Sûreté du Typage de ML : Spécification et Preuve en Coq

Catherine Dubois

LaMI, Université d'Évry Val d'Essonne
Bd. des Coquibus – F-91025 Évry Cedex FRANCE

dubois@lami.univ-evry.fr

Résumé

Nous spécifions en Coq la syntaxe abstraite, la sémantique statique et la sémantique dynamique d'un langage à la Mini-ML. Nous formalisons une version *déterministe* du système de types que nous prouvons équivalente à la définition initiale de Damas et Milner. Nous donnons ensuite une preuve formelle mécanisée de la sûreté du typage par rapport à l'évaluation. Ce travail complète la certification de l'algorithme d'inférence de types pour ML réalisée en Coq par l'auteur et V. Ménissier-Morain.

1. Introduction

Le travail présenté dans cet article est complémentaire à la certification exposée dans [6, 5] dont l'objectif était de prouver la correction et la complétude de l'algorithme d'inférence de types pour ML par rapport aux règles de typage. Cependant cette certification ne faisait aucune liaison avec la sémantique dynamique de ML.

L'ensemble du travail présenté dans cet article et dans [6, 5] constitue une certification formelle complète des aspects liés au typage de ML. En effet, nous fournissons de cette façon une spécification de la sémantique dynamique, de la sémantique statique de ML et de l'algorithme d'inférence de ML appelé traditionnellement W. Nous prouvons la sûreté du typage par rapport à la sémantique dynamique et nous prouvons que l'inférence de types est correcte et complète par rapport aux règles de typage, plus précisément que W calcule pour toute expression bien typée son type principal.

Au delà du défi que représente la mécanisation de ces preuves classiques, la motivation est de disposer d'une spécification formelle

et vérifiée mécaniquement. Cette vérification, du fait qu'elle comporte la preuve de propriétés non triviales du langage établies à l'aide d'un démonstrateur de théorèmes, apporte non seulement une plus grande confiance en la sémantique formelle du langage mais aussi assure une meilleure *maintenabilité*. En effet l'outil de preuve aide à cerner l'impact des modifications de sémantique sur les propriétés déjà établies du langage. On peut aussi examiner ce travail sous l'angle de la réutilisation, notre spécification formelle constitue alors une base pour envisager des variantes du système de types, nombreuses dans la littérature autour de ML.

A notre connaissance, aucune publication ne mentionne une certification formelle mécanisée d'un tel ensemble. Naraschewski et Nipkow [11] ont certifié W avec le démonstrateur Isabelle/HOL, l'approche (menée indépendamment et simultanément) est assez similaire à la nôtre mais ne fait aucune liaison avec la sémantique dynamique de ML. D'autres auteurs, comme Michaylov et Pfenning [12], Frost [7] , plus récemment VanInwegen pour Core-SML [19] ont traité la sûreté du typage par rapport à la sémantique dynamique mais sans liaison avec W. Les travaux [13, 17] concernent la sûreté du typage d'un sous-ensemble du langage Java respectivement avec les systèmes de preuve Isabelle/HOL et DECLARE.

Nous avons choisi le système Coq (version 6.1) pour réaliser notre certification. En effet, cet assistant de preuve est bien adapté pour prouver des propriétés sur les langages de programmation pour diverses raisons. Par exemple, la syntaxe abstraite du langage est parfaitement décrite par un type inductif et Coq fournit des tactiques spécialisées pour manipuler les définitions inductives. Une rapide présentation du système Coq est faite dans la section 2.

Dans cet article, tout d'abord (section 3), nous formalisons en Coq les termes du langage retenu, très proche de Mini-ML [3]. Puis nous présentons la modélisation de sa sémantique dynamique (section 4) et de son système de types (section 5) avec, entre autres, la théorie sous-jacente des substitutions. Notons que la spécification de la syntaxe abstraite et du système de types est commune avec la certification de W dans [6], par conséquent les choix faits pour la certification de W, en particulier le parti de rester proche d'une implantation fonctionnelle de W, se répercutent ici. Cependant bon nombre d'entre eux s'imposent également ici. Nous avons choisi d'utiliser une version déterministe du système de types, similaire à celle donnée par Kahn et al. dans [3]. Cependant ce n'est pas le système de types initialement proposé par Milner [4], la partie 6 est une preuve formelle en Coq de l'équivalence de ces deux systèmes de types. Ensuite, dans la section 7, nous prouvons une propriété intrinsèque du système de types, à savoir la conservation du typage par

substitution, c'est aussi une propriété utilisée à maintes reprises dans notre contexte. La preuve de sûreté du typage est développée dans la partie 8. Plus précisément, nous établissons la propriété de sûreté forte : l'évaluation d'une expression de type τ, si elle termine, produit une valeur qui, sémantiquement, appartient bien au type τ. Enfin, nous étudions brièvement les approches prises dans les autres travaux relatifs à ce sujet. Nota Bene : les parties 5 et 7 de cet article reformulent certaines parties de l'article co-écrit par l'auteur et Valérie Ménissier-Morain.

Ce papier ne retient des preuves que quelques aspects en donnant en général les lemmes intermédiaires les plus fondamentaux et parfois un script de preuve sommaire. Cependant la formalisation complète est accessible via `http://pauillac.inria.fr/~menissier`.

2. L'assistant de preuve Coq

Nous présentons ici brièvement le système interactif de preuve Coq (voir [1] pour une description détaillée). Les éléments syntaxiques nécessaires à la compréhension des phrases Coq seront fournis tout au long du texte.

Le système Coq permet le développement de preuves formelles vérifiées. Les axiomatisations et les spécifications sont écrites dans le langage logique Gallina fondé sur le Calcul des Constructions Inductives [16], λ-calcul typé où les types sont eux-mêmes des termes typés du langage et qui permet des définitions inductives. Les types inductifs sont proches des types concrets à la ML et les relations inductives sont comparables à des prédicats Prolog. De la définition d'une construction inductive, le système Coq engendre automatiquement le principe d'induction associé et fournit les outils de preuve pour les manipuler, comme par exemple les tactiques `Induction` et `Inversion`.

L'utilisateur peut définir des fonctions éventuellement récursives : Coq fournit des facilités syntaxiques (les constructions `Fixpoint` et `Recursive`) pour décrire des fonctions mettant en oeuvre une récursion structurelle. Coq offre également la construction `Cases` proche du filtrage à la ML.

Enfin, Coq permet l'extraction de programme à partir du terme de preuve associée à une spécification [15]. L'approche inverse est possible : une tactique spécialisée, la tactique `Program` [14], appliquée à un programme écrit dans un style à la ML et à une spécification, permet de prouver que le programme est correct par rapport à la spécification. Ces deux derniers aspects n'ont pas été utilisés dans le travail présenté ici.

3. Le langage

Dans cette partie, nous présentons la syntaxe abstraite du langage considéré, représentatif du noyau fonctionnel de ML.

Les expressions du langage sont les constantes entières, les identificateurs (`x`), les abstractions (λ`x.e`), les applications (`e e'`), les expressions `let` (`let x=e in e'`) et enfin les fonctions récursives (`Rec f x.e`).

La récursion est présentée dans notre formalisme comme une extension de l'abstraction ; ceci montre explicitement que les seules expressions récursives de notre langage sont les fonctions.

On pourra trouver une présentation élégante d'un langage similaire avec les preuves associées, dans le premier chapitre de la thèse de Leroy [10].

En Coq la définition des termes du langage se fait au moyen du type inductif `expr` :

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

Cette définition est très proche du type concret ML correspondant :

```
type expr =
  Const_int  of nat
| Variable of ident
| Lambda of ident * expr
| Rec of ident * ident * expr
| Apply of expr * expr
| Let_in of ident * expr * expr;;
```

Remarque : l'ajout au langage des constructions standards telles que les paires, les expressions conditionnelles, les listes ... ne relève d'aucune difficulté dans l'ensemble de ce travail.

4. La sémantique dynamique

La sémantique dynamique donne un sens aux expressions du langage en décrivant l'évaluation de chacune d'elles. Nous choisissons de la définir dans le style de la Sémantique Naturelle [9], comme un ensemble de règles

d'inférence. Seuls les programmes qui terminent sans fournir un résultat d'erreur y sont considérés. Auparavant, il nous faut préciser les valeurs que peuvent prendre les expressions.

4.1. Valeurs sémantiques et contextes d'évaluation

Le domaine des valeurs sémantiques est l'ensemble des valeurs que peuvent prendre les expressions. Ici on retient comme seules valeurs possibles les nombres et les valeurs fonctionnelles appelées aussi fermetures.

La valeur d'une expression e dépend des valeurs des identificateurs libres de e. Ainsi on évalue une expression e relativement à un contexte d'évaluation (raccourci parfois en contexte), noté Δ dans la suite, qui associe à toute variable libre de e sa valeur. Il peut être considéré comme une liste de couples *(identificateur, valeur)*.

Les notions de valeur et de contexte sont des notions mutuellement récursives. En effet, une fermeture est constituée d'une expression fonctionnelle et d'un contexte. Nous distinguons 2 sortes de fermetures : les fermetures simples associées aux fonctions non récursives (de la forme $<< \lambda\, x.e, \Delta >>$) et les fermetures récursives associées aux fonctions récursives (de la forme $<<_{@\,\texttt{Rec}}\ f\ x.e, \Delta >>$). Boutin propose également cette distinction dans son travail concernant la certification du compilateur ML [2]. On aurait pu également retenir la notion de fermeture opaque : une fermeture opaque est une fermeture dont on ne peut pas inspecter le contenu, c'est la valeur associée à une opération pré-définie. La prise en compte de cette notion n'ajoute et ne retire rien à la généralité de l'approche prise ici.

En Coq, on spécifie ces 2 notions par 2 types mutuellement inductifs `val` et `ctx`

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

Les opérations fondamentales sur les contextes sont :

- la recherche de la valeur associée à un identificateur donné : on écrira $\Delta(x)$ ou (`assoc_ident_in_ctx` Δ x) en Coq. La fonction `assoc_ident_in_ctx` est une fonction partielle, elle est écrite en

Coq en simulant le mécanisme des exceptions : on définit pour cela le type inductif `ident_in_ctx` qui comprend 2 constructeurs, `Ident_not_in_ctx` qui marque l'échec et `Ident_in_ctx` qui introduit une valeur de type `val`.

```
Inductive ident_in_ctx: Set :=
  Ident_not_in_ctx: ident_in_ctx
| Ident_in_ctx: val -> ident_in_ctx.
```

– et l'ajout d'un couple (x, v) dans un contexte Δ. Le contexte résultant, noté $\Delta \oplus x : v$, garde inchangées les informations trouvées dans Δ, exceptée celle relative à x. La représentation linéaire des contextes permet une implantation très simple de l'opération $\oplus$: un *cons*. Combinée à l'implantation de $\Delta(x)$ qui retourne la valeur associée à la *première* occurrence de x dans Δ, la représentation naïve pour $\oplus$ garantit le respect des règles de visibilité.

4.2. Les règles d'inférence de la sémantique dynamique

Nous commençons par présenter les règles d'inférence dans le style de la Sémantique Naturelle (voir la figure 1). Le séquent $\Delta \vdash e : v$ signifie que l'expression e s'évalue en la valeur v relativement au contexte Δ. La sémantique met en oeuvre une sémantique de l'appel par valeur. Elle comprend deux règles concernant l'application : (APP1) s'applique lorsque l'expression gauche s'évalue en une fonction simple, (APP2) s'applique lorsque l'expression gauche s'évalue en une fonction récursive.

4.3. Formalisation en Coq

Les règles de sémantique naturelle de la figure 1 se traduisent en Coq en le prédicat inductif `val_of`. La correspondance est quasi syntaxique : on trouve un constructeur par règle d'inférence. On pourra consulter [18] pour un cadre plus général de traduction de la Sémantique Naturelle en Coq.

```
Inductive val_of : ctx -> expr -> val->Prop:=
 Val_of_num: (n: nat)(c: ctx)(val_of c (Const_nat n) (Num n))
|Val_of_ident: (c: ctx)(i: ident)(v: val)
  (assoc_ident_in_ctx i c) = (Ident_in_ctx v) ->
    (val_of c (Variable i) v)
|Val_of_lambda: (c: ctx)(i: ident)(e : expr)
  (val_of c (Lambda i e) (Clos i e c))
|Val_of_rec: (c: ctx)(f, i: ident)(e: expr)
```

$$(\textsc{cst}) \quad \Delta \vdash n : n$$

$$(\textsc{id}) \quad \Delta \vdash x : \Delta(x)$$

$$(\textsc{abs}) \quad \Delta \vdash \lambda\ x.e : << \lambda\ x.e, \Delta >>$$

$$(\textsc{rec}) \quad \Delta \vdash \texttt{Rec}\ f\ x.e : <<_{@}\ \texttt{Rec}\ f\ x.e, \Delta >>$$

$$(\textsc{app1}) \quad \frac{\Delta \vdash e : << \lambda\ x.e_f, \Delta_f >> , \quad \Delta \vdash e' : v, \quad \Delta_f \oplus x : v \vdash e_f : v'}{\Delta \vdash e\ e' : v'}$$

$$(\textsc{app2}) \quad \frac{\Delta \vdash e : <<_{@}\ \texttt{Rec}\ f\ x.e_f, \Delta_f >> , \quad \Delta \vdash e' : v, \quad \Delta_f \oplus x : v \oplus f : <<_{@}\ \texttt{Rec}\ f\ x.e_f, \Delta_f >> \vdash e_f : v'}{\Delta \vdash e\ e' : v'}$$

$$(\textsc{let}) \quad \frac{\Delta \vdash e : v, \quad \Delta \oplus x : v \vdash e' : v'}{\Delta \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : v'}$$

FIG. 1 – *La sémantique dynamique*

```
  (val_of c (Rec f i e) (Rec_clos f i e c))
|Val_of_apply:
  (c, c1: ctx)(e1,e2, e : expr)(i: ident)(u,v: val)
  (val_of c e1 (Clos i e c1)) ->
    (val_of c e2 u)->
       (val_of (Ccons i u c1) e v)->
         (val_of c (Apply e1 e2) v)
|Val_of_apply_rec:
  (c, c1: ctx)(e1, e2, e : expr)(f, i: ident)(u, v: val)
  (val_of c e1 (Rec_clos f i e c1)) ->
    (val_of c e2 u)->
      (val_of (Ccons f (Rec_clos f i e c1) (Ccons i u c1)) e v)->
        (val_of c (Apply e1 e2) v)
|Val_of_let: (c: ctx)(i: ident)(e1, e2 : expr)(u, v: val)
  (val_of c e1 u)->
    (val_of (Ccons i u c) e2 v)->
      (val_of c (Let_in i e1 e2) v).
```

A ce point on peut établir une propriété intéressante (prouvée également dans [2]) mais non nécessaire pour la sûreté du typage : le prédicat `val_of` est une fonction partielle injective ou encore la sémantique dynamique de ML est déterministe.

```
Lemma DS_is_deterministic: (c: ctx)(e: expr)(v: val)
   (val_of c e v) ->
      (v': val) (val_of c e v') -> v=v'.
```

La preuve se fait par induction sur `(val_of c e v)` et utilise largement l'inversion que nous illustrons ci-dessous sur deux exemples.

• Supposons le but suivant :

```
c,c0 : ctx    v, v' : val   n : nat
H0 : (val_of c0 (Const_nat n) v')
============================
 (Num n)=v'
```

La commande `Inversion H0` déduit de la définition du prédicat inductif `val_of` que la seule valeur possible pour `v'` est la constante entière `(Const_nat n)`. La tactique introduit dans le contexte des hypothèses l'égalité `(Num n)=v'` et récrit le but en tenant compte de cette égalité. Celui-ci devient :

```
c,c0 : ctx    v, v' : val   n : nat
H0 : (val_of c0 (Const_nat n) v')
H2 : (Num n)=v'
============================
 (Num n)=(Num n)
```

• Supposons maintenant que l'on dispose de l'hypothèse `H1 : (val_of c (Apply e0 e1) v)`. En examinant la définition du prédicat `val_of` (ou les règles d'inférence correspondantes), on constate qu'il existe deux façons de dériver `H1` (on peut utiliser (APP1) ou (APP2)). On va alors prouver le but initial `B` dans chacun des cas. Ainsi, l'inversion de `H1` engendre les 2 sous-buts suivants (le premier correspond au cas où `e0` s'évalue en une fonction simple, le deuxième au cas où `e0` s'évalue en une fonction récursive) :

```
....
H2 : (val_of c e0 (Clos i e4 c1))
H3 : (val_of c e1 u)
H4 : (val_of (Ccons i u c1) e4 v)
============================
    B1

....
H2 : (val_of c e0 (Rec_clos f i e4 c1))
H3 : (val_of c e1 u)
H4 : (val_of (Ccons f (Rec_clos f i e4 c1) (Ccons i u c1)) e4 v)
============================
    B2
```

5. Le système de types

Le système de types associe un type à chaque expression du programme. Les informations de type relatives aux identificateurs sont maintenues dans un environnement. Une fonction polymorphe, c'est-à-dire une fonction qui peut s'appliquer sur des objets de différents types, ne peut être introduite que par l'utilisation de la construction `let` . Son type contient donc des variables quantifiées. Il est nécessaire de conserver les informations concernant les variables quantifiées dans l'environnement afin de vérifier la légalité des différentes applications de la fonction polymorphe. La notion de type quantifié correspond intuitivement à la notion de *schéma de type* que nous développons ci-dessous. Nous formalisons ensuite la notion d'environnement, la généralisation, seule opération permettant de découvrir de nouveaux schémas de type. Avant d'aborder les règles de typage, nous spécifions les notions de substitution et instance.

5.1. Types et schémas de type

Les types considérés sont :

– le type de base *int*

- les variables de type α, β ...
- et les types fonctionnels $\tau \to \tau'$ (où τ et τ' sont des types).

A cette définition correspond l'introduction du type inductif `type` en Coq :

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

Par abus de langage, on peut confondre ici le type `stamp` avec le type `nat` des entiers naturels.

Un schéma de type σ est défini comme un type quantifié universellement par un ensemble fini (éventuellement vide) de variables de types : $\forall\alpha_1 \ldots \alpha_n.\tau$ (où τ est une expression de type). On appelle alors $\alpha_1 \ldots \alpha_n$ les variables génériques du schéma de type. Un schéma de type peut contenir des variables libres : par exemple dans le schéma $\forall\alpha.\alpha \to \beta$, la variable α est générique mais β est libre.

Un schéma de type sans variable générique est dit *trivial* et noté $\forall.\tau$.

Nous avons retenu une formalisation en Coq qui permet de distinguer syntaxiquement dans l'expression de type les variables génériques des autres. Par conséquent, le type `type_scheme` est défini inductivement et comprend 2 constructeurs différents pour les variables, `Gen_var` pour les variables génériques et `Var_ts` pour les autres.

```
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.
```

Conformément à cette définition, le schéma de type $\forall\alpha.\alpha \to \beta$ est représenté par le terme Coq `(Arrow_ts (Gen_var alpha) (Var_ts beta))` où `alpha` et `beta` correspondent respectivement à α et β.

5.2. Environnement

Nous avons mentionné plus haut qu'un environnement peut être considéré comme une liste de paires *(identificateur, schéma de type)*. Cette vue est retenue en Coq, le type `environment` est donc défini par `list (ident * type_scheme)`. Structurellement, la notion d'environnement est très proche de celle de contexte d'évaluation. On utilise pour les environnements les mêmes notations que pour les contextes : $\Gamma(x), \Gamma \oplus y : \sigma$.

5.3. Généralisation de type

Seule la construction `let` introduit dans l'environnement des identificateurs avec des types polymorphes, c'est-à-dire des schémas de type non triviaux, via l'opération de généralisation `gen_type` qui construit un schéma de type à partir d'un type τ et d'un environnement Γ : cette opération rend génériques les variables de τ qui n'apparaissent pas libres dans Γ.

$$\texttt{gen\_type}\ \tau\ \Gamma = \forall\alpha_1 \ldots \alpha_n.\tau \text{ avec } \alpha_i \in (\texttt{FV\_type}\ \tau) - (\texttt{FV\_env}\ \Gamma)$$

Cette définition suggère une implantation récursive par cas sur le type τ, naturelle, simple et facile à manipuler dans le cadre de la preuve. Malheureusement, le couperet de l'alpha-conversion projette son ombre ! En effet, si α et β dénotent 2 variables distinctes non libres dans l'environnement considéré, les types $\alpha \to \alpha$ et $\beta \to \beta$ produisent lors de la généralisation les schémas de type respectifs $\forall\alpha.\alpha \to \alpha$ et $\forall\beta.\beta \to \beta$ qui, quoique équivalents au sens où ils définissent le même ensemble de types, sont représentés en Coq par 2 termes syntaxiquement différents. Certes, ni le système de types, ni même l'algorithme d'inférence W n'ont à décider de l'équivalence de 2 schémas de type, mais la preuve de la conservation du typage par substitution en a besoin. Plus précisément elle teste l'équivalence de 2 schémas de type produits par généralisation. Trois solutions sont alors possibles :

- gérer explicitement l'alpha-conversion : on pourrait alors adopter l'implantation naturelle de la généralisation, mais l'ensemble de la spécification serait alors pollué. Dans le même esprit, on pourrait aussi envisager une syntaxe pour les schémas de type qui incorporerait la notion d'alpha-conversion comme dans les travaux de Gordon et Melham en HOL [8] (nous n'avons pas du tout exploré cette voie).
- implanter `gen_type` de manière à ce que 2 schémas de type équivalents soient *syntaxiquement identiques* : c'est la solution que nous avons retenue
- ou enfin utiliser la relation d'ordre $\succ$ entre schémas de type (définie plus loin) : cette solution élégante permet elle aussi de garder la formalisation simple de la généralisation, elle est proposée par Naraschewski et Nipkow dans leur preuve de W [11].

Implantation de la généralisation

Nous proposons un codage linéaire pour les schémas de type produits par généralisation : toute occurrence de la variable générique α est

écrite `(Gen n)`, si α est la nième variable découverte au cours de la généralisation. Par exemple, la généralisation de $(\alpha \to \beta) \to (\beta \to \alpha)$ par rapport à un environnement vide serait `((Gen 0) ->(Gen 1)) ->((Gen 1) ->(Gen 0))`. De même (`gen_type` $\alpha\ \emptyset$) = `(Gen 0)` = (`gen_type` $\beta\ \emptyset$).

La définition de `gen_type` utilise une fonction auxiliaire récursive `gen_type_aux`: `(gen_type_aux t env l)` calcule le couple `(ts, l')` où `ts` est la généralisation de `t` par rapport à l'environnement `env` quand `l` contient les variables déjà découvertes comme génériques et `l'` la liste `l` à la fin de laquelle on a ajouté les nouvelles variables génériques produites par le calcul.
L'index `k` attribué à la variable `v` de `t` non libre dans `env` est aussi la position de `v` dans `l'`. L'implantation fait échouer la fonction `index` lors de la recherche d'une variable non présente.

```
Fixpoint gen_type_aux [t: type]: environment -> (list stamp) ->
type_scheme*(list stamp):= [env:environment] [l: (list stamp)]
  Cases t of
    Nat => (Nat_ts,l)
  | (Var v) =>
    (if_ type_scheme*(list stamp) (in_list_stamp v (FV_env env))
          ((Var_ts v), l)
          (Cases (index l v) of
             Stamp_not_in =>
                  ((Gen_var (Stamp (length stamp l))),
                   (app stamp l (cons stamp v (nil stamp))))
           | (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 stamp))).
```

Remarque: Une telle définition pour `gen_type` complique certains lemmes. En effet, les lemmes relatifs à cette opération sont souvent doublés d'un lemme de même nature portant sur `gen_type_aux` généralement peu intuitif. La découverte de ces lemmes intermédiaires s'apparente souvent à la découverte des invariants dans les traitements itératifs.

5.4. Substitutions et Instances

La notion d'instance est définie de manière usuelle relativement à la notion de substitution. On distingue principalement deux catégories de substitutions :

- les substitutions qui agissent sur les variables libres d'un type, d'un schéma de type et d'un environnement.
- les substitutions génériques qui agissent uniquement sur les variables génériques d'un schéma de type.

Dans la suite, nous abordons la spécification en Coq de ces notions.

5.4.1. Instance de type et substitution

Un type τ' est une *instance* d'un type τ si il existe une substitution s qui transforme τ en τ', i.e. telle que $s\tau = \tau'$.

Nous spécifions une substitution comme une liste de couples *(variable de type, expression à substituer)*.
Les opérations sur les substitutions sont : la recherche du type associé à une variable donnée (`assoc_stamp_in_subst`), l'application d'une substitution à un type, un schéma de type ou un environnement, la composition de substitutions (`compose_subst`), le domaine, le co-domaine d'une substitution .... Ces opérations sont écrites en Coq dans un style fonctionnel très proche de leur implantation en ML par exemple. Ci-dessous nous détaillons deux d'entre elles : l'application d'une substitution sur un schéma de type `apply_subst_type_scheme` et la composition `compose_subst`.

```
Recursive Definition apply_subst_type_scheme [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_type_scheme s ts1)
                                   (apply_subst_type_scheme s ts2)).
```

Notation: `Definition id [x: t]: t':= e` définit une fonction nommée `id` dont le paramètre est `x` de type `t` et dont le corps est l'expression `e` déclarée de type `t'`. Ici le corps de `apply_subst_type_scheme` est une fonction qui réalise un filtrage sur son argument.

Appliquer une substitution `s` sur un schéma de type `ts` consiste à remplacer uniquement les variables libres apparaissant dans `ts` et `s` par l'expression correspondante dans `s` : une conversion est parfois nécessaire, elle est assurée par la fonction `type_to_type_scheme`.

```
Definition compose_subst:= [s1, s2: substitution]
  (app stamp*type (subst_diff s2 s1) (apply_subst_list s1 s2)).
```

Intuitivement, la substitution `(compose_subst s1 s2)` (`s1` puis `s2`) est calculée en appliquant `s2` sur chacun des types de `s1` : `(apply_subst_list s1 s2)`. Reste ensuite à ajouter les paires dont la variable est dans le domaine de `s2` mais pas dans le domaine de `s1`.

Ces définitions sont accompagnées de nombreux lemmes, par exemple ceux relatifs à la composition de 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)).
```

Le lemme `composition_of_substitutions_stamp` établit que l'opération de composition réalise bien ce que son nom indique : appliquer la substitution dénotée par `(compose_subst s1 s2)` sur la variable `st` produit le même terme que celui obtenu en appliquant `s2` au résultat de l'application de `s1` sur `st`. La preuve de ce lemme réclame un effort important et compte environ 600 lignes.

Un dernier exemple de lemmes relatifs aux substitutions :

```
Lemma subset_FV_type: (t:type)(s: substitution)(alpha: stamp)
 (in_list_stamp alpha (FV_type t))=true ->
    (is_subset_list_stamp (FV_type ( s (Var alpha)))
                          (FV_type ( s t))).
```

Ce lemme établit le lien entre les variables d'un type `t` (aussi appelées ses variables libres et calculées par la fonction `FV_type`) et celles de `(s t)` :

$$\alpha \in FV(t) \Rightarrow FV(s\ \alpha) \subset FV(s\ t)$$

Remarque: on peut également proposer de représenter une substitution par une abstraction Coq et profiter ainsi de ses capacités d'ordre supérieur.

```
Definition substitution := stamp -> type.
```

Composer deux substitutions devient alors un jeu d'enfant ! Naraschewski et Nipkow [11] ont choisi cette représentation dans leur certification de W sans avoir à ajouter aucune condition de finitude sur les substitutions contrairement à ce que nous anticipions au commencement de ce travail [6].

5.4.2. Instance générique et substitution générique

Nous avons retenu une représentation spécifique pour les substitutions génériques, sans référence aucune aux noms des variables auxquelles la substitution s'applique. Une substitution générique est, en Coq, un vecteur de types.

```
Definition gen_substitution := (list type).
```

Toute substitution générique se manipule en respectant la règle suivante : *le type à substituer à la* n*ième variable générique est le* n*ième élément de la liste substitution.* Pour être plus juste, il faudrait écrire : le type à substituer à la variable générique `(Gen n)` est le nième élément de la substitution. C'est bien sûr la seconde règle qui est implantée, mais on peut la confondre avec la première puisque la majorité des schémas de type manipulés au cours du typage est produite par l'opération de généralisation (qui engendre des schémas codés linéairement).

Il ressort de cette règle que la longueur d'une substitution générique s_g appliquée à un schéma σ doit être *au moins* le nombre de variables génériques du schéma (en fait, le plus grand des entiers dénotant les variables génériques). Par conséquent la fonction `apply_subst_gen` qui applique une substitution générique sur un schéma de type en vue de produire un type est partiellement définie. On la formalise en Coq en simulant le mécanisme des exceptions :

```
Fixpoint apply_subst_gen [s: gen_substitution; ts: type_scheme]:
subst_gen_answer:=
  Cases ts of
    Nat_ts => (Some_subst_gen Nat)
  | (Var_ts v) => (Some_subst_gen (Var v))
  | (Gen_var (Stamp x)) =>
       Cases (nth x s) of
         Type_not_in => Error_subst_gen
       | (Nth_in t) => (Some_subst_gen t)
       end
  | (Arrow_ts ts1 ts2) =>
       Cases (apply_subst_gen s ts1) of
         Error_subst_gen => Error_subst_gen
       | (Some_subst_gen t1) =>
           Cases (apply_subst_gen s ts2) of
            Error_subst_gen => Error_subst_gen
           |(Some_subst_gen t2) => (Some_subst_gen (Arrow t1 t2))
           end
       end
```

Un type τ' est une *instance générique* d'un schéma de type $\forall\alpha_1 \ldots \alpha_n.\tau$ si et seulement si il existe une substitution s_g pour $\alpha_1 \ldots \alpha_n$

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

$$(\text{ID})\quad \frac{\Gamma(x) = \sigma, \quad \tau \text{ instance générique de } \sigma}{\Gamma \vdash x : \tau}$$

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

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

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

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

FIG. 2 – *Les règles de typage du système DM'*

telle que $s_g(\tau) = \tau'$. Par exemple, le type $int \rightarrow \beta$ est une instance générique du schéma $\forall\alpha.\alpha \rightarrow \beta$.

Le prédicat `is_gen_instance` spécifie qu'un type est instance générique d'un schéma donné :

```
Definition is_gen_instance:= [t: type] [ts: type_scheme]
 {sg: gen_substitution|(apply_subst_gen sg ts)=(Some_subst_gen t)}.
```

Notation Coq : l'expression Coq {`x:t|P`} se lit *il existe x de type t tel que P soit satisfait.*

5.5. Les règles de typage

Les règles de typage sont décrites, elles aussi, dans le style de la Sémantique Naturelle (voir la figure 2). Le séquent $\Gamma \vdash e : \tau$ signifie que l'expression e a le type τ relativement à l'environnement Γ. Nous avons choisi d'utiliser la présentation de ces règles dite dirigée par la syntaxe [3], notée dans la suite DM' et non la version initiale de Damas Milner,

appelée DM. En effet, cette présentation est plus proche de l'algorithme d'inférence W, en outre elle rend notre preuve plus aisée. Ainsi seule la syntaxe de l'expression détermine l'unique règle applicable, tandis que les prémisses des règles ne sont relatives qu'aux sous-termes de leur sujet. L'induction structurelle est alors un outil puissant bien adapté à la manipulation des règles de typage. Enfin, les 2 versions sont équivalentes : de nombreux travaux mentionnent cette équivalence, comme par exemple [3, 11] mais n'en fournissent pas de preuve. Dans la partie suivante, nous prouvons formellement cette équivalence.

La spécification en Coq des règles de typage de DM' se fait via l'introduction du prédicat inductif `type_of` de type `environment ->expr ->type ->Prop`. La traduction est immédiate.

```
Inductive type_of: environment -> expr -> type -> Prop :=
  type_int_const: (env: environment)(n: nat)
    (type_of env (Const_nat n) Nat)
| type_var: (env: environment)(x: ident)(t: type)(ts: type_scheme)
    (assoc_ident_in_env x env)=(Ident_in_env ts) ->
        (is_gen_instance t ts) ->
           (type_of env (Variable x) t)
| type_lambda: (env: environment)(x: ident)(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'))
| type_rec_fun: (env: environment)(e: expr)
                  (f, x: ident)(t, t': type)
    (type_of (add_env (add_env env x (type_to_type_scheme t))
                f (type_to_type_scheme (Arrow t t'))) e t') ->
         (type_of env (Rec f x e) (Arrow t t'))
| type_app: (env: environment) (e, e': expr) (t, t': type)
    (type_of env e  (Arrow t t') ) -> (type_of env e' t )  ->
        (type_of env (Apply e e') t')
| type_let_in: (env: environment)(e, e': expr)
                 (x: ident)(t, t': type)
    (type_of env e t) ->
       (type_of (add_env env x (gen_type t env)) e' t') ->
          (type_of env (Let_in x e e') t').
```

6. Équivalence entre DM et DM'

6.1. Formalisation du système DM de Damas-Milner

En ce qui concerne le système non déterministe DM de Damas-Milner, nous suivons la définition de ce système en Sémantique Naturelle donnée par Kahn et al. dans [3]. Cependant, nous avons adapté cette définition de manière à utiliser les notions introduites dans le reste de

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

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

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

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

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

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

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

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

FIG. 3 – *Le système DM*

notre approche (voir la figure 3). Tout séquent de typage dans DM prend la forme $\Gamma \ \vdash_{DM} e : \ \sigma$, il contient un schéma de type et non plus un type comme dans DM'. Les règles (CST), (ABS), (REC), (APP) sont identiques dans les 2 systèmes : dans DM celles-ci ne manipulent que des schémas de type triviaux, assimilables à des types. La règle relative aux identificateurs (TAUT) se contente d'extraire l'information de type de l'environnement. La règle concernant l'expression `let` de DM ne mentionne aucune généralisation explicite ; en revanche la règle (GEN) peut introduire de réels quantificateurs. La règle (INST) utilise la relation d'ordre sur les schémas de type `more_general` encore notée $\succ$. Cette relation est requise par la preuve de la complétude de W, nous utilisons donc ici la théorie relative à cette notion. Nous en rappelons la définition

ci-dessous et donnons sa spécification en Coq :

Un schéma de type σ_1 est dit *plus général* qu'un schéma de type σ_2 si et seulement si toute instance générique de σ_2 est aussi une instance générique de σ_1. On écrit $\sigma_1 \succ \sigma_2$ ou en Coq (`more_general` σ_1 σ_2). Par exemple, $\forall\alpha\beta.\alpha \to \beta \succ \forall\alpha.\alpha \to \alpha$.

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

Le prédicat inductif `type_of_DM`, dont on donne un extrait ci-dessous, spécifie le système DM en Coq. La correspondance avec les règles d'inférence est encore ici directe, à l'exception de la clause `type_of_DM_gen` relative à la règle (GEN) : elle quantifie une liste de variables non libres dans l'environnement et non pas une seule variable. La fonction Coq `bind_list` réalise cette dernière opération. Sa définition est très similaire à celle de la fonction `gen_type` : elle introduit un codage linéaire de même nature.

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

6.2. Correction de DM' par rapport à DM

Nous prouvons la correction de DM' par rapport à DM en montrant que si on peut dériver dans le système DM' que e a le type τ relativement

à Γ alors on peut déduire dans DM que e est lié au schéma de type $\forall.\tau$.

```
Lemma sound: (e: expr)(env: environment)(t: type)
  (type_of env e t) ->
     (type_of_DM env e (type_to_type_scheme t)).
```

La preuve se fait par induction sur e. La plupart des cas se résout facilement en utilisant la règle correspondante de DM et par application des hypothèses d'induction. Le cas où e est un identificateur demande l'application successive des règles d'inférence (TAUT) et (INST). Le cas du `let` réclame plus d'effort, il s'agit alors de récrire

```
        (bind_list (Snd (gen_type_aux t env (nil stamp)))
                   (type_to_type_scheme t))
```

en

```
        (gen_type t env).
```

Intuitivement, ce jeu d'écriture signifie que quantifier (d'un seul coup) toutes les variables de `t` non libres dans `env` dans le schéma trivial $\forall$.`t`, c'est exactement généraliser `t` par rapport à `env`. Il est question ici d'égalité syntaxique grâce au codage linéaire introduit dans `bind_list` et `gen_type`. La preuve de ce lemme passe par la preuve d'un lemme intermédiaire à propos des fonctions auxiliaires définissant `bind_list` et `gen_type`.

6.3. Complétude de DM' par rapport à DM

Le théorème de complétude s'énonce de la façon suivante: si on dispose d'une preuve dans DM que e a le schéma de type σ alors on peut montrer dans DM' que e a un type τ dont la généralisation par rapport à Γ produit un schéma de type plus général que σ.

```
Lemma complete: (env: environment)(e: expr)(ts: type_scheme)
  (type_of_DM env e ts) ->
    {t : type | (type_of env e t) /\
                (more_general (gen_type t env) ts)}.
```

La preuve se fait par induction sur `(type_of_DM env e ts)` et repose sur de nombreux lemmes relatifs à la relation `more_general`. Par exemple:

```
(more_general ts1 ts2) -> (more_general (s ts1) (s ts2))

(are_disjoints l (FV_env env)) ->
    (more_general (gen_type t env) ts) ->
        (more_general (gen_type t env) (bind_list l ts))
```

```
(more_general (bind l ts) ts)

(more_general (gen t env) t)
```

Les preuves de ces lemmes se révèlent lourdes et calculatoires. En effet, pour la plupart, elles demandent d'exhiber des substitutions génériques.

7. Conservation du typage par substitution

La conservation du typage par substitution correspond à une propriété classique des règles de typage qui établit que si $\Gamma \vdash e : \tau$ est satisfait alors pour toute substitution s, le séquent de typage $s\Gamma \vdash e : s\tau$ est encore satisfait. Cela signifie que si τ est un type possible pour e relativement à Γ, on peut obtenir un autre type pour e en appliquant une substitution sur Γ et τ.

```
Lemma typing_is_stable_by_substitution:
(e: expr) (t: type) (env: environment) (s: substitution)
  (type_of env e t) ->
     (type_of (apply_subst_env env s) e (apply_substitution s t)).
```

Le preuve du lemme `typing_is_stable_by_substitution` ne pose pas de problème particulier dans la cas monomorphe mais devient difficile dans le cas polymorphe, principalement à cause du procédé de généralisation des types. Elle demande d'importants développements sur les substitutions de renommage par exemple. Cette preuve est faite par induction sur l'expression typée e, la plupart des cas se résout en appliquant l'hypothèse d'induction. Au contraire, les cas `ident` et `let` demandent un plus gros effort. La preuve de ces 2 cas est détaillée dans [6]. Seuls les lemmes intermédiaires principaux sont mentionnés ici, nous indiquons les difficultés.

Le premier lemme que nous détaillons établit que la propriété d'être une instance générique est conservée par substitution :

```
Lemma is_gen_instance_stable_by_substitution:
(ts: type_scheme) (s: substitution) (t: type)
   (is_gen_instance t ts) ->
     (is_gen_instance (apply_subst_type s t)
                      (apply_subst_type_scheme s ts)).
```

A cet endroit, il est demandé de montrer comment les opérations `apply_subst_type` et `apply_subst_gen` commutent: appliquer la substitution `s` sur l'instance générique `s_g ts` donne le même type que celui obtenu en appliquant la substitution générique `(map_apply_subst_type sg s)` sur le schéma de type `s ts`.

```
Lemma subst_gen_subst_type:
(s: substitution)(ts: type_scheme)(t: type)(sg: gen_substitution)
   (apply_subst_gen sg ts) = (Some_subst_gen t) ->
     (apply_subst_gen (map_apply_subst_type sg s)
                      (apply_subst_type_scheme s ts))
     =(Some_subst_gen (apply_subst_type s t)).
```

La principale difficulté du cas `let` vient de ce que *la généralisation et la substitution ne commutent que sous certaines conditions.* Plus précisément, appliquer une substitution `phi` sur le schéma de type obtenu en généralisant le type `t` par rapport à l'environnement `env` produira le même schéma de type que celui obtenu en généralisant `(phi t)` par rapport à `(phi env)` uniquement si `phi` n'est pas concerné par les variables qui sont précisément quantifiées durant la généralisation de `t` par rapport à l'environnement `env` (i.e. `(gen_vars t env)`).

```
Lemma gen_in_subst_env:
(env: environment)(phi: substitution)(t: type)
  (are_disjoints (FV_subst phi) (gen_vars t env)) ->
    (apply_subst_type_scheme phi (gen_type t env)) =
    (gen_type (apply_subst_type phi t) (apply_subst_env env phi)).
```

Au cours de la preuve, on sera donc amené à construire une substitution qui permettra la permutation avec la généralisation. Toute cette partie passe par la définition des substitutions dites de renommage qui sont des substitutions injectives à valeurs dans l'ensemble des variables de type, dont les domaines et co-domaines sont disjoints. La propriété essentielle ici est que l'opération qui consiste à renommer les variables à quantifier au cours d'une généralisation est transparente pour celle-ci :

```
Lemma gen_renaming: (env: environment)(rho: ren_substitution)
(t: type)(s: substitution)
    (is_rename_subst rho) ->
    (domain_of_rename_subst rho)=(gen_vars t env) ->
    (are_disjoints (range_of_rename_subst rho)) (FV_env env) ->
    (are_disjoints (range_of_rename_subst rho) (FV_subst s)) ->
      (gen_type t env)=
      (gen_type (apply_subst_type (rename_to_subst rho) t) env).
```

8. Sûreté du système de types

L'objectif principal lié à la conception d'un système de types pour un langage est de sélectionner parmi les programmes syntaxiquement valides ceux qui ne provoqueront pas d'erreur de type à l'exécution. Jusqu'à présent, nous avons défini les règles de typage indépendamment de la sémantique dynamique. Nous allons donc maintenant prouver

avec Coq que l'objectif est atteint, c'est-à-dire que les règles de typage sont correctes par rapport aux règles d'évaluation : nous commençons par rapprocher les notions de valeur et type, puis nous établissons la sûreté du typage, plus précisément le théorème traditionnellement appelé *Subject Reduction Theorem.*

8.1. Lien entre types et valeurs

Le lien entre les notions de type et de valeur passe par la définition de 3 relations sémantiques implantées en Coq par des prédicats mutuellement inductifs : la relation `type_of_val` entre valeurs et types, la relation `ctx_env_match` entre environnements et contextes et la relation `sem_gen` entre valeurs et schémas de type.

Le prédicat `(type_of_val v t)` indique que *la valeur* `v` *a le type* `t` ; il est défini par cas sur la valeur `v` :
• une valeur entière a le type `Int`
• une fermeture $<< \lambda\ x.e, \Delta >>$ a un type fonctionnel, c'est le type qui permet de prouver que la fonction $\lambda\ x.e$ est bien typée relativement à un environnement `env` concordant avec Δ (l'utilisation des règles de typage dans ce cadre est due à Tofte). Idem pour la fermeture récursive.

Une valeur `v` *a le schéma de type* σ `(sem_gen v` σ`)` si on peut lui attribuer tous les types instances de σ :

```
(t: type) (is_gen_instance t σ) -> (type_of_val v t).
```

Enfin, par extension, un contexte Δ et un environnement Γ *concordent* (`ctx_env_match` $\Delta\ \Gamma$)), si et seulement si ils ont le même domaine de définition et si pour tout identificateur i de leur domaine, $\Delta(i)$, la valeur associée à i dans Δ a le schéma de type $\Gamma(i)$: (`sem_gen` $\Delta(i)\ \Gamma(i)$).

```
Mutual Inductive  type_of_val: val -> type -> Prop :=
   type_num: (n: nat) (type_of_val (Num n) Nat)
|type_closure:
 (i: ident)(e: expr)(c: ctx)(env: environment)(t1, t2 : type)
  (ctx_env_match c env) ->
   (type_of env (Lambda i e) (Arrow t1 t2)) ->
    (type_of_val (Clos i e c) (Arrow t1 t2))
|type_rec_closure:
 (f,i: ident)(e: expr)(c: ctx)(env: environment)(t1, t2: type)
 (ctx_env_match c env) ->
  (type_of env (Rec f i e) (Arrow t1 t2)) ->
   (type_of_val (Rec_clos f i e c) (Arrow t1 t2))

with  ctx_env_match: ctx -> environment -> Prop :=
 match_nil: (ctx_env_match Cnil (nil ident*type_scheme))
```

```
|match_cons:
 (c: ctx)(env: environment)(i: ident)(ts: type_scheme)(v: val)
  (sem_gen v ts) -> (ctx_env_match c env) ->
   (ctx_env_match (Ccons i v c)
                  (cons ident*type_scheme (i,ts) env))

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

Remarque : la définition en Coq ajoute l'hypothèse que les identificateurs figurent dans le même ordre dans le contexte et l'environnement. Celle-ci simplifie l'écriture et la preuve mais n'est nullement restrictive.

8.2. Sûreté du système de types : le théorème dit *subject reduction*

On cherche à montrer que si l'évaluation de l'expression e termine et retourne la valeur v et que le type de e est t alors v est aussi de type t. Évaluation et typage se font dans des environnements qui concordent.

Le théorème à démontrer se formule en Coq de la façon suivante :

```
Lemma subject_reduction_theorem:
(e: expr)(v: val)(c: ctx)(env: environment)(t: type)
  (val_of c e v) ->(type_of env e t) ->(ctx_env_match c env)
      ->(type_of_val v t)
```

La preuve se fait par induction sur `(val_of c e v)` et utilise l'inversion. On se retrouve donc avec 7 cas, en particulier 2 pour l'application $e_1\ e_2$: un cas où l'évaluation de e_1 retourne une fermeture simple, et un cas où l'évaluation de e_2 retourne une fermeture récursive. Dans chacun des 2 cas qui se prouvent de manière similaire, on dispose d'une hypothèse d'induction concernant non seulement e_1 et e_2 mais aussi l'expression enfermée dans la fermeture.

Détaillons le cas de la construction `let` : il nécessite le lemme intermédiaire suivant : si une valeur `u` est d'un type `t` alors elle a aussi le schéma de types obtenu en généralisant le type `t`.

```
Lemma sem_gen_gen_type: (u: val)(t: type)(env: environment)
 (type_of_val u t) -> (sem_gen u (gen_type t env)).
```

Prouver ce lemme consiste à montrer pour toute instance générique `t'` de `(gen_type t env)` que `u` a le type `t'`. Dans la certification de W, on a montré un lemme qui établit que si `t'` est une instance générique de

`(gen_type t env)`, on peut aussi écrire `t'` comme instance du type `t`. On montre alors la propriété suivante qui se rapproche de la propriété de conservation du typage par substitution : si `u` est de type `t` alors `u` est aussi de type `s t` pour toute substitution `s`.

```
Lemma type_of_val_stable_subst: (v: val)(t: type)(s: substitution)
(type_of_val v t) -> (type_of_val v (extend_subst_type s t)).
```

La preuve de ce dernier lemme utilise un schéma d'induction mutuelle entre `val` et `ctx` illustré ci-dessous :

```
(P:val->Prop) (P0:ctx->Prop)
  ((n:nat)(P (Num n)))
   ->((i:ident)(e:expr)(c:ctx)(P0 c)->(P (Clos i e c)))
      ->((i,i0:ident)
          (e:expr)(c:ctx)(P0 c)->(P (Rec_clos i i0 e c)))
         ->(P0 Cnil)
            ->((i:ident)
               (v:val)(P v)->(c:ctx)(P0 c)->(P0 (Ccons i v c)))
                 ->(v:val)(P v)
```

Voici par exemple le sous-but concernant la fermeture simple :

```
(i:ident)(e:expr)(c:ctx)(s:substitution)
((env:environment)
   (ctx_env_match c env)
      ->(ctx_env_match c (apply_subst_env env s)))
    ->(t:type)
        (type_of_val (Clos i e c) t)
          ->(type_of_val (Clos i e c) (extend_subst_type s t))
```

La preuve de ce sous-but utilise la propriété de conservation du typage par substitution. Nous en donnons ici un script : par inversion, on déduit de l'hypothèse `(type_of_val (Clos i e c) t)` que `t` est un type fonctionnel `t1 ->t2` et que les 2 assertions `(ctx_env_match c env)` et `(type_of env (Lambda i e) (t1 ->t2))` sont vraies. Le but se récrit donc `(type_of_val (Clos i e c) ( (s t1) ->(s t2)))`. On va le montrer en se reportant à l'environnement de types `(s env)`. D'après la définition de `sem_gen`, cela signifie que :

- le contexte `c` concorde avec `(s env)`, ce qui est trivial compte tenu de l'hypothèse d'induction sur `c`
- le séquent de typage `(type_of (s env) (Lambda i e) ((s t1) ->(s t2)))` est satisfait, celui-ci se déduit de la propriété de conservation du typage.

Plus avant dans la vérification, nous rencontrons le sous-but suivant (nous avons omis les hypothèses de typage) :

```
H : (t:type)(s:substitution)
      (type_of_val v0 t)->(type_of_val v0 (s t))
H0 : (env:environment)(s:substitution)
       (ctx_env_match c env)
         ->(ctx_env_match c (apply_subst_env env s))
H2 : (sem_gen v0 ts)
H3 : (ctx_env_match c env0)
============================
 (sem_gen v0 (extend_subst_type_scheme s ts))
```

Cette partie est un peu plus technique: il s'agit de montrer que pour toute instance générique t de `(s ts)`, `v` a le type t . Soit t une telle instance, soit s_g la substitution générique telle que $t = s_g$`(s ts)`. On peut montrer alors que t est une instance du type T (soit ϕ la substitution telle que $t = \phi T$) avec T instance générique fraîche (obtenue en remplaçant toutes les variables génériques de `ts` par des nouvelles variables). Cette propriété a été montrée dans la preuve de complétude de W, la notion de *nouvelle variable* y a été formalisée.

Il faut alors prouver `(type_of_val v0` ϕT). Ceci se déduit de l'hypothèse de récurrence (`H`) sur `v0` et de la preuve que `v0` a le type T,. Cette dernière preuve découle de la définition de `(sem_gen v0 ts)` et du fait que T est une instance générique de `ts` (lemme auxilaire provenant de la certification de W).

9. Autres travaux

Nous faisons référence ici aux autres travaux qui concernent la vérification de la sûreté du typage de ML à l'aide d'un démonstrateur de théorèmes.

D. Terrasse a prouvé à l'aide de Coq le théorème subject reduction pour un mini-ML sans `let` , donc sans polymorphisme.

En 1991, S. Michaylov et F. Pfenning [12] ont traité le problème en utilisant le système Elf. La syntaxe des expressions du langage (Mini-ML) est une syntaxe d'ordre supérieur, ce trait transparait dans la définition de la sémantique dynamique. D'autre part, le système de types est modélisé en utilisant la notion de substitution dans une expression : ainsi, le type de `let x = e1 in e2` est obtenu en typant l'expression `e2` après avoir remplacé les occurrences de `x` par `e1`.

Citons également J. Frost [7] qui comme dans nos travaux manipule des fonctions récursives. Il associe à celles-ci une fermeture infinie qu'il

représente par des types coinductifs. Il prouve la sûreté du typage dans Isabelle/HOL et Isabelle/ZF.

Les travaux précédents, comme le nôtre d'ailleurs, concernent des langages jouets. M VanInwegen [19] a entrepris de prouver avec HOL la sûreté du typage pour core-SML. Ce travail est en cours : à notre connaissance, aucune publication ne traite du travail achevé.

10. Perspectives et conclusion

Dans cet article, nous avons spécifié en Coq la syntaxe abstraite, la sémantique statique et la sémantique dynamique d'un langage à la Mini-ML. Nous avons formalisé une version *déterministe* du système de types que nous avons prouvée équivalente à la définition initiale de Damas et Milner. Nous avons ensuite donné une preuve formelle mécanisée de la sûreté du typage par rapport à l'évaluation. On pourrait compléter cette théorie du typage de ML par la propriété dite du type principal : elle établit que toute expression qui admet un type a un type principal, i.e. un type dont tous les autres sont instances, c'est précisément celui calculé par W. Par conséquent, on réutiliserait les preuves faites pour W.

La sémantique dynamique est ici une sémantique dite à grands pas. Une perspective intéressante serait de considérer maintenant une sémantique à réduction, dite à petits pas, comme celle proposée par Wright et Felleisen [20]. On pourrait alors montrer que le typage est conservé à chaque étape de réduction, sans se limiter aux programmes qui terminent. On pourrait envisager également de montrer que toute réduction bloquée correspond à un programme mal typé.

Enfin, une autre ouverture à ce sujet serait de *franchir le pas*, c'est-à-dire de passer à un langage réel, Objective ML par exemple ! Un petit pas a été fait dans ce sens, puisque nous avons introduit les références. Pour l'instant seule la correction de l'inférence de types a été établie dans un contexte où le polymorphisme est restreint aux valeurs. Cette expérience est décrite dans [6].

Références

[1] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Pascal Manoury, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi and

Benjamin Werner. The Coq Proof Assistant, Reference Manual, Version 6.1. INRIA, Rocquencourt, December 1996.

[2] Samuel Boutin. Proving Correctness of the Translation from Mini-ML to the CAM with the Coq Proof Development System. Research report RR-2536, INRIA, Rocquencourt, April 1995.

[3] Dominique Clement, Joelle Despeyroux, Thierry Despeyroux, and Gilles 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] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[5] Catherine Dubois and Valérie 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.

[6] Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. Rapport de recherche no 30, Université d'Evry, octobre 1997.

[7] Jacob Frost. A Case Study of Co-induction in Isabelle. Technical Report 359 University of Cambridge, Computer Laboratory, February 1995.

[8] Andrew D. Gordon and Tom Melham. Five Axioms of Alpha-Conversion In *Proceedings of Theorem Proving in Higher Order Logics*, LNCS 1125, Springer-Verlag, 173190, 1996.

[9] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, 1987.

[10] Xavier Leroy. Polymorphic typing of an algorithmic language. research report (english version of his PhD thesis at université Paris 7) RR-1778, INRIA, Rocquencourt, 1992.

[11] Wolfgang Naraschewski and Tobias Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. In C. Paulin-Mohring, editor, *Proceedings of the International Workshop TYPES'96*, LNCS, Springer-Verlag, 1997.

[12] S. Michaylov and F. Pfenning. Natural Semantics and some of its meta-theory in Elf. In Lars Halln, editor, *Proceedings of the Second Workshop on Extensions of Logic Programming*, Springer-Verlag LNCS, 1991. Also available as a Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrucken, Germany, August 1991.

[13] Tobias Nipkow and David von Oheimb. Java_{light} is Type-Safe - Definitely. POPL'98.

[14] Catherine Parent. Developing certified programs in Coq - The Program Tactic. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, LNCS 806, pages 291–312, Springer-Verlag, 1993.

[15] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML Programs in the System Coq. *Journal of Symbolic Computation–special issue on automated programming*, 15(5&6):607–640, May&June 1993.

[16] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, LNCS 442. Springer-Verlag, 1990. Also available as technical report CMU-CS-89-209.

[17] David Syme. Proving Java type soundness Technical Report 427, University of Cambridge Computer Laboratory, 1997.

[18] Delphine 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.

[19] Maria VanInwegen. Towards type preservation for core SML. University of Cambridge Computer Laboratory, 1997.

[20] Andrew Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness In *Information and Computation* 115(1), pp.38-94, 1994.