Typage de ML : Spécification et Preuve en Coq

Catherine Dubois
LaMI, Université d'Évry Val d'Essonne,
Catherine.Dubois@lami.univ-evry.fr

Valérie Ménissier-Morain
LIP6, Université Paris 6,
Valerie.Menissier@lip6.fr

1 Introduction

Le travail présenté dans cet article constitue une certification formelle complète des aspects liés au typage de ML. En effet, nous fournissons, pour un sous-ensemble de ML, une spécification de la sémantique dynamique, de la sémantique statique et de l'algorithme d'inférence appelé traditionnellement W. Nous prouvons la correction 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. Nous avons pris le parti de rester le plus proche possible d'une implantation fonctionnelle de W.

Au delà du challenge 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 permet 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.
De plus, l'inférence de types est une étape majeure durant le processus de compilation, par conséquent toute certification de compilateur ML ne peut ignorer cette phase et se doit d'incorporer la certification de W.

A notre connaissance, aucune publication ne mentionne une certification formelle mécanisée d'un tel ensemble. Dans une approche assez similaire à la nôtre (menée indépendamment et simultanément), Naraschewski et Nipkow [12] ont certifié W avec le démonstrateur Isabelle/HOL mais sans faire de liaison avec la sémantique dynamique de ML. D'autres auteurs, comme Michaylov et Pfenning [13], Frost [8] , plus récemment VanInwegen pour Core-SML [15] ont traité la correction du typage par rapport à la sémantique dynamique mais sans liaison avec W.

Nous avons choisi le système Coq (version 6.1) pour réaliser notre validation. 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.

Dans cet article, tout d'abord (section 2), 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 3) et de son système de types (section 4) avec, entre autres, la théorie sous-jacente des substitutions. Ensuite, dans la section 5, nous prouvons la correction du typage par rapport à l'évaluation et étudions brièvement les approches adoptées dans les autres travaux relatifs à ce sujet. L'algorithme W est implanté en Coq dans la partie 6 et nous présentons sa certification par rapport aux règles de typage dans la section 7.

Journées du GDR Programmation.
12, 13, 14 novembre 1997.
Rennes.

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 est accessible via `http://pauillac.inria.fr/~menissier`.

2 Le langage

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

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

On pourra trouver une présentation élégante d'un langage similaire au nôtre avec les preuves associées, dans le premier chapitre de la thèse de Leroy [11].

3 La sémantique dynamique

Nous définissons la sémantique dynamique, i.e. la relation d'évaluation des expressions, dans le style de la Sémantique Naturelle [10]. Seuls les programmes qui terminent sans fournir un résultat d'erreur y sont considérés. Auparavant, il nous faut préciser les valeurs sémantiques, c'est-à-dire les valeurs que peuvent prendre les expressions.

3.1 Valeurs sémantiques et contextes d'évaluation

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)*.

On retient ici comme seules valeurs possibles les nombres et les valeurs fonctionnelles appelées aussi fermetures. 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 $<<_{@}$ `Rec` $f\ x.e, \Delta >>$). Boutin propose également cette distinction dans son travail concernant la certification du compilateur ML [2].

Les notions de valeur et de contexte donnent lieu en Coq à la définition des 2 types mutuellement inductifs `val` et `ctx` :

```
Mutual Inductive val : Set := Num: nat -> val
                            | Closure : identifier -> expr -> ctx -> val
                            | Rec_closure :identifier -> identifier -> expr -> ctx -> val
with ctx : Set :=  Cnil: ctx
                 | Ccons: identifier -> val -> ctx -> ctx.
```

On notera $\Delta(x)$ ou (`assoc_ident_in_ctx` Δ x) (en Coq) la valeur associée à un identificateur donné x dans le contexte Δ et $\Delta \oplus x : v$ l'ajout du couple (x, v) dans Δ. La fonction `assoc_ident_in_ctx` est une fonction partielle, elle échoue si l'identificateur n'appartient pas au domaine du contexte.

3.2 Les règles d'inférence de la sémantique dynamique et leur formalisation en Coq

La figure 1 présente les règles d'inférence dans le style de la Sémantique Naturelle. 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 œuvre une sémantique de l'appel par valeur. On remarquera les 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.

$$
\begin{array}{ll}
(\textsc{cst}) & \Delta \vdash n : n \\[2ex]
(\textsc{id}) & \Delta \vdash x : \Delta(x) \\[2ex]
(\textsc{abs}) & \Delta \vdash \lambda\ x.e : << \lambda\ x.e, \Delta >> \\[2ex]
(\textsc{rec}) & \Delta \vdash \texttt{Rec}\ f\ x.e : <<_{@}\ \texttt{Rec}\ f\ x.e, \Delta >> \\[2ex]
(\textsc{app1}) & \dfrac{\Delta \vdash e : << \lambda\ x.e_f, \Delta_f >>\ , \quad \Delta \vdash e' : v, \quad \Delta_f \oplus x : v \vdash e_f : v'}{\Delta \vdash e\ e' : v'} \\[2ex]
(\textsc{app2}) & \dfrac{\Delta \vdash e : <<_{@}\ \texttt{Rec}\ f\ x.e_f, \Delta_f >>\ , \quad \Delta \vdash e' : v, \quad \Delta_f \oplus x : v \oplus f : <<_{@}\ \texttt{Rec}\ f\ x.e_f, \Delta_f >> \vdash e_f : v'}{\Delta \vdash e\ e' : v'} \\[2ex]
(\textsc{let}) & \dfrac{\Delta \vdash e : v, \quad \Delta \oplus x : v \vdash e' : v'}{\Delta \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : v'}
\end{array}
$$

FIG. 1 – *La sémantique dynamique*

Les règles de sémantique naturelle de la figure 1 se traduisent en Coq en le prédicat inductif `val_of` de type `ctx ->expr ->val ->Prop`. La correspondance est quasi syntaxique : on trouve un constructeur par règle d'inférence. On pourra consulter [14] pour un cadre plus général de traduction de la Sémantique Naturelle en Coq.

4 Le système de types

Le système de types que nous décrivons ci-dessous met en œuvre le polymorphisme paramétrique. Seule la construction `let` peut introduire des fonctions polymorphes auxquelles on associe un type quantifié, plus précisément un schéma de types. Dans la suite, nous formalisons les notions de type et schéma de type en Coq. Nous abordons ensuite la notion d'environnement et discutons l'opération de généralisation Avant de détailler les règles de typage, nous spécifions les notions de substitution et d'instance.

4.1 Types et schémas de type

Les types considérés sont le type de base *int*, les variables de type α, β ... et les types fonctionnels $\tau \rightarrow \tau'$.
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` ave 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 $\forall\alpha.\alpha \rightarrow \beta$, 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.
```

4.2 Environnement

Les informations de type relatives aux identificateurs non liés des expressions sont maintenues dans un environnement. Nous avons mentionné plus haut qu'un environnement On peut considérer une telle structure comme une liste de paires *(identificateur, schéma de type)*. Cette vue est retenue en Coq : le type `environment` est donc défini par `list (identifier * type_scheme)`. Structurellement, la notion d'environnement est très proche de celle de contexte d'évaluation. On utilise ainsi pour les environnements les mêmes notations que pour les contextes : $\Gamma(x), \Gamma \oplus y : \sigma$.

4.3 Généralisation de type

Seule la construction `let` introduit dans l'environnement 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'implanation 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 [9] (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 [12].

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 `n`ième 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` α $\emptyset$) = `(Gen 0)` = (`gen_type` β $\emptyset$).

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.

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.

4.4 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.

4.4.1 Instance de type et substitution

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. Nous détaillons ci-dessous l'opération de composition des substitutions.

```
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` à 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)` à la variable `st` produit le même terme que celui obtenu en appliquant `s2` au résultat de l'application de `s1` à `st`. La preuve de ce lemme réclame un effort important et compte environ 600 lignes.

Remarque: on peut également proposer de représenter une substitution par une abstraction Coq et profiter ainsi de ses capacités d'ordre supérieur. Composer deux substitutions devient alors un jeu d'enfant ! Naraschewski et Nipkow [12] 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 [7].

4.4.2 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 exact, il faudrait écrire : le type à substituer à la variable générique `(Gen n)` est le `n`-ième 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).

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

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

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

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

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

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

FIG. 2 – *Les règles de typage*

Par conséquent la fonction `apply_subst_gen` qui applique une substitution générique s_g à un schéma de type σ en vue de produire un type est partiellement définie : en effet, elle n'a de sens que si la longueur du vecteur s_g concorde avec la plus grande des variables génériques de σ. Elle est formalisée en Coq en simulant le mécanisme des exceptions.

Le prédicat `is_gen_instance` spécifie qu'un type τ est instance générique d'un schéma $\forall \alpha_1 \dots \alpha_n.\tau$ si et seulement si il existe une substitution s_g pour $\alpha_1 \dots \alpha_n$ telle que $s_g(\tau) = \tau'$:

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

4.4.3 La relation `more_general` entre schémas de type

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, $\forall \alpha\beta.\alpha \rightarrow \beta \succ \forall \alpha.\alpha \rightarrow \alpha$.

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

Cette définition est étendue aux environnements : un environnement Γ_1 is dit plus général qu'un environnement Γ_2 ($\Gamma_1 \succ \Gamma_2$ ou (`more_general_env` Γ_1 Γ_2)) si et seulement si ils sont relatifs aux mêmes identificateurs $x_1, x_2 \dots x_n$ et $\forall i \in [1, n],\ \Gamma_1(x_i) \succ \Gamma_2(x_i)$.

4.5 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] et non la version initiale de Damas Milner. 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 en fait équivalentes (C. Dubois a prouvé formellement cette propriété en Coq, voir [5] pour plus de détails).

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

4.6 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. Le preuve de cette propriété 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. La preuve est détaillée dans [7].

Cette propriété est utile à plusieurs reprises dans l'ensemble du travail présenté ici.

5 Correction 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, i.e 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 correction du typage, plus précisément le théorème traditionnellement appelé *Subject Reduction Theorem.*

5.1 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` Δ Γ)), 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` $\Delta(i)$ $\Gamma(i)$).

5.2 Correction 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. Le cas de la construction `let` 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)).
```

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

$W(\Gamma, x) =$
 let $(\forall\ v_1 \ldots v_n.\tau) = \Gamma(x)$
 let $var_1, \ldots, var_n$ des nouvelles variables
 let $S = \{v_i \mapsto var_i\}$
 in $(S\tau, \emptyset)$

$W(\Gamma, \lambda x.e) =$
 let α une nouvelle variable
 let $\tau, S = W(\Gamma \oplus x : \forall.\alpha, e)$
 in $(S\alpha \to \tau, S)$

$W(\Gamma, \texttt{Rec}\ f\ x.e) =$
 let α, β 2 nouvelles variables
 let $\tau, S = W(\Gamma \oplus f : \forall.\alpha \to \beta \oplus x : \forall.\alpha, e)$
 let $\mu = \texttt{unify}(\tau, S\ \beta)$
 in $(\mu\ o\ S(\alpha \to \beta), \mu\ o\ S)$

$W(\Gamma, e_1\ e_2) =$
 let $\tau_1, S_1 = W(\Gamma, e_1)$
 let $\tau_2, S_2 = W(S_1\Gamma, e_2)$
 let α une nouvelle variable
 let $\mu = \texttt{unify}(S_2\tau_1, \tau_2 \to \alpha)$
 in $(\mu\alpha, \mu \circ S_2 \circ S_1)$

$W(\Gamma, \texttt{let}\ x = e_1 \texttt{in}\ e_2) =$
 let $\tau_1, S_1 = W(\Gamma, e_1)$
 let $\sigma = \texttt{gen\_type}\ \tau_1\ S_1\Gamma$
 let $\tau_2, S_2 = W(S_1\Gamma \oplus x : \sigma, e_2)$
 in $(\tau_2, S_2 \circ S_1)$

FIG. 3 – *L'algorithme W*

Prouver ce lemme consiste à montrer pour toute instance générique `t'` de `(gen_type t env)` que `u` a le type `t'`. On dispose d'un lemme qui établit que toute instance générique `t'` de `(gen_type t env)` peut aussi s'écrire comme instance du type `t`. On démontre 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`. La preuve de cette dernière proposition met en œuvre un schéma d'induction mutuelle sur `val` et `ctx`. Les sous-buts concernant les fermetures utilisent la propriété de conservation du typage par substitution.

5.3 Autres travaux relatifs

Nous faisons référence ici aux travaux relatifs à la vérification de la correction du typage de ML à l'aide d'un démonstrateur de théorèmes.

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

En 1991, Michaylov et Pfenning [13] 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 Frost [8] qui comme dans nos travaux manipule des fonctions récursives. Il associe à celles-ci une fermeture infinie qu'il représente par un type coinductif. Il prouve la correction du typage dans Isabelle/HOL et Isabelle/ZF.

Les travaux précédents, comme le nôtre d'ailleurs, concernent des langages jouets. Dans un travail en cours, VanInwegen [15] a entrepris de prouver avec HOL la correction du typage pour core-SML.

6 Inférence de types

Nous adaptons l'algorithme W de Damas-Milner [4] à notre langage. Nous définissons $W(\Gamma, e)$ comme la paire (τ, S) : le type τ est le type le plus général de l'expression e et la substitution S contient les instantiations effectuées durant le calcul. L'algorithme W est décrit dans la figure 3.

6.1 Unification

L'algorithme W utilise le mécanisme de l'unification : `unify (t1, t2)` calcule l'unificateur le plus général (mgu) des 2 types `t1` et `t2` si ceux-ci sont unifiables, échoue sinon. La fonction `unify` n'est pas explicitement définie dans notre modélisation mais uniquement spécifiée par son type `Parameter unify : type*type ->unify_check` (le type `unify_check` étant le type *substitution ou erreur*) et quelques axiomes. En effet, la certification de l'inférence de types est indépendante de l'implantation de l'algorithme d'unification. Nous avons emprunté cette méthodologie à Naraschewski et Nipkow [12].

Les axiomes énoncent la correction et la complétude de `unify`. Un axiome plus technique assure que l'algorithme d'unification calcule un unificateur dont le domaine et le co-domaine ne contiennent que des variables issues du problème initial.

6.2 Formalisation de W en Coq

La formalisation en Coq est très proche de cette définition. A nouveau, on utilise un type inductif pour simuler les échecs de W. En outre, il nous faut faire apparaître explicitement la gestion des nouvelles variables de type (encore appelées variables fraîches). Les variables étant codées par des entiers, il est fondamental de savoir à tout instant quels sont les entiers déjà utilisés au cours du calcul. Tout naturellement, pour simplifier la gestion, les variables sont engendrées *dans l'ordre* depuis 0. Ainsi W prend comme argument supplémentaire la variable la plus grande déjà utilisée (plus exactement son successeur). En cas de succès, W retourne également ce paramètre modifié en conséquence. On montre formellement que celui-ci croît au cours du calcul (propriété requise ultérieurement par la preuve) :

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

7 Certification de l'inférence de types

Certifier l'inférence de types signifie prouver formellement que W *calcule le type principal de toute expression bien typée ou échoue sinon.* Cette certification comporte 2 étapes. Tout d'abord nous prouvons que W est correct par rapport aux règles de typage. Puis nous montrons qu'il est complet vis-à-vis d'elles.

7.1 Correction de l'inférence de types

La propriété de correction établit que si $W(\Gamma, e)$ calcule le type τ et la substitution S alors le séquent $S\Gamma \vdash e : \tau$ est satisfait. Soit en Coq :

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

La preuve est faite par induction sur l'expression e. Pour chacun des cas, nous introduisons dans le contexte des hypothèses les déductions qui dérivent de l'hypothèse *$W(\Gamma, e)$ réussit.* Cette méthode est comparable aux tactiques prédéfinies d'inversion fournies pour les prédicats inductifs. Les cas de l'identificateur et du `let` sont les plus intéressants, ce sont aussi ceux qui ont résisté le plus. Ils sont détaillés dans [7]. Notons que la preuve utilise la propriété de conservation du typage par substitution.

7.2 Complétude de l'inférence de types

Caractérisation des nouvelles variables

La preuve de complétude nécessite de caractériser la notion de *nouvelle variable*, encore appelée *variable fraîche*. La plupart des travaux «papier» relatifs à l'inférence de types ne formalise pas cette notion et la laisse très floue. En revanche toute vérification mécanisée n'échappe pas à cette formalisation.

La notion de nouvelle variable se définit relativement à un type, un schéma de type, un environnement ou une substitution (ci dessous r désigne une telle structure).

Une variable α est une nouvelle variable de type pour r si elle n'est pas utilisée en tant que variable libre dans r. Nous proposons une définition plus spécifique à notre modélisation : elle prend en compte l'implantation entière des variables par des entiers et la gestion des variables fraîches dans W. Ainsi une variable α est une nouvelle variable pour r si et seulement si toute variable libre de r est strictement inférieure à α. Le prédicat inductif `new_tv_type_scheme` implante cette définition pour le cas des schémas de type :

```
Inductive new_tv_type_scheme: type_scheme -> stamp -> Prop:=
  new_tv_ts_Nat: (st: stamp) (new_tv_type_scheme Nat_ts st)
| new_tv_ts_Var: (st,st': stamp) (lt_stamp st st') -> (new_tv_type_scheme (Var_ts st) st')
| new_tv_ts_Gen: (st,st': stamp) (new_tv_type_scheme (Gen_var st) st')
| new_tv_ts_Arrow: (ts1, ts2 : type_scheme) (st : stamp)
   (new_tv_type_scheme ts1 st) -> (new_tv_type_scheme ts2 st)
     -> (new_tv_type_scheme (Arrow_ts ts1 ts2)  st).
```

Ces définitions sont complétées par de nombreux lemmes techniques, comme par exemple `new_tv_W` qui établit que l'entier résultat de W est une nouvelle variable par rapport aux 2 autres résultats.

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

Formulation de l'énoncé de complétude

Dans les grandes lignes, la complétude de W signifie que *si on peut proposer une solution au problème de typage de l'expression e relativement à l'environnement* Γ, *alors l'inférence va réussir et retourner la solution la plus générale dont la solution proposée se déduit.* Plus formellement, la complétude établit que si on peut associer un type τ' à e relativement à Γ où quelques variables libres ont éventuellement été contraintes (via la substitution ϕ), alors pour *st* un entier *correct* (au sens des nouvelles variables), (`W` *st* Γ *e*) produit un type τ et une substitution S, de plus τ' et ϕ se déduisent de τ et S. Nous raffinons cette dernière propriété de la façon suivante :
• τ' est une instance de τ : soit s' la substitution telle que $s'\ \tau = \tau'$
• et $\phi = s' \circ S$; cette égalité n'a de sens que pour les variables apparaissant dans la dérivation de $\phi\Gamma \vdash e :\ \tau'$.
Enfin, la propriété de complétude se formule en pseudo-Coq de la manière suivante :

```
Lemma completeness: (e: expr)(env: environment)(t': type)(phi: substitution)(st: stamp)
 (type_of (apply_subst_env env phi) e t') -> (new_tv_env env st) ->
   {S, T, st', s'| (W st env e) = (Some_infer T S st')) /\ t'=(apply_subst_type s' T) /\
     ((x: stamp) (lt_stamp x st) -> phi(x) = (compose_subst S s') (x))}.
```

Preuve de la complétude

La preuve de la complétude requiert une autre propriété importante du système de types, à savoir :

$$si\ \ \Gamma_2 \vdash\ e :\ \tau,\quad \Gamma_1 \succ \Gamma_2\ \ alors\ \Gamma_1 \vdash\ e :\ \tau.$$

Cette dernière propriété se prouve facilement par induction sur e. Le cas du `let` découle de la «monotonie» de la généralisation par rapport à la relation $\succ$: *si* $\Gamma_1 \succ \Gamma_2$, *alors* (`gen_type` τ Γ_1) $\succ$ (`gen_type` τ Γ_2).

La preuve de complétude, quant à elle, se fait par induction sur e. La vérification de chacun des cas suit le processus suivant : des hypothèses on déduit que W réussit et nous exhibons ses résultats S et T. Cette étape est lourde, en particulier pour l'application et la construction `Rec` (car il faut prendre en compte des *mgus*). Ensuite on doit construire la substitution s' qui permet de recréer la solution proposée à partir de S et T. Cette phase comporte de longs développements calculatoires.

Cette fois encore, la difficulté vient du `let`, seul cas où on ne peut faire un usage direct des hypothèses d'induction. A ce point de la certification, la solution vient de la propriété fondamentale, mais peu facile à établir, entre substitution et généralisation : σ(`gen_type` τ Γ) $\succ$ (`gen_type` $\sigma\tau$ $\sigma\Gamma$).

8 Conclusion

Dans cet article, nous avons fourni une théorie complète du typage de ML en Coq, ce qui représente un développement conséquent : nombreuses théories complexes (substitutions, renommage, nouvelles variables ...), environ 8500 lignes et des mois de travail ! Une étape de développement ultérieur sera de passer au stade de réutilisation, i.e. d'intégrer certaines des variantes proposées dans la littérature.

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. Correction du Typage de ML : Spécification et Preuve en Coq. Septembre 1997.

[6] 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.

[7] Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. Mai 1997, available via `http://pauillac.inria.fr/~menissier`.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] Spiro Michaylov and Frank 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.

[14] 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.

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