e \mid if\ E\ then\ e\ else\ e \\
& \mid \mathbf{E; e} \mid \mathbf{ref\ E} \mid \mathbf{E := e} \mid \mathbf{!E} \mid \mathbf{l := E}
\end{aligned}
$$

where $\bullet$ is a hole, e an expression, v a value, and l a location. $E[e]$ denotes E where the hole has been filled with e.

The relation $\longrightarrow$ is defined as the smallest relation containing $\longrightarrow_\epsilon$ and satisfying the rule CONTEXT given below that allows to reduce in a context :

$$
\frac{a_1/m_1 \longrightarrow a_2/m_2}{E[a_1]/m_1 \longrightarrow E[a_2]/m_2}\ \text{(CONTEXT)}
$$

4.2 Substitution

The β-reductions in Fig.1 make appear substitutions of an identifier by an expression in another one. The substitution operation must ensure not to substitute bound variables, neither to capture variables. Modelling the operation of substitution by a function would require to rename some bound variables to prevent captures, so we have represented this operation in Coq with a predicate `(subst_expr e x e' e'')` which means $e'' = e[e'\backslash x]$. This predicate is defined by case analysis on e, and it uses the predicate `free_ident` to know if an identifier is free or not in a given expression. This last one is also defined by induction on the expression. A fragment of the code is given below :

```
Inductive free_ident : identifier → expr → Prop :=
 free_var    : ∀x:identifier,(free_ident x (Var x))
|free_apply1 : ∀x:identifier,∀e1,e2:expr,(free_ident x e1)→
                (free_ident x (Apply e1 e2))
|free_apply2 : ∀x:identifier,∀e1,e2:expr,(free_ident x e2)→
                (free_ident x (Apply e1 e2))
...
```

```
Inductive subst_expr : expr→identifier→expr→expr→Prop :=
 subst_const: ∀x:identifier,∀e':expr,∀c:Constant
              (subst_expr (Const c) x e' (Const c))
|subst_var  : ∀x:identifier,∀e':expr
              (subst_expr (Var x) x e' e')
|subst_nvar : ∀x,y:identifier,∀e':expr, ¬x=y →
              (subst_expr (Var y) x e' (Var y))
| subst_fun  : ∀x:identifier,∀e,e':expr,
              (subst_expr (Fun x e) x e' (Fun x e))
...
```

4.3 Formalization of State Memory

A memory is an association table from keys of type *location* to objects of type *value*. So, to represent a memory, we use the generic structure *table* defined in section 2.

```
Definition memory := (table location value).
```

4.4 The Reduction Relation in Coq

We follow the informal presentation given in section 4.1. So, we begin to define $\longrightarrow_\epsilon$ as the inductive predicate `red_epsilon`.

```
Definition configuration := expr*memory.

Inductive red_epsilon : configuration → configuration → Prop :=
  beta1 : ∀x:identifier,∀e,er,v:expr,∀m:memory,
          (is_value v)→ (subst_expr e x v er)→
          (red_epsilon ((Apply (Fun x e) v),m) (er,m))
| beta2 : ∀x:identifier,∀e,er,v:expr,∀m:memory,
          (is_value v)→ (subst_expr e x v er)→
          (red_epsilon ((Let_in x v e),m) (er,m))
| red_ref:∀m:memory,∀v:expr,∀l:location,∀V:(is_value v))
         ¬(in_dom l m)→
         (red_epsilon ((Ref v),m)
                      ((Loc l),(add_table l (value_intro v V) m)))
...
```

When we want to reduce e in the context E, we have to reduce the expression E in which the hole has been replaced by e. It means we need a mechanism to fill a hole. It's interesting and practical to represent a context by a function `expr → expr`. In that case, we benefit from the functionality of Coq, and filling a hole is simply applying a function. However all the functions of type `expr → expr` are not valid contexts. We precise the set of valid contexts with the help of the inductive predicate `is_context`. This technique was previously introduced by one of the authors in [5].

```
Definition context := expr → expr.

Inductive is_context : context → Prop :=
  hole      : (is_context (λx:expr. x))
| app_left  : (e:expr)(E:context) (is_context E) →
              (is_context (λx:expr. (Apply (E x) e)))
| app_right : (v:expr)(E:context) (is_context E) → (is_value v)→
              (is_context (λx:expr. (Apply v (E x))))
...
```

Now, the reduction relation $\longrightarrow$ in Coq is simply :

```
Inductive red : configuration → configuration → Prop :=
  epsilon  : ∀c1,c2:configuration,
           (red_epsilon c1 c2)→(red c1 c2)
| cont   : ∀c1,c2:configuration,∀E:context, (red c1 c2) →
           (is_context E)→
           (red (app_ctx E c1) (app_ctx E c2)).
```

where `app_ctx` is a function which applies a context E on a configuration c, that is computes $E[c]$.

```
Definition app_ctx :=
 λE:context.λc:configuration.
  Cases c of (e,m) ⇒ ((E e),m) end.
```

4.5 From Mini-ML to Reference-ML

The relation `red` rewrites a configuration into another configuration. In Mini-ML, a configuration is an expression, whereas in Reference-ML, a configuration is a (expression/memory state)-pair. By re-defining `configuration` and the function `app_ctx`, the relation `red` is binary in both cases, and its definition has exactly the same formulation in both languages.

The relation `red_epsilon` in Reference-ML can be considered as a conservative extension of the Mini-ML one : five rules are added for the imperative features, and the old rules are the same (except the supplementary parameter, a memory state, that doesn't change in these rules).

The predicates `is_context`, `free_ident`, and `subst_expr` are simply extended with the new cases.

5 Typing

5.1 The Types

The type algebra of our language is defined as follows :

$$\tau ::= \textit{basic types} \mid \alpha \mid \tau \rightarrow \tau \mid \boldsymbol{\tau}\ \mathbf{Ref}$$

The basic types are Int, Bool, and **Unit**. The symbol α denotes a type variable. The type $\tau \rightarrow \tau$ denotes the functional type, and $\tau\ Ref$ is the type of a reference that contains a value of type τ. Here again, the constructors required for Reference-ML are printed in the bold font.

We assume a set *stamp* for the type variables, and the translation within Coq introduces the simple following inductive types :

```
Inductive Basic : Set := Int : Basic | Bool : Basic | Unit : Basic.

Inductive type  : Set :=
   Const_t: Basic → type
 | Var_t: stamp → type
 | Arrow: type  → type → type
 | Ref_t: type  → type.
```

5.2 The Typing Rules

We define the typing rules, with a set of inference rules, given in a natural deduction style [7], between judgments $\Gamma \vdash e : \tau$ meaning that in the type environment Γ, the expression e has the type τ. The function *type_of_constant* relates a constant with its type. The typing rules in Fig.2 define the well-typed expressions of the language. The rules in the right side of the figure are specific to Reference-ML. We also have to give the typing rule (Loc) for locations: they can appear during the reduction process and thus, it is necessary to be able to type expressions that contain locations.

$$\frac{}{\Gamma \vdash c : type\_of\_constant(c)}\ (\textsc{Cst}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash ref\ e : \tau\ Ref}\ \textbf{(Ref)}$$

$$\frac{\Gamma(x)=\tau}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : \tau\ Ref}{\Gamma \vdash !e : \tau}\ \textbf{(Deref)}$$

$$\frac{\Gamma \oplus (x:\tau) \vdash e : \tau'}{\Gamma \vdash fun\ x\ e : \tau \rightarrow \tau'}\ (\textsc{Fun}) \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ ;\ e_2 : \tau}\ \textbf{(Seq)}$$

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}\ (\textsc{App}) \qquad \frac{\Gamma \vdash e_1 : \tau\ Ref \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : Unit}\ \textbf{(Assign)}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \oplus (x:\tau) \vdash e' : \tau'}{\Gamma \vdash let\ x=e\ in\ e' : \tau'}\ (\textsc{Let}) \qquad \frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash while\ e_1\ do\ e_2 : Unit}\ \textbf{(While)}$$

$$\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}\ (\textsc{If}) \qquad \frac{\Gamma(l)=\tau}{\Gamma \vdash l : \tau\ Ref}\ \textbf{(Loc)}$$

Fig. 2. Typing rules

In this context, type environments contain not only type information about free identifiers, but also type information about locations. A type environment may be considered as a pair (Γ_{id},Γ_{loc}) :

$$\Gamma_{id} ::= \emptyset \mid \Gamma_{id} \oplus (x : \tau) \quad \Gamma_{loc} ::= \emptyset \mid \Gamma_{loc} \oplus (l : \tau)$$

made up of a part Γ_{id} relative to identifiers, and a part Γ_{loc} relative to locations. Each part of the pair is an association table, and is specified in Coq by the generic structure *table* with the necessary parameters. This decomposition in the implementation makes easier the treatment.

So Γ_{id}, Γ_{loc}, and Γ are respectively specified by

```
Definition typ_env_ident := (table identifier type).
Definition typ_env_loc   := (table location   type).
Definition typ_env       := typ_env_ident*typ_env_loc.
```

We require two operations on type environments : the updating $\oplus$ and the application, using the corresponding operations on tables :

$$\begin{aligned}(\Gamma_{id}, \Gamma_{loc}) \oplus (k : \tau) &= (\Gamma_{id} \oplus (k : \tau), \Gamma_{loc}) \text{ if } k \text{ is an identifier} \\ &= (\Gamma_{id}, \Gamma_{loc} \oplus (k : \tau)) \text{ if } k \text{ is a location} \\ (\Gamma_{id}, \Gamma_{loc})(k) &= \Gamma_{id}(k) \text{ if } k \text{ is an identifier} \\ &= \Gamma_{loc}(k) \text{ if } k \text{ is a location}\end{aligned}$$

To define the typing rules in Coq, we use an inductive predicate that contains a constructor per typing rule :

```
Inductive type_of  : typ_env → expr → type → Prop :=
 type_of_const : ∀ Γ:typ_env,∀c:Constant
      (type_of Γ (Const c) (type_of_constant c)))
|type_of_var: ∀ Γ:typ_env_ident,∀x:identifier,∀ τ:type
      Γ(x)=τ → (type_of Γ (Var x) τ)
...
```

5.3 Configuration Types

Nothing ensures for a given location that the type of the value associated in the memory, and the type of this location in the type environment are the same. So, we define the notion of well-typed memory in an environment : the domain of the memory `m` and the domain of the locations part of the type environment Γ must be the same, and for each location l of this domain, if the type of l is τ Ref in the environment, then the value of l in the memory has the type τ in the environment. The Coq definition `memory_respects_env` follows this informal specification. To denote this relation informally, we write $\Gamma \vdash m$.

```
Definition memory_respects_env [Γ:typ_env;m:memory] : Prop :=
 Cases Γ of (Γ_id,Γ_loc) ⇒
```

```
(∀l:location,∀τ:type,v:expr,(apply_table m l v) →
 (apply_table Γ_loc l τ) → (type_of Γ (value2expr v) τ))
∧ dom(m)=dom(Γ_loc)
```

A configuration a/m is well-typed in an environment Γ, if a is well-typed in Γ and if m respects Γ. We define the typing of a configuration in Γ as the conjunction of the typing of the expression and the well-typing of the memory, both in Γ.

```
Definition type_of_config  [Γ:typ_env;c:configuration;τ:type] :=
 Cases c of (e,m) ⇒
  (type_of Γ e τ) ∧ (memory_respects_env Γ m)
```

5.4 From Mini-ML to Reference ML

The type algebra is extended with the type for references. Again we define a supertype. For the constructors common to Mini-ML and Reference-ML, the typing rules remains identical up to the re-definition of the operations $\oplus$ and application on environments. The typing relation we want for Reference-ML has to type configurations - not only expressions. It means the memory has to respect the type environment, so we add this condition to the type relation as a conjunction.

6 Type Soundness

Among the syntactically correct programs, we can distinguish three groups, those whose evaluation :

- terminates on a value
- indefinitely loops
- blocks (*i.e.* can't be reduced anymore but is not a value)

The role of the typing is to limit the set of syntactically correct programs. We describe in this section the proof of the safety (or the soundness) of our type system. It means that the evaluation of a well-typed program, if it terminates, is a value. In other words, no type errors can appear during the computation, and the typing eliminates blocking-programs.

We follow the structure of the paper-and-pencil proof given in [8], adapted to the monomorphic case.

6.1 The *less typable* Relation

We define an ordering between configurations as follows :

a_1/m_1 is less typable than a_2/m_2, written $a_1/m_1 \sqsubseteq a_2/m_2$, if for all environment Γ and all type τ, there exists an extension Γ' of Γ so that

$$(\Gamma \vdash a_1 : \tau \wedge \Gamma \vdash m_1) \Longrightarrow (\Gamma' \vdash a_2 : \tau \wedge \Gamma' \vdash m_2)$$

For instance, $let\ x = (ref\ 1)\ in\ !x\ /\ \emptyset\ \sqsubseteq\ !l\ /\ (l:1)$.

Here is the Coq specification of $\sqsubseteq$:

```
Definition less_typable [c1,c2:configuration]: Prop :=
 ∀Γ:typ_env,∀τ:type,
  (type_of_config Γ c1 τ) →
   (∃Γ':typ_env, (extend Γ' Γ) ∧ (type_of_config Γ' c2 τ)).
```

The extension of an environment consists in adding type information about new locations. So, Γ' extends Γ, if $\Gamma'{=}\Gamma \oplus (l_1 : \tau_1) \oplus \ldots \oplus (l_n : \tau_n)$, with $l_i \notin dom(\Gamma)$. We denote it $\Gamma' \gg \Gamma$. For the Coq specification, we require the inclusion of the locations parts of the environments, and the equality on the identifier parts.

```
Definition extend [Γ',Γ:typ_env] : Prop :=
Cases Γ' Γ of
 (Γid',Γloc') (Γid,Γloc) ⇒ Γid'=Γid →
         ∀l:location,∀τ:type,(l ∈ Γloc)→
         (apply_table Γloc' l τ)→(apply_table Γloc l τ)
```

6.2 Subject Reduction

This theorem means the reduction preserves the type, and it ensures the computation won't create any type error.

Theorem 1 (Subject Reduction).
If $a_1/m_1 \longrightarrow a_2/m_2$, *then* $a_1/m_1 \sqsubseteq a_2/m_2$

To prove this theorem, we use the type preservation by ϵ-reduction, and the growing of $\sqsubseteq$ by context application. Then the proof of Subject Reduction holds in three Coq lines :

```
Theorem Subject_reduction : ∀c1,c2:configuration,
        (red c1 c2) → (less_typable c1 c2).
Induction 1;Intros.
Apply type_preservation_by_red_epsilon; Assumption.
Apply less_typable_grows; Assumption.
```

Proposition 1 (Type preservation by epsilon-reduction).
if $a_1/m_1 \longrightarrow \epsilon\ a_2/m_2$, *then* $a_1/m_1 \sqsubseteq a_2/m_2$

```
Lemma type_preservation_by_red_epsilon : ∀c1,c2:configuration,
  (red_epsilon c1 c2) → (less_typable c1 c2).
```

The proof is done by case analysis on the ϵ-reduction rule. In each case, we give the extended environment which allows to type the configuration $c2$. In the β-reduction cases, we use the following lemma :

Lemma 1 (Substitution lemma).
if $\Gamma \vdash e : \tau$ *and* $\Gamma \oplus (i,\tau) \vdash e_1 : \tau_1$ *then* $\Gamma \vdash e_1[i\backslash e] : \tau_1$

translated into Coq by :

```
Lemma substitution_lemma:
 ∀e1,e2,e:expr,∀i:identifier,∀Γ:typ_env,∀τ,τ1:type,
 (subst_expr e1 i e e2)→
   (type_of Γ ⊕ (i : τ) e1 τ1) →
     (type_of Γ e τ) →
       (type_of Γ e2 τ1).
```

This lemma is proved by induction on the expression e_1. The proof is easy, but requires to show that if an expression is well-typed in an environment then it is well-typed in an environment equivalent to the first one and furthermore the type is the same.

Proposition 2 (Growing of $\sqsubseteq$).
if $a_1/m_1 \sqsubseteq a_2/m_2$, *then* $E[a_1]/m_1 \sqsubseteq E[a_2]/m_2$

```
Lemma less_typable_grows :∀E:context,∀c1,c2:configuration,
         (is_context E) → (less_typable c1 c2) →
    (less_typable (app_ctx E c1) (app_ctx E c2)).
```

The proof is a structural induction on the predicate *is_context* defining E. For each case, we give the extended environment Γ' of Γ, given by the hypothesis (*less_typable* $c1$ $c2$), to prove $\Gamma' \vdash E[e2] : \tau$ with the condition $\Gamma \vdash E[e1] : \tau$.

6.3 Normal Form Theorem

The normal form theorem establishes that an expression in normal form, well-typed in an environment which types no identifier, is a value. We allow to have locations in the typing environment, because a location is a value, and to type a location we need the type information in the typing environment.

If an expression is not in normal form, it means it can be reduced. That is to say, there exists a configuration that the relation `red` can reach. It is specified in Coq with :

```
Definition is_reducible [c:configuration] :=
 (∃c':configuration. (red c c')).

Definition config_is_value [c:configuration] :=
 Cases c of (e,m) ⇒ (is_value e) end.

Theorem nf_typed_are_value:∀Γ_loc:typ_env_loc,∀τ:type,∀c:configuration,
   (type_of_config (∅,Γ_loc) c τ) → ¬(is_reducible c) →
     (config_is_value c).
```

The normal form theorem is a consequence of the progression lemma :

Lemma 2 (Progression).
If an environment types only locations, if a/m is well-typed in this environment, and if a is not a value, then a can be reduced.

```
Lemma progression : ∀Γ_loc:typ_env_loc,∀c:configuration,∀τ:type,
 (type_of_config (∅,Γ_loc) c τ) → ¬(config_is_value c) →
  (is_reducible c).
```

We prove it by induction on a where $c = a/m$. When a is not a value, we explicitly give the expression in which a can reduce. In the case of a β-redex, the expression in which it reduces makes appear an expression resulting from a substitution. We have to prove that this expression really exists (because the substitution is not a total function). It is expressed by the following lemma, established by induction on the expression e.

```
Lemma substitute_succeeds : ∀e,e1:expr,i:identifier,∀τ:type,
 ∀Γ_loc:typ_env_loc, (type_of (∅,Γ_loc) e1 τ) →
  (is_value e1) → (∃ e':expr. (subst_expr e i e1 e')).
```

6.4 Strong Type Soundness

The theorem *Subject Reduction* ensures types are preserved across a reduction step. The theorem of the normal form ensures a well-typed program in the initial environment, which can't reduce anymore, is a value. As a consequence of this two previous theorems, we obtain the type soundness : the evaluation of a term of type τ doesn't block, but furthermore if it terminates, we obtain a value of type τ. It's the *strong* version of the type soundness.

Formally, we have to define the reflexive and transitive closure of the reduction relation `red_star` :

```
Inductive red_star  : configuration → configuration → Prop:=
  red_0 : ∀sn:configuration,(red_star sn sn)
| red_n : ∀sn,s0,s':configuration, (red s0 s')→(red_star s' sn)→
            (red_star s0 sn).
```

The type soundness is specified as follows :

```
Theorem type_safe : ∀a:expr,∀τ:type,∀c':configuration,
 (red_star (a,∅) c') → (type_of_config (∅,∅) (a,∅) τ) →
 ¬(is_reductible c') →
  (config_is_value c') ∧ (∃env:typ_env (type_of_config env s' t))
```

This theorem is proved by induction on the length of the reduction. In the case corresponding to `red_0`, we apply the normal form theorem. In the induction case `red_n`, the property follows from the induction hypothesis and the Subject Reduction theorem.

6.5 From Mini-ML to Reference ML

In Mini-ML, $a_1 \sqsubseteq a_2$ means if a_1 has the type τ in an environment Γ, then a_2 has the type τ in Γ. The relation $\sqsubseteq$ of Reference-ML is an extension of that of Mini-ML, if we re-define `type_of` by `type_of_config` and if we accept to extend the type environment to type the second configuration. It is a conservative extension, because the extension of an environment concerns the location part, and there isn't location part in Mini-ML.

The proofs of Type preservation by ϵ-reduction, substitution lemma, growing of $\sqsubseteq$, progression and `substitute_succeeds`, `nf_typed_are_values` are inductive proofs. All the proof cases we had in Mini-ML are quite the same in Reference-ML, and of course, new cases are treated.

The proof of *Subject Reduction*, as it uses the previous propositions - is exactly the same in both languages. Our final theorem, `type-safe`, concludes `(config_is_value c')`. It is the same theorem as in Mini-ML if we re-define `config_is_value`, and the proof (by induction) has the same cases as Mini-ML plus its own cases.

7 Conclusion

This work has two interests. The first is that, as far as we know, it is the first machine-ckecking of the type-soundness of an ML-like language with references in a reduction semantics. The second is that it focuses on the impact on the definitions and the proofs when the language is extended. We can find in many papers where a language is extended "the proof is similar to the previous case ...". We wanted to quantify this fact. Our experience is that the machine-checking verifies this, if we take care to extend and re-define the good notions. In our study, our Coq script rises from 2100 to 3700 lines (both including 1000 lines for the package on the generic table). The paper tries to trace the different supertypes, extensions and redefinitions that the new features introduce.

In a future work, we'll extend Reference-ML in another way, with polymorphism, or object for example. Paper-pencil proofs exist for many years. All have proved that some notions are exactly the same, others are extended, others are re-defined. Indeed, our final aim is to define a formal method to specify the extension L_{i+1} of a language L_i, and to use the proofs of the properties of L_i to show those of L_{i+1}. This method wouldn't be valid in a non-conservative extension. Several works [1], [2], [10], deal with proof reuse in the context of inductive types, it would be very interesting to try to merge them with our concerns and develop ad hoc tools.

Acknowledgements We thank Véronique Viguié Donzeau-Gouge for the useful discussions we had with her.

References

[1] G. Barthe and O. Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*,

volume 2030 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 2001.

[2] G. Barthe and F. van Raamsdonk. Constructor Subtyping in the Calculus of Inductive Constructions. In J. Tuyrin, editor, *Proceedings of FOSSACS'00*, volume 1784 of *Lecture Notes in Computer Science*, pages 17–34. Springer-Verlag, 2000.

[3] Ana Bove. A Machine Assisted Proof that Well-Typed Expressions Cannot Go Wrong. *Technical report, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.*, May 1998.

[4] V. Ménissier-Morain C. Dubois. Certification of a Type Inference Tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning, Vol 23, nos 3-4, 319-346*, 1999.

[5] C. Dubois. Proving ML Type Soundness Within Coq. *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000, Lecture Notes in Computer Science*, 1869:Springer–Verlag, 126–144, 2000.

[6] B. Barras et al. *The Coq Proof Assistant, Reference Manual (v6.3.1).* INRIA Rocquencourt, 2000.

[7] G. Kahn. Natural Semantics. *In Lecture Notes in Computer Science*, 247:22–39. Springer, 1987.

[8] Xavier Leroy. Typage et Programmation, DEA course notes - http://pauillac.inria.fr/~xleroy.

[9] Dieter Nazareth and Tobias 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.

[10] Erik Poll. Subtyping and Inheritance for Inductive Types. *In Proceedings of TYPES'97 Workshop on Subtyping, inheritance and modular development of proofs*, September 1997.

[11] F. Pfenning S. Michaylov. 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.

[12] Don Syme. Proving Java Type Sound. *In Jim Alves-Foss, editor, Formal Syntax and Semantics of Java, volume 1523 of LNCS. Springer*, 1999.

[13] D. Terrasse. Vers un Environnement d'Aide au Développement de Preuves en Sémantique Naturelle. *Thèse de Doctorat, Ecole Nationale des Ponts et Chaussées*, 1995.

[14] Myra VanInwegen. Towards Type Preservation in Core SML. *Technical report, Cambridge University*, 1997.

[15] T. Nipkow W. Naraschewski. Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning, Vol 23, nos 3-4, 299-318*, 1999.

[16] Wright and Felleisen. A Syntactic Approach To Type Soundness. *Information and computation*, 115(1):38–94, 1994.