Sémantique Naturelle et Coq : vers la spécification et les preuves sur les langages à objets

Olivier Laurent

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Sémantique Naturelle et Coq : vers la spécification et les preuves sur les langages à objets

Olivier LAURENT

N° 3307

Novembre 1997

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Sémantique Naturelle et Coq : vers la spécification et les preuves sur les langages à objets

Olivier LAURENT*

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport de recherche n° 3307 — Novembre 1997 — 45 pages

Résumé : Le système Centaur permet la description d'un langage de programmation en Sémantique Naturelle. Cette description peut être exécutée (en Prolog) et testée puis traduite en données pour le système Coq pour permettre la réalisation de preuves de propriétés de cette sémantique. Nous présentons ici les premiers pas dans l'application de cette approche aux langages à objets en s'appuyant sur le sigma-calcul défini par M. Abadi et L. Cardelli dans leur livre A Theory of Objects.

La réalisation en Coq de la preuve de conservation des types (SRT) pour le sigma-calcul avec récursion et sous-typage (Ob1<mu) a confirmé certaines faiblesses des systèmes actuels en particulier dans le domaine de la lisibilité des règles de Sémantique Naturelle décrites en Typol (le langage d'implémentation de la Sémantique Naturelle). Nous proposons dans cet article une solution possible à ces problèmes à travers un enrichissement du langage Typol.

Mots-clés : Sémantique, Sémantique Naturelle, Centaur, Coq, preuves, langages à objets, sigma-calcul

* e-mail : olivier.laurent@ens.fr http://www.eleves.ens.fr:8080/home/laurent/

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 93 65 77 77 - International : +33 4 93 65 77 77 — Fax : 04 93 65 77 65 - International : +33 4 93 65 77 65
à partir du 01/01/1998
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Natural Semantics and Coq : towards the specification and proofs on object-oriented languages

Abstract: The Centaur system enables one to specify programming languages in Natural Semantics. These specifications may be executed (in Prolog), tested, and then translated into the Coq system in order to prove some of their properties. We present here the first experiment on object-oriented languages, following this approach. As a small yet interesting example, we chosed the sigma-calculus with recursivity and subtyping (Ob1<mu) defined by M. Abadi et L. Cardelli in their book (A Theory of Objects).

The formalization of the sigma-calculus Ob1<mu in Centaur and the proof of type soundness for it in the Coq system have confirmed some drawbacks of the systems used. In particular, most of the inference rules written in Typol (the implementation language for Natural Semantics in Centaur) are far from being easy to read. We present in this paper a possible answer to the difficulties we encountered, through some extensions of the Typol language.

Key-words: Semantics, Natural Semantics, Centaur, Coq, proofs, object-oriented languages, sigma-calculus

Table des matières

1 Introduction

L'étude des langages de programmation à travers la Sémantique Naturelle [8, 12, 14] est actuellement effectuée au sein du projet CROAP de l'INRIA Sophia-Antipolis grâce à deux outils : Centaur et le langage Typol, d'une part, qui permettent de définir la Sémantique Naturelle [19, 23] de langages et d'étudier son exécution et le système d'aide à la preuve Coq, d'autre part, qui permet de prouver des propriétés de cette sémantique. D. Terrasse-Kaplan a développé une traduction entre ces deux systèmes [22] qui permet d'effectuer des preuves sur des données Coq générées automatiquement à partir des spécifications Typol sans avoir à écrire deux définitions (dont la cohérence resterait à prouver). Ainsi Centaur et Coq forment un système complet et certifié qui permet la spécification de la Sémantique Naturelle d'un langage et son étude simultanément sur son exécution et sur ses propriétés, garantissant la validité des *preuves* effectuées sur la *spécification exécutable*.

Nous présentons les premiers pas effectués dans l'étude des langages à objets à travers ce système dont l'ouverture vers Coq n'a été jusqu'ici que peu utilisée. Pour ce faire, nous nous appuyons principalement sur le ζ-calcul (prononcer "sigma-calcul") avec récursion et sous-typage ($Ob_{1<:\mu}$) décrit par M. Abadi et L. Cardelli [1] pour lequel nous prouvons le théorème de conservation des types (SRT), ce qui permet de faire ressortir les principaux problèmes spécifiques au domaine de l'orienté objet mais également des problèmes plus généraux.

La réalisation de cette étude de $Ob_{1<:\mu}$ nécessite tout d'abord la définition de sa Sémantique Naturelle à travers le formalisme Typol apportant d'importants problèmes de complexité et de lisibilité liés à des faiblesses de Typol à s'adapter au domaine des objets. La traduction vers le système Coq permet ensuite la démonstration de la propriété de conservation des types en Coq ce qui met en avant la difficulté de réaliser certaines preuves inductives en Coq et permet de dégager des améliorations possibles passant principalement par l'enrichissement de Typol. L'étude d'autres preuves confirme alors les problèmes rencontrés et les solutions apportées tout en permettant de les préciser. Par ailleurs, les preuves effectuées permettent l'étude de diverses propriétés de la Sémantique Naturelle du langage considéré amenant à une analyse des modifications possibles des systèmes de règles de typage et d'évaluation.

2 Outils utilisés

Ce stage a eu pour support différents outils développés par deux projets de l'INRIA : les systèmes Centaur et CtCoq dans le projet CROAP de l'INRIA Sophia-Antipolis et le système Coq par le projet Coq de l'INRIA Rocquencourt en collaboration avec l'équipe Coq-Lyon de l'ENS Lyon.

2.1 Centaur

Le système Centaur [7, 18] permet de d'élaborer des environnements interactifs d'évaluation de langages de programmation. Pour cela, l'utilisateur définit les syntaxes concrètes et abstraites de son langage dans le méta-langage Métal, puis écrit un "*pretty printer*" (formattage) grâce à Ppml et enfin décrit la Sémantique Naturelle de son langage par un système de règles d'inférence en Typol données sur la syntaxe abstraite du langage.

Centaur a été utilisé pour le développement d'environnements d'étude de langages plus complexes que ceux étudiés ici comme Esterel, Eiffel, Eiffel//, Java, ...

2.1.1 Métal, AS et CS

Le méta-langage Métal [20] permet la définition de la syntaxe d'un langage aussi bien abstaite que concrète et pour cela contient deux parties assez distinctes. La définition de la syntaxe concrète (gérée ensuite comme une grammaire LALR(1)) d'un micro-langage comportant une addition sur les entiers associative à gauche peut s'écrire :

```
<axiom> ::= <exp> ;                 -- point d'entree pour l'analyse
   <exp>
<exp>   ::= <term> ;
   <term>
<exp>   ::= <exp> "+" <term> ;      -- analyse proprement dite
   add (<exp>, <term>)              -- génération de l'arbre de syntaxe
                                        abstraite correspondant
<term> ::= "(" <exp> ")" ;
   <exp>
<term>  ::= <int> ;
   <int>
<int> ::= %INTEGER ;
   int-atom(%INTEGER)
```

Cette définition permet de construire par exemple un arbre de syntaxe abstraite contenant les nœuds **add** et **int**. La définition de la syntaxe abstraite correspondante, enrichie de définitions pour les types, s'écrira :

```
abstract syntax
   add  -> EXP EXP ;
   EXP  ::= add int ;
   int  -> implemented as INTEGER ;
   tint -> implemented as SINGLETON ;
   TYPE ::= tint ;
```

Le nœud add prend deux paramètres tous deux de *phylum* (type) EXP et renvoie une valeur de *phylum* EXP tout comme int. Le nœud int est un nœud de valeur entière alors que tint est un nœud sans valeur (feuille).

Actuellement, T. Despeyroux travaille au développement de deux autres langages destinés à remplacer Métal et pourvus d'une syntaxe plus lisible. Les syntaxes concrètes et abstraites y sont séparées dans les langages CS et AS [16]. Les deux exemples précédents peuvent alors s'écrire :

```
concrete syntax std of L is
  use L;
```

```
  term : EXP;
  <A> --> EXP(A) ;       -- point d'entree pour l'analyse
  @EXP:
     A --> term(A) ;
     add(A, B) --> EXP(A) "+" term(B) ;
  term:
     A --> "(" EXP(A) ")" ;
     int A --> <"INTEGER" A::integer> ;
end L;
```

où @ sert à exprimer explicitement que EXP est recursif (ce caractère devra être généré dans une version ultérieure).

Et pour la syntaxe abstraite, on obtient en AS :

```
abstract syntax of L is
  add  : EXP # EXP -> EXP;
  int  : integer -> EXP;
  tint : -> TYPE;
end L;
```

2.1.2 Ppml

A partir de la définition d'une syntaxe abstraite en Métal (ou en AS), le langage Ppml permet de définir un *pretty printer* par cas sur chaque constructeur de la syntaxe. Ainsi dans le cas de notre micro-langage, on peut définir :

```
prettyprinter basic of L is
  default
      <h 0>;
  function prec is
      prec (*op where *op in {add})= 1;
      prec (*op)= 0;
  end prec ;
  rules
    *x where *x in {add(*e1, *e2)} ->
        [<h 1> if prec(*e1) > prec(*x) then [<h 0> "(" *e1 ")"]
               else *e1
               end if
               [<h> in class = signe : "+"]
               if prec(*e2) >= prec(*x) then [<h 0> "(" *e2 ")"]
               else *e2
               end if];
    int *nb -> [<h> in class = nb : [<h> integerpp(*nb)]];
end prettyprinter
```

où l'on construit les parenthèses minimales pour une addition associative à gauche grâce à la fonction de priorité **prec**. La définition se fait sous forme de boites (horizontales : [<h 1> ...] ou verticales [<v 0> ...]) placées les unes par rapport aux autres (<h 1> signifie *boites séparées d'un espace*).

Un des problèmes actuels de Ppml est que rien n'assure que le texte généré soit compatible avec la syntaxe concrète définie indépendamment en Métal. Pour résoudre ce problème, T. Despeyroux développe CLF (Computer Language Factory) qui rassemblera entre autres AS et CS, CS étant étendu pour décrire à la fois la syntaxe concrète d'un langage et son *pretty print*.

2.1.3 Typol

Le langage Typol [15] permet de définir la Sémantique Naturelle [8, 12, 19] d'un langage objet directement par un système de règles d'inférence. Ces règles agissent sur la syntaxe abstraite du langage définie en Métal (ou en AS). On pourrait ainsi définir l'évaluation (sémantique dynamique) de notre micro-langage par le jugement $\vdash E \rightarrow v$ qui se lit "l'expression E s'évalue en v" :

```
program eval_L is
  use L;
  import sum(integer, integer, integer) from pl_functions;

  export |- E -> v as eval_L(E) = v ; -- definition d'une fonction permettant
                                     -- l'execution du programme Typol sous
                                     -- Centaur
  judgement |- EXP -> EXP;

  Red_int: |- int n -> int n ;

  Red_add:
      |- e1 -> int n1  &  |- e2 -> int n2  &  sum(n1, n2, n)
      ----------------
      |- add(e1, e2) -> int n ;
end eval_L;
```

où l'on définit la fonction sum directement en Prolog dans le programme *pl_functions* extérieur. L'instruction *export* réalise l'interface de Typol avec Centaur en fournissant à celui-ci une fonction exécutable. On peut de la même manière définir le typage (sémantique statique) de L par le jugement $\vdash E : T$ qui se lit "l'expression E a le type T" :

```
program typ_L is
  use L;
  export |- E : T as typ_L(E) = (T) ;

  judgement |- EXP : TYPE;

  Val_Int: |- int N : tint() ;
```

```
  Val_Add:
     |- a : tint()  &  |- b : tint()
     ----------------
     |- add(a, b) : tint() ;
end typ_L;
```

Comme on le voit ici, les jugements de Typol doivent nécessairement être typés ce qui permet une détection efficace des erreurs les plus flagrantes et assure la cohérence du langage décrit. Pour de gros systèmes de règles, on peut structurer les programmes Typol sous forme de *sets* en explicitant les appels inter-*sets*.

Remarque : cependant cet exemple reste extrèmement simple et les jugements qui apparaissent ici ne sont qu'un cas particulier des jugements que l'on rencontre généralement. En effet, comme nous le verrons plus loin dans la description de langages plus intéressants, l'évaluation et le typage des expressions font souvent intervenir un environnement : $\rho \vdash E \rightarrow v$ ou $\rho \vdash E : T$.

L'environnement Centaur permet l'exécution d'un programme Typol grâce à une traduction des règles en prédicats de Prolog. Centaur fournit un système de débogage qui permet l'exécution pas à pas d'un système de règles, l'utilisation de points d'arrêt, l'observation des différents paramètres durant l'évaluation... On peut ainsi détecter les différents retours en arrière (*backtracks*) que l'on peut parfois éviter pour accélérer l'exécution et de manière plus générale mettre au point la sémantique décrite. Si cette méthode permet la détection de la très grande majorité des erreurs, elle n'est pas toujours suffisante et certaines faiblesses de la sémantique n'apparaissent que lors de la réalisation de preuves sur la sémantique.

Le système de règles, pouvant être a priori non déterministe, est exécuté par défaut de manière déterministe en tenant compte de l'ordre des règles dans le programme et de l'ordre des prémisses dans chaque règle (ce qui peut entraîner d'importants problèmes car cet ordre n'ayant pas véritablement de sens, Typol peut tomber dans des chaînes d'application de règles infinies alors qu'une autre exécution finie est possible).

2.2 Coq

Les différentes preuves sur des sémantiques écrites en Typol ont été réalisées dans le système Coq et plus précisément à l'aide de l'environnement graphique CtCoq construit autour de Coq grâce à une traduction automatique des spécifications écrites en Typol vers des données pour Coq (voir section 4).

2.2.1 Le système Coq

Le système Coq [9] est un système d'aide à la preuve basé sur un λ-calcul typé d'ordre supérieur. Il permet d'énoncer des assertions puis d'effectuer des démonstrations de ces énoncés vérifiées automatiquement par la machine. Les preuves se font par applications successives de tactiques prédéfinies (application d'hypothèses ou de lemmes déjà démontrés, élimination de constructeurs, induction sur un prédicat, coupure, ...) mais il est également possible de définir de nouveaux principes d'induction.

Ainsi, on peut démontrer la symétrie du $\wedge$ logique dans le système Coq de manière assez proche de ce que l'on ferait sur papier :

```
Lemma sym : (A:Prop) (B:Prop) A /\ B -> B /\ A.  (* enonce *)

Intros.    (* elimination de -> et generation de 3 hypotheses
              A:Prop, B:Prop et H:A/\B *)
Elim H.    (* destruction du constructeur /\ dans l'hypothese H:A/\B *)
Intros.    (* mise en hypothese des preuves de A et de B obtenues
              H0:A et H1:B *)
Split.     (* destruction du /\ dans le but courant ce qui genere deux
              sous-buts B et A *)
Apply H1.  (* application de l'hypothese H1:B qui demontre le premier but *)
Apply H0.  (* application de l'hypothese H0:A qui demontre le deuxieme but *)
Qed.       (* verification de la preuve et mise en memoire de l'enonce
              qui pourra etre reutilise *)
```

La notation (A : *Prop*) de Coq signifie $\forall A$: *proposition* avec, par l'isomorphisme de Curry-Howard, correspondance entre proposition et type, entre preuve et terme.

2.2.2 CtCoq

L'environnement CtCoq [5] fournit un contexte graphique de développement de preuves en Coq. Il permet de visualiser de manière beaucoup plus confortable les différents buts à prouver ainsi que la preuve en cours. De plus la possibilité d'utiliser plusieurs fenêtres rend plus facile l'introduction de lemmes intermédiaires dans une preuve. Par ailleurs l'usage de la souris facilite grandement les preuves en particulier grâce à la *preuve par sélection* [6] qui permet de générer les tactiques de preuve directement à la souris. Les preuves ainsi faites peuvent se révéler assez différentes d'une preuve en Coq pur (en particulier en ce qui concerne le script obtenu qui s'avère souvent moins lisible car plus surchargé) :

```
Lemma sym : (A:Prop) (B:Prop) A /\ B -> B /\ A.

Intros A B H'; Elim H'; Intros H'0 H'1; Try Exact H'1; Clear H'.
Split; [Try Assumption | Idtac].
Try Exact H'0.
Qed.
```

La preuve précédente peut se faire sans l'usage du clavier simplement par *trois clics* de souris aux endroits appropriés dans la fenêtre CtCoq. Mais le bon usage de tous les raccourcis nécessite un certain apprentissage et cette méthode n'est pas pour l'instant très appropriée pour les preuves en Sémantique Naturelle (souvent essentiellement inductives).

3 Spécification du ζ-calcul

Nous étudions ici l'adéquation des méta-langages Métal et Typol pour l'implémentation du ζ-calcul $Ob_{1<:\mu}$ défini par M. Abadi et L. Cardelli [1]. En particulier en ce qui concerne la difficulté de transcrire des règles complexes en Typol et la lisibilité des règles de typage et d'évaluation obtenues.

3.1 Syntaxe et règles

La syntaxe utilisée ici ainsi que la forme des règles sont intermédiaires entre celles présentées dans [1] et celles obtenues en Métal et Typol pour des raisons de lisibilité. Nous exposerons plus tard (sections 3.2 et 3.3) les problèmes liés aux spécifications Typol elles-mêmes.

3.1.1 Syntaxe de $Ob_{1<:\mu}$

Nous présentons ici un langage légèrement enrichi par rapport à $Ob_{1<:\mu}$, en effet ce calcul constitue un ensemble de fonctionnalités objets greffables sur tout langage aussi bien fonctionnel qu'impératif, ce qui justifie l'ajout d'un langage minimal (dans un soucis de réalisme). Toutefois celui-ci est choisi extrèmement pauvre (entiers et addition) car le but de notre étude n'est pas l'utilisation d'un véritable langage (actuellement impossible ou du moins déraisonnable).

On définit les types, les termes et les valeurs ("résultats" d'une évaluation) du langage obtenu de la manière suivante :

TYPE	::=	*stint*	– type des entiers
		stop	– type maximal pour la relation de sous-typage
		X	– variable de type
		$[LT]$	– type d'un objet
		$\mu(X)B(X)$	– type récursif par rapport à X
TERM	::=	VALUE	– toute "valeur" est un terme
		x	– variable
		$sadd(n_1, n_2)$	– addition d'entiers
		$sinvoc(a, l)$	– appel de la méthode de label l de l'objet (terme) a
		$supdate(a, l, \zeta(x : A)b)$	– remplacement du corps de la méthode de label l du terme a par le terme b
		$sunfold(a)$	– dépliage récursif du terme a
VALUE	::=	*sint* n	– entier
		$[L]$	– objet
		$sfold(A, a)$	– pliage récursif du terme a par rapport au type A

La notation $[L]$ désigne un objet constitué d'une liste L de méthodes. Une méthode est représentée sous la forme d'un label l et du corps $\zeta(x : A)b$ où x représente la variable *self* (liée) dans le code b de la méthode, A étant "le" type de l'objet $[L]$. De la même manière, $[LT]$ contient une liste LT de couples (l, B) où l est un label et B un type.

Les deux constructeurs *sfold* et *sunfold* permettent d'utiliser les types récursifs [2], en effet un tel type $\mu(X)B(X)$ doit être vu comme un point fixe vérifiant $\mu(X)B(X) \approx B[^{\mu(X)B(X)}/_X]$. Pour des raisons opérationnelles, il est préférable de décrire les types récursifs en effectuant le passage d'une forme à l'autre quand cela est nécessaire sans définir de relation d'équivalence générale. Ainsi lors d'un appel ou d'une modification de méthode le terme doit être sous sa forme *objet* de type $B[^{\mu(X)B(X)}/_X]$ mais il est préférable de le garder le reste du temps sous sa forme *repliée* de manière à éviter des dépliages successifs inutiles et à toujours connaître sa forme courante. Ainsi les opérations décrites par ces deux constructeurs sont simplement :

$$\begin{array}{llllll} \textit{si } a & : & \mu(X)B(X) & \textit{alors} & \textit{sunfold}(a) & : & B[^{\mu(X)B(X)}/_X] \\ \textit{si } b & : & B[^{\mu(X)B(X)}/_X] & \textit{alors} & \textit{sfold}(\mu(X)B(X), b) & : & \mu(X)B(X) \end{array}$$

Le problème d'un tel calcul est que, tout comme un λ-terme, un programme même très simple écrit dans ce langage est quasi-illisible. D'où le besoin de définir un langage plus compréhensible pour écrire du code (voir section 6.1) qui soit au ζ-calcul ce que ML est au λ-calcul. Cependant, pour une étude théorique, un calcul formé de peu de constructeurs tout en étant assez expressif est beaucoup plus approprié. Ainsi on constate que les objets sont ici constitués uniquement de méthodes et que les champs doivent être codés eux aussi par des méthodes (ce qui revient en fait à créer une variable *self* inutilisée dans le corps).

Par ailleurs, pour pouvoir définir l'évaluation et le typage du langage, il nous faut définir les environnements associés qui sont des listes de définitions de variables pour l'évaluation :

Environnements d'évaluation :

SVENV	::=	{SVALUING}	– liste de SVALUING
SVALUING	::=	$(x = e)$	– définition de x comme associé au terme e

et des listes de relations de typage et de sous-typage pour le typage :

Environnements de typage :

SENV	::=	{STYPING}	– liste de STYPING
STYPING	::=	$(x : T)$	– définition de x comme étant de type T
		$(X <: T)$	– déclaration de la relation $X <: T$ où X est une variable de type

3.1.2 Evaluation

La sémantique dynamique de ce langage est tout à fait naturelle. Nous considérons ici celle décrite dans [1] qui correspond à une réduction faible où l'on ne réduit pas l'intérieur des objets (considérés comme

des "valeurs d'évaluation"). Le jugement $E \vdash t \hookrightarrow v$ signifie que, dans l'environnement d'évaluation E, le terme t s'évalue en v :

$$SRed\_x \ : \ \frac{E \vdash_{svenv} x = t \quad E \vdash t \hookrightarrow v}{E \vdash x \hookrightarrow v}$$

$$SRed\_int \ : \ E \vdash sint\ n \hookrightarrow sint\ n$$

$$SRed\_add \ : \ \frac{E \vdash t_1 \hookrightarrow sint\ n_1 \quad E \vdash t_2 \hookrightarrow sint\ n_2}{E \vdash sadd(t_1, t_2) \hookrightarrow sint\ n} \quad n = n_1 + n_2$$

$$SRed\_object \ : \ E \vdash [L] \hookrightarrow [L]$$

$$SRed\_invoc \ : \ \frac{E \vdash a \hookrightarrow [L] \quad E \vdash b[^{[L]}/_x] \hookrightarrow v}{E \vdash sinvoc(a, l) \hookrightarrow v} \quad (l, \zeta(x : A)b) \in L$$

$$SRed\_update \ : \ \frac{E \vdash a \hookrightarrow [L]}{E \vdash supdate(a, l, \zeta(x : A)b)) \hookrightarrow [L[^{(l,\zeta(x:A_0)b)}/_{(l,\zeta(x_0:A_0)b_0)}]]} \quad (l, \zeta(x_0 : A_0)b_0) \in L$$

$$SRed\_fold \ : \ \frac{E \vdash a \hookrightarrow v}{E \vdash sfold(T, a) \hookrightarrow sfold(T, v)}$$

$$SRed\_unfold \ : \ \frac{E \vdash a \hookrightarrow sfold(T, v)}{E \vdash sunfold(a) \hookrightarrow v}$$

La règle *SRed_x* ajoutée par rapport à [1] provient du fait que l'on veut pouvoir exécuter des programmes dans un environnement initial où des termes ont été associés à des variables. Elle correspond à la recherche de la valeur d'une variable dans cet environnement. Les autres variables correspondant à l'utilisation de *self* sont traitées par une substitution comme dans la règle *SRed_invoc*.

Remarque : il faut tout de même souligner que l'on ne substitue pas A à A_0 dans la règle *SRed_update*, en effet ceci aurait pour conséquence de modifier le type de l'expression durant l'exécution et invaliderait le *théorème de conservation des types*, il faut ici masquer le type de b par celui de b_0.

3.1.3 Typage

Le choix des règles de typage est très proche de celui fait dans [1] complété par deux nouvelles règles correspondant aux deux nouveaux constructeurs du langage : **sint** et **sadd**. Toutefois, de légères modifications ont été apportées au niveau de la gestion des environnements par la vérification systématique de la validité des enrichissements d'environnements (jugement $\{(x : A).E\} \vdash \diamond$). Ceci permet d'améliorer la réalisation des preuves (section 5.2.4) sans pénaliser l'efficacité de l'exécution (section 3.3). Le jugement $E \vdash t : T$ signifie que, dans l'environnement de typage E, le terme t a le type T :

$$SVal\_x \ : \ \frac{E \vdash_{senv} x : A}{E \vdash x : A}$$

$$SVal\_int \ : \ E \vdash sint\, n : stint$$

$$SVal\_add \ : \ \frac{E \vdash a : stint \quad E \vdash b : stint}{E \vdash sadd(a,b) : stint}$$

$$SVal\_object \ : \ \frac{(\{(x_l : A).E\} \vdash \diamond \quad \{(x_l : A).E\} \vdash b_l : B_l) \quad \forall (l, \zeta(x_l : A) b_l) \in L}{E \vdash [L] : A} \quad A = [l : B_l]_{(l,\zeta(x_l:A)b_l)\in L}$$

$$SVal\_invoc \ : \ \frac{E \vdash a : [LT]}{E \vdash sinvoc(a,l) : B} \quad (l, B) \in LT$$

$$SVal\_update \ : \ \frac{E \vdash a : [LT] \quad \{(x : [LT]).E\} \vdash \diamond \quad \{(x : [LT]).E\} \vdash b : B}{E \vdash supdate(a, l, \zeta(x : [LT])b) : [LT]} \quad (l, B) \in LT$$

$$SVal\_fold \ : \ \frac{E \vdash b : B[^{A}/_{X}]}{E \vdash sfold(A,b) : A} \quad A = \mu(X)B(X)$$

$$SVal\_unfold \ : \ \frac{E \vdash a : A}{E \vdash sunfold(a) : B[^{A}/_{X}]} \quad A = \mu(X)B(X)$$

$$SVal\_subsumption \ : \ \frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

Remarque : le jugement $E \vdash_{senv} x : A$ correspond à l'extraction de la définition $(x : A)$ de l'environnement E par un parcours de liste.

Ce système de règles de typage est totalement non-déterministe du fait de la présence de la règle *SVal_subsumption*. Ce non-déterminisme a des conséquences très importantes en terme d'exécution en Typol en effet, comme nous l'avons déjà vu (section 2.1.3), Typol exécute les systèmes de règles de manière déterministe. Notre système est donc exécutable en Typol mais peut dans bien des cas ne pas terminer alors qu'un typage existe. Dans leur livre, M. Abadi et L. Cardelli décrivent un autre système de règles, appelé $MinOb_{1<:\mu}$, qui est déterministe et donc exécutable sans aucun problème. Cependant, comme nous le verrons en 5.3.1, $MinOb_{1<:\mu}$ pert la propriété de conservation des types pour ne plus vérifier qu'une propriété plus faible.

L'exécutabilité restant une propriété importante d'un langage pour qu'il ne demeure pas purement théorique, on aimerait pouvoir exécuter de manière efficace des systèmes non-déterministes en Typol. Pour cela il conviendrait de déterminer une tactique d'exécution réalisant tout typage possible (en un temps raisonnable), toutefois une telle tactique indépendante du langage objet n'existant probablement pas il faudrait faciliter l'implémentation d'une tactique valable pour tous les programmes d'un langage donné. Une autre possibilité serait de laisser l'utilisateur guider l'application des règles ce qui, par contre, ne peut s'appliquer qu'à de petits programmes...

3.1.4 Sous-typage

La notion de sous-typage qui apparaît dans la règle *SVal_ subsumption* est une caractéristique essentielle de $Ob_{1<:\mu}$ pour la réutilisation de code. La définition adoptée ci-dessous est celle de [1] où le jugement $E \vdash T <: T'$ signifie que, dans l'environnement de typage E, le type T est un sous-type de T' :

$$SSub\_Refl \ : \ \frac{E \vdash A}{E \vdash A <: A}$$

$$SSub\_Trans \ : \ \frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

$$SSub\_Top \ : \ \frac{E \vdash A}{E \vdash A <: stop}$$

$$SSub\_X \ : \ \frac{E \vdash_{senv} X <: T}{E \vdash X <: T}$$

$$SSub\_Object \ : \ \frac{E \vdash [LT_1]}{E \vdash [LT_1] <: [LT_2]} \quad LT_1 \supset LT_2$$

$$SSub\_Rec \ : \ \frac{E \vdash \mu(X)A(X) \quad E \vdash \mu(Y)B(Y) \quad \{(X <: \mu(Y)B(Y)).E\} \vdash A <: B[^{\mu(Y)B(Y)}/_Y]}{E \vdash \mu(X)A(X) <: \mu(Y)B(Y)}$$

Remarque : le jugement $E \vdash_{senv} X <: T$ correspond à la recherche de la relation $(X <: T)$ directement dans l'environnement E par un parcours de liste.

Cette définition mérite quelques précisions, tout d'abord la règle *SSub_ Object* donne une notion assez faible de sous-typage, on aurait pu attendre la relation plus forte donnée par la règle :

$$SSub\_Object' \ : \ \frac{E \vdash LT_1 \quad (E \vdash B_1 <: B_2 \quad \forall (l, B_2) \in LT_2 / (l, B_1) \in LT_1)}{E \vdash [LT_1] <: [LT_2]}$$

mais une telle relation entraîne, sans autre changement du typage, qu'un terme typable peut s'exécuter en un terme non typable (voir section 5.3.2). Par ailleurs, le choix un peu complexe de la règle *SSub_ Rec* est expliqué dans [2], avec ici une légère complication liée à l'implémentation par *sfold* et *sunfold* (voir section 3.1.1).

3.1.5 Substitutions

Dans les différentes règles de typage et d'évaluation apparaît le besoin de définir des substitutions. Or actuellement aucun outil spécifique n'a été développé pour cela en Typol, il est donc nécessaire d'ajouter un autre système de règles. De plus il faut en fait ajouter un système de règles pour chaque type de substitution (expression dans une expression, type dans une expression, ...). La définition d'une substitution impose de donner une règle par constructeur non liant (excepté le cas de la variable qui

necessite toujours deux règles) et deux règles pour un constructeur liant (variable substituée liée ou non à ce nœud de l'arbre d'expression).

On obtient ainsi les règles suivantes dans le cas particulier de la substitution d'une variable de type par un type pour les constructeurs **stint** et **mtype** :

$$Stype\_stype\_int \ : \ stint[^T/_X] = stint$$

$$Stype\_stype\_mtype\_X \ : \ \mu(X)B(X)[^T/_X] = \mu(X)B(X)$$

$$Stype\_stype\_mtype\_notX \ : \ \frac{B(Y)[^T/_X] = Bs(Y)}{\mu(Y)B(Y)[^T/_X] = \mu(Y)Bs(Y)} \quad X \neq Y$$

Hélas de telles règles ne peuvent pas être écrites en Typol qui ne connait aucune écriture fonctionnelle mais uniquement des jugements. On devra donc déclarer :

$$judgement \vdash STYPE \mid STYPE \ / \ STVAR \mid = STYPE \ ;$$

et écrire $\vdash B|T/X| = Bs$ pour $B[^T/_X] = Bs$ (les [ ] étant de plus interdits en Typol car utilisés pour les listes...) au niveau des règles de substitutions. L'absence de fonctions se traduit également de manière assez gênante dans les règles faisant appel à une substitution, ainsi *SVal_fold* s'écrit en Typol :

$$SVal\_fold \ : \ \frac{E \vdash b : Bs \quad \vdash B|A/X| = Bs}{E \vdash sfold(A, b) : A} \quad A = \mu(X)B(X)$$

Il devrait être possible de simplifier grandement ce problème en définissant un système générique de substitutions capable de construire les règles nécessaires à partir de la syntaxe abstraite et de la donnée explicite de la liste des constructeurs liants. L'ajout d'écritures fonctionnelles étant un plus mais posant un problème plus difficile lié au fait que Typol est compilé en Prolog, qui ne travaille que sur des prédicats.

De tels problèmes sont entièrement résolus dans le cadre de la syntaxe abstraite d'ordre supérieur [21, 17, 14] où les substitutions sont remplacées par de simples applications de fonctions.

3.2 Le problème des $\forall$

Comme nous l'avons déjà signalé, les règles présentées plus haut sont en réalité bien loin de celles que l'on peut écrire en Typol. En particulier, il est actuellement impossible de générer une famille de prémisses dans une règle en faisant varier un paramètre (par exemple $\forall x \in X$). L'implémentation d'une telle règle nécessite de rassembler les différentes prémisses sous forme d'un unique jugement que l'on définit par ailleurs sous forme d'un parcours de liste (ce qui nécessite systématiquement au moins deux règles supplémentaires). Ainsi la spécification Typol correspondant à la règle *SVal_object* est en fait :

```
set Typ0_scalc1 is
judgement SENV |- TERM : STYPE;

[...]
```

```
SVal_Object:
   SForall(E, stobject(LT_S) |- * D_S : LT_S)  &  SListOp(|- D_S ~ LT_S)
   ----------------
   E |- sobject(D_S) : stobject(LT_S) ;

[...]

end Typ0_scalc1;

set SListOp is
judgement |- SIGLBLDECL_S ~ SIGTYP_S;

SListOp_equip_void: |- siglbldecl_s[] ~ sigtyp_s[] ;

SListOp_equip_list:
   |- D_S ~ LT_S
   ----------------
   |- siglbldecl_s[D.D_S] ~ sigtyp_s[LT.LT_S] ;
end SListOp;

set SForall is
judgement SENV, STYPE |- SIGLBLDECL : SIGTYP_S;
judgement SENV, STYPE |- * SIGLBLDECL_S : SIGTYP_S;

SForall_singl_t_vt:
   SEnv(senv[styping(x, A).E] |-)  &
   Typ0_scalc1(senv[styping(x, A).E] |- b : B)
   ----------------
   E, A |- siglbldecl(l, sigdecl(x, A, b)) : sigtyp_s[sigtyp(l, B).T_S] ;

SForall_singl_f_vt:
   E, A |- siglbldecl(l, D) : R
   ----------------
   E, A |- siglbldecl(l, D) : sigtyp_s[sigtyp(l1, B).R] ;
            provided DIFF_SLBL(l, l1);

SForall_void_vt: E, A |- * siglbldecl_s[] : L ;

SForall_list_vt:
   E, A |- LD : L  &  E, A |- * D_S : L
   ----------------
```

```
    E, A |- * siglbldecl_s[LD.D_S] : L ;
end SForall;
```

Les prémisses de la règle principale *SVal_Object* prennent la forme de deux jugements. Le premier ($SForall(E, stobject(LT\_S) \vdash *D\_S : LT\_S)$) signifie que D_S est de type LT_S au sens où chaque couple $(l, \zeta(x : A)b)$ de D_S vérifie la propriété suivante : *on peut typer le corps b en B associé au même label l dans la liste LT_S*, comme cela apparaît dans la règle *SForall_singl_t_vt*. Il s'agit uniquement d'une propriété d'inclusion (car il peut exister dans LT_S des labels qui n'apparaissent pas dans D_S). Le second jugement ($SListOp(\vdash D\_S \sim LT\_S)$) est donc nécessaire pour vérifier que les deux listes ont bien le même nombre d'éléments.

Devoir écrire de telles règles rend les choses peu lisibles et la multiplication des jugements amène des problèmes pour les preuves (voir section 5.2.3). Il est donc indispensable de pouvoir obtenir des règles plus synthétiques et ceci passe par l'enrichissement de Typol. Une solution pour résoudre ce problème ainsi que d'autres liés à la nécessité de définir de nombreux parcours de listes (appel de méthode, recherche dans un environnement, ...) consiste à définir dans un Typol enrichi non plus uniquement la structure de liste mais également celle de dictionnaire à clef unique et celle d'ensemble.

Un dictionnaire à clef unique est constitué de triplets (clef signe valeur) dont toutes les clefs sont distinctes pour un signe donné et muni de deux fonctions :

- l'extraction par clef sous la forme $D.(c) = v$ par exemple où D est un dictionnaire, c est une clef, $:$ un signe et v est l'unique valeur associée à cette clef pour ce signe dans D
- l'insertion d'un nouveau triplet qui peut s'écrire $D + (c : v)$ où D est un dictionnaire et $(c : v)$ un triplet (*clef signe valeur*) dont la clef n'apparaît pas dans D avec ce signe.

Une notion de dictionnaire est déjà implantée dans Typol mais sous une forme assez externe (modules séparés) qui interdit l'utilisation des notations décrites ci-dessus qui assureraient une bien meilleure lisibilité comme c'est le cas pour les listes. Par ailleurs, elle ne permet que la définition de couples et donc que d'un unique type de relation dans un même dictionnaire, toutefois elle permet de coder notre définition en regroupant clef et signe de notre dictionnaire comme étant la clef du dictionnaire de couples.

La notion d'ensemble qu'il serait également utile d'ajouter au Typol actuel est essentiellement la possibilité de s'affranchir de la contrainte d'ordre donnée par les listes. Par ailleurs pour pouvoir construire ces ensembles, certaines notions prédéfinies sont nécessaires :

- $E_1 \cup E_2$ réunion des deux ensembles E_1 et E_2
- $E_1 \cap E_2$ intersection des ensembles E_1 et E_2
- $E_1 \subset E_2$ inclusion de E_1 dans E_2
- $E_1 \setminus E_2$ différence ensembliste

Il convient alors de définir le lien entre ces deux structures sous la forme d'une fonction $lbl(D)$ associant à un dictionnaire l'ensemble de ses clefs.

Pour des raisons essentielles de cohérence, ces deux nouvelles notions doivent être, comme le sont les listes actuelles, totalement typées. L'utilisateur devra définir un sur-type commun à toutes les clefs qu'il désire utiliser dans un même dictionnaire de même que pour les signes et les valeurs correspondants ou pour les objets apparaissant dans un même ensemble.

On est ensuite en mesure de définir l'outil le plus important basé sur toutes ces constructions et capable de représenter une famille de prémisses dans une règle :

$$\frac{\ldots \quad \& \quad (\quad \textit{prémisse} \quad \& \quad \ldots \quad \& \quad \textit{prémisse} \quad) \quad \forall x \in E \quad \& \quad \ldots}{\textit{conclusion}}$$

où E est un ensemble (du type décrit précedemment) construit par exemple à partir des clefs d'un dictionnaire et où x est un paramètre apparaissant dans les différentes prémisses concernées.

Une fois tout ceci ajouté à Typol, on pourra se rapprocher des écritures plus naturelles de la section précédente et la règle *SVal_object* deviendra :

$$\textit{SVal\_object} \ : \ \frac{E \vdash [LT] \quad \& \quad (L_{=}(l) = \zeta(x : [LT])b \quad \& \quad \{E + (x : [LT])\} \vdash b : LT_{:}(l)) \quad \forall l \in lbl(L)}{E \vdash [L] : [LT]}$$

où L, LT et E sont implémentés par des dictionnaires. De plus on admet ici la possibilité d'écrire directement $LT_{:}(l)$ dans un jugement sans passer par un jugement supplémentaire $\vdash LT_{:}(l) = B$ ce qui rejoint le problème des écritures fonctionnelles vu à la section 3.1.5.

3.3 Gestion des environnements

Nous considérons, comme cela est apparu dans les différents systèmes de règles de la section 3.1, des environnements sous forme de listes formées soit de définitions de variables soit de relations de sous-typage.

On peut ensuite définir la validité de ces environnements, en prenant par exemple le cas d'un environnement de typage. Contrairement à [1], on définit ici deux relations de validité. La première, validité forte ($\vdash E$), correspond à la validité de l'environnement tout entier (utilisée dans [1]) alors que la seconde, validité faible ($E \vdash \diamond$), n'assure que la validité du dernier ajout par rapport au reste de l'environnement.

$$\textit{SEnv\_good\_void} \ : \ \vdash \{\}$$

$$\textit{SEnv\_good\_list} \ : \ \frac{\{t.E\} \vdash \diamond \quad \vdash E}{\vdash \{t.E\}}$$

$$\textit{SEnv\_typing} \ : \ \frac{E \vdash T \quad x \notin E}{\{(x : T).E\} \vdash \diamond}$$

$$\textit{SEnv\_subtyp} \ : \ \frac{E \vdash T \quad X \notin E}{\{(X <: T).E\} \vdash \diamond}$$

où les relations "$\notin$" se définissent par parcours de liste.

Il nous faut par ailleurs définir la possibilité d'extraire un élément d'un environnement. Cette définition est tout à fait caractéristique des définitions par parcours de liste qui nécessitent toujours deux règles.

$$SEnv\_typing\_t \ : \ \{(x:T).E\} \vdash_{senv} x:T$$

$$SEnv\_typing\_f \ : \ \frac{E \vdash_{senv} x:T}{\{t.E\} \vdash_{senv} x:T}$$

et de même pour l'extraction de relations de sous-typage.

Pour des raisons d'efficacité lors de l'exécution, les règles de typage sont conçues pour n'effectuer qu'une seule fois chaque vérification. On teste donc la validité forte de l'environnement initial et une validité faible à chaque fois que l'environnement est enrichi comme dans la règle *SVal_object* par exemple.

Cette manière d'opérer diffère de celle décrite dans [1] où les vérifications de validité sont effectuées à chaque recherche dans l'environnement ce qui entraîne de nombreuses redondances et ralentit beaucoup l'exécution.

On constate à nouveau la lourdeur des règles à écrire en Typol pour coder ce qui précéde. En effet il faut définir l'extraction pour toutes les structures possibles mises dans l'environnement, définir la relation "$\notin$" dans tous les cas également, ... ce qui entraîne une multiplication du nombre de règles.

On constate alors que la notion de dictionnaire définie à la section 3.2 s'applique très bien à nos environnements et permet d'obtenir directement les notions d'extraction d'élément et d'insertion valide.

4 Traduction vers Coq

La définition de formalismes et de règles de typage ou d'évaluation en Typol permet leur "prototypage" grâce à l'environnement Centaur, cependant on désire également effectuer des preuves sur ces Sémantiques Naturelles. Pour cela D. Terrasse-Kaplan a défini un outil de traduction [22] des langages Métal et Typol vers le système d'aide à la preuve Coq dont elle a prouvé la correction.

4.1 Exemple de traduction

Revenons au micro-langage défini dans la section 2, le programme Métal que nous avions défini se traduit en Coq par un ensemble de types inductifs (mutuellement inductifs si les besoins de la définition le nécessitent) :

```
Mutual Inductive
   EXP: Set :=
        add: EXP -> EXP ->EXP
       | int: nat ->EXP.

Mutual Inductive
```

```
TYPE: Set :=
     tint: TYPE.
```

On peut ensuite traduire les différents programmes Typol en Coq où les jugements apparaissent sous forme de prédicats et où les règles sont regroupées selon leur jugement de conclusion et apparaissent comme des constructeurs du prédicat associé à ce jugement :

```
Mutual Inductive
   eval_L: EXP -> EXP ->Prop :=
        Red_int: (n:nat)(eval_L (int n) (int n))
       | Red_add:
           (e2, e1:EXP)
           (n2, n, n1:nat)
           (eval_L e1 (int n1)) -> (eval_L e2 (int n2)) -> (sum n1 n2 n) ->
           (eval_L (add e1 e2) (int n)).

Mutual Inductive
   typ_L: EXP -> TYPE ->Prop :=
        Val_int: (n:nat)(typ_L (int n) tint)
       | Val_add:
           (b, a:EXP) (typ_L a tint) -> (typ_L b tint) ->
           (typ_L (add a b) tint).
```

La porte est ainsi ouverte à de nombreuses preuves sur la Sémantique Naturelle de notre micro-langage...

4.2 Quelques défauts

La version actuelle de la traduction possède quelques lacunes, en particulier en ce qui concerne les éléments "périphériques" à Typol. Les imports déclarés en Typol par *use* ou *import* ne sont absolument pas traduits alors qu'ils correspondent très directement à l'instruction *Require* de Coq, le graphe des dépendances entre programmes Typol se retrouvant à l'identique entre les fichiers Coq. Un seul cas fait exception, celui des imports de prédicats Prolog qui doivent être à nouveau définis dans Coq alors que les quelques prédicats de base déjà définis pour Typol devraient être traduits automatiquement vers des définitions correspondantes en Coq.

La traduction des listes de Typol pose également un problème car une nouvelle définition des listes en Coq est générée à chaque fichier traduit. Il faut alors définir une unique notion de liste et remplacer les définitions inutiles par un appel à la définition commune.

La maintenance de la traduction est rendue difficile par l'évolution constante de la syntaxe *vernac* de Coq. Ainsi la possibilité de déclarer des opérateurs de coercion n'est pas prise en compte et nécessite une modification par l'utilisateur des fichiers générés. Le principal problème lié à toutes ces modifications manuelles n'est pas leur difficulté mais le fait qu'elles doivent être répétées à chaque nouvelle traduction, en particulier lors d'une modification des programmes Typol.

De manière à ne pas obtenir de prédicats de même nom provenants de fichiers indépendants, la traduction génère (s'il n'existe pas) un fichier de renommage des prédicats mais, à chaque évolution du Typol il est nécessaire d'enrichir ce fichier, il faudrait donc une génération quasi-automatique des noms en demandant par exemple à l'utilisateur un préfixe associé à chaque programme Typol et en génerant les noms sous la forme : *[préfixe]_[nom de set]_[numéro de jugement]*.

4.3 Enrichissement pour les $\forall$

Comme tout constructeur de la syntaxe de Typol, les nouvelles notions introduites en 3.2 doivent être traduites en Coq. G. Kahn a déjà défini en Coq les notions d'ensemble, d'appartenance, d'inclusion, ... Il suffirait donc de définir les dictionnaires (de manière assez similaire) et de traduire les structures Typol dans les structures Coq correspondantes. La traduction de la construction $\forall$ s'effectuant alors tout simplement vers l'équivalent Coq par :

$$\frac{\ldots \& (P_1 \& \ldots \& P_n)\ \forall x \in E\ \& \ldots}{C} \quad \rightsquigarrow \quad \ldots \Rightarrow (\forall x, x \in E \Rightarrow P_1 \wedge \cdots \wedge P_n) \Rightarrow \ldots \Rightarrow C.$$

L'utilisateur devrait de plus pouvoir disposer d'un module Coq de lemmes prédémontrés sur ces structures comme par exemple :

$$\forall D : \mathit{dict}(\mathit{LABEL}, \mathit{SIGNE}, \mathit{VALEUR}), \forall l : \mathit{LABEL}, \forall s : \mathit{SIGNE}, \forall V_1 : \mathit{VALEUR}, \forall V_2 : \mathit{VALEUR},$$

$$D_s(l) = V_1 \Rightarrow D_s(l) = V_2 \Rightarrow V_1 = V_2$$

où $\mathit{dict}(\mathit{LABEL}, \mathit{SIGNE}, \mathit{VALEUR})$ serait le type des dictionnaires dont les clefs ont le type *LABEL*, les signes le type *SIGNE* et les valeurs le type *VALEUR*.

Remarque : de la même manière il serait agréable de disposer en Coq d'un système de substitutions vers lequel on puisse traduire directement celles "définies" en Typol à la section 3.1.5 (en attendant de pouvoir utiliser la syntaxe abstraite d'ordre supérieur).

5 Preuve de la conservation des types (SRT)

A partir des définitions obtenues par la traduction des programmes décrivant la syntaxe et la sémantique de $Ob_{1<:\mu}$, nous allons pouvoir utiliser le système Coq pour prouver des propriétés de cette sémantique. Nous nous intéressons ici à la preuve du théorème de conservation des types (pour d'autres preuves voir 6).

5.1 Le théorème de conservation des types

Le théorème de conservation des types (Subject Reduction Theorem) que nous avons démontré est proche de celui énoncé dans [1], toutefois nous considérons des environnements initiaux non vides de manière à modéliser la possibilité d'avoir des définitions provenant de modules extérieurs au programme.

Théorème 5.1 *Soient E un environnement d'exécution, e un programme du langage $Ob_{1<:\mu}$, v une valeur et T un type alors si $E \vdash e \hookrightarrow v$ et si $E \vdash e : T$ on a $E \vdash v : T$.*

Pour obtenir ce résultat, on passe en fait par un résultat intermédiaire, plus "proche du typage", où l'on explicite la traduction de l'environnement d'exécution en un environnement de typage définie par le système de règles suivant :

$$STypEnv\_void \ : \ \{\} \rightsquigarrow \{\}$$

$$STypEnv\_list\_valuing \ : \ \frac{E \rightsquigarrow E' \quad E \vdash e : T}{\{(x = e).E\} \rightsquigarrow \{(x : T).E'\}}$$

Proposition 5.2 *Soient E un environnement d'exécution, E' un environnement de typage, e un programme du langage $Ob_{1<:\mu}$, v une valeur et T un type alors si $E \vdash e \hookrightarrow v$ et si $E' \vdash e : T$ (avec $E \rightsquigarrow E'$ et $\vdash E'$) on a $E' \vdash v : T$.*

Démonstration : La preuve de ce théorème se fait par induction sur les règles définissant le jugement $E \vdash e \hookrightarrow v$. Cette induction donne huit cas à traiter, chacun se faisant lui-même par induction sur les règles ayant pu produire le jugement $E' \vdash e : T$. Le fait de considérer des environnements quelconques modifie peu les démonstrations faites dans [1], cependant cela ajoute le cas de l'évaluation d'une variable définie dans l'environnement. Par ailleurs nous avons enrichi le langage avec *sint* et *sadd* avec pour chacun une règle d'évaluation, ce qui génère encore deux cas supplémentaires par rapport à [1].

- Le cas d'une variable est assez caractéristique du type de raisonnements faits en Coq. Le fait $E \vdash x \hookrightarrow v$ provient nécessairement de l'application de la règle *SRed_x* donc il existe un terme t tel que $E \vdash_{svenv} x = t$ et $E \vdash t \hookrightarrow v$. Par hypothèse, $E \rightsquigarrow E'$ et $E' \vdash x : T$ on raisonne alors par induction sur ce dernier jugement qui provient de $E' \vdash_{senv} x : T$ par application de la règle *SVal_x* ou de $E' \vdash x : T'$ et $E' \vdash T' <: T$ par l'application de *SVal_subsumption*. Dans le premier cas, on a $(x : T) \in E'$ (par $E' \vdash_{senv} x : T$) or la traduction des environnements d'exécution vers les environnements de typage $E \rightsquigarrow E'$ se faisant par la règle *STypEnv_list_valuing* qui conserve les labels, il existe nécessairement un terme t' tel que $(x = t') \in E$ (d' où $E \vdash_{svenv} x = t'$), avec de plus $E \vdash t' : T$. Les clefs de E' étant uniques ($\vdash E'$) et provenant de celles de E, on a, par unicité dans E, $t = t'$ et donc $E \vdash t : T$. Alors par hypothèse d'induction $E' \vdash v : T$. Dans le second cas, on a $E' \vdash x : T'$ et $E' \vdash T' <: T$ or, par hypothèse d'induction sur $E' \vdash x : T$, on sait que $E' \vdash v : T'$ et donc par *SVal_subsumption*: $E' \vdash v : T$.
- Le cas de *sint* est trivial car *sint n* s'évalue en lui-même et donc $e = v$.
- Le cas de *sadd* est à peine plus complexe mais nécessite l'application du raisonnement par induction effectué pour les variables.

5.2 Difficultés de la preuve en Coq

La réalisation d'une telle preuve en Coq, qui offre l'assurance d'un raisonnement entièrement juste, apporte de nombreuses difficultés qui sont bien loin d'apparaître lors d'une preuve sur papier. En effet Coq est bien adapté (à quelques nuances près : voir 5.2.2) aux démonstrations par induction mais beaucoup moins à d'autres types de raisonnements (notamment les problèmes de réécriture). La preuve du théorème

de conservation des types réalisée en Coq est complète à l'exception de quelques lemmes que les outils actuels rendent très fastidieux à démontrer et des lemmes de substitution (non démontrés dans [1]) qui sont essentiellement techniques et n'apporteraient rien de nouveau par rapport au reste de la preuve. On trouvera les énoncés des lemmes utilisés (dont certains sont laissés en axiomes) pour organiser la preuve de *SRT* en annexe A.

5.2.1 Modification du raisonnement

La preuve du théorème 5.1 que nous avons donnée est volontairement proche de celle faite en Coq pour ce qui est du schéma de raisonnement. La preuve donnée dans [1] est plus naturelle mais n'est pas réalisable en Coq car elle utilise comme schéma d'induction un schéma un peu éloigné de l'application directe du système de règles. On aimerait pourtant pouvoir réaliser une preuve de la forme :

- Le fait $E \vdash x \hookrightarrow v$ provient nécessairement de l'application de la règle $SRed\_x$ donc il existe un terme t tel que $E \vdash_{svenv} x = t$ et $E \vdash t \hookrightarrow v$. Par hypothèse, $E \rightsquigarrow E'$ et $E' \vdash x : T$ **ce qui provient de** $E' \vdash_{senv} x : T'$ **par application de la règle** $SVal\_x$ **suivie d'éventuelles applications de** $SVal\_subsumption$ **avec** $E' \vdash T' <: T$.

  [...]

  Il reste ensuite à appliquer $SVal\_subsumption$ **pour obtenir** $E' \vdash v : T$.

L'utilisation d'un tel principe d'induction nécessite de donner un nouveau système de typage pour lequel on fait jouer un rôle particulier à la règle $SVal\_subsumption$ (qui s'intègre au niveau de chacune des autres règles) et dont on prouve l'équivalence avec celui que nous avons utilisé. Obtenir une preuve du théorème plus agréable et plus naturelle impose alors en contre-partie la preuve de cette équivalence. Toutefois le système de typage initial avec une règle de *subsumption* qui vient s'ajouter aux autres étant usuel pour les langages à objets, on peut espérer savoir abstraire suffisament l'équivalence entre les deux systèmes pour en déduire une preuve générale qui s'applique aux différents langages à objets (pour un premier pas dans cette direction voir annexe B).

5.2.2 Autres problèmes liés à Coq

Les preuves en Coq sont basées sur l'application de tactiques utilisant des noms donnés arbitrairement aux objets et aux différentes hypothèses manipulés sans considération de leur nature exacte. La réutilisation de parties de preuves pour des cas extrèmement similaires ou après une modification même insignifiante des définitions ou du contexte est ainsi rendue difficile car nécessitant la modification des références contenues dans presque toutes les tactiques.

La tactique *Elim* permet de réaliser des raisonnements par induction sur les constructeurs d'un prédicat et génère donc les hypothèses appropriées. Toutefois la diversité des types d'induction utilisés sur papier fait qu'il n'est pas toujours facile de faire générer l'hypothèse d'induction recherchée. En particulier le principal défaut de cette tactique sous sa forme actuelle est son incapacité à gérer les jugements partiellement (ou totalement) instanciés.

Ainsi dans la preuve du cas *sadd* du théorème 5.1 apparaît l'hypothèse $E' \vdash sadd(t_1, t_2) : T$ sur laquelle on désire effectuer une induction. L'application de la tactique *Elim* génère une hypothèse d'induction inutilisable pour la suite du raisonnement et des buts non prouvables. Pour obtenir la bonne hypothèse il est alors nécessaire d'attribuer une notation t au terme $sadd(t_1, t_2)$ pour obtenir un jugement "factorisé" $E' \vdash t : T$ sur lequel le comportement de *Elim* sera ensuite valable. La réalisation de ce type de manipulations nécessite l'application d'une quinzaine de tactiques et augmente rapidement le nombre d'hypothèses à gérer lors de chaque utilisation correcte de *Elim*, les preuves se trouvent surchargées inutilement et beaucoup plus fastidieuses à réaliser.

Par ailleurs la tactique *Elim* génère un cas par constructeur du prédicat concerné même si plusieurs n'ont pas lieu d'être (on retrouve ici la mauvaise gestion des informations contenues dans le jugement) alors que la tactique *Inversion* est tout à fait capable d'utiliser tous les renseignements qui lui sont fournis et ne génère que les cas nécessaires. Cependant l'utilisation d'*Elim* reste dans bien des cas indispensable car elle est la seule tactique qui permette de générer des hypothèses d'induction.

Les principes d'induction générés par Coq ne sont malheureusement pas toujours suffisamment puissants pour réaliser certaines preuves. En effet en cas de récursivité mutuelle entre deux prédicats $\mathcal{P}_1$ et $\mathcal{P}_2$, le principe d'induction généré par Coq ne permet pas, lors de la preuve d'une propriété P de $\mathcal{P}_1$, d'utiliser l'hypothèse que celle-ci est vraie pour les occurences de $\mathcal{P}_1$ dans $\mathcal{P}_2$ non visibles à travers l'apparition de $\mathcal{P}_2$ dans $\mathcal{P}_1$. La commande *Scheme* permet de résoudre ce problème en demandant à l'utilisateur une propriété P' à vérifier sur $\mathcal{P}_2$ et en demandant de prouver que *(pour toute occurence de* $\mathcal{P}_2$ *dans* $\mathcal{P}_1$, $P'(\mathcal{P}_2)) \Rightarrow P(\mathcal{P}_1)$ et *(pour toute occurence de* $\mathcal{P}_1$ *dans* $\mathcal{P}_2$, $P(\mathcal{P}_1)) \Rightarrow P'(\mathcal{P}_2)$ mais son utilisation est assez lourde et rend le cas de trois (ou plus) prédicats mutuellement récursifs (par exemple $\mathcal{P}_1(\mathcal{P}_2)$, $\mathcal{P}_2(\mathcal{P}_3)$, $\mathcal{P}_3(\mathcal{P}_1)$) véritablement déraisonnable.

5.2.3 Problèmes liés aux ∀

L'utilisation de règles complexes en Typol (en l'absence de construction ∀) se traduit essentiellement en Coq par une amplification très rapide des problèmes cités précédemment. Des implémentations de règles comme celles décrites à la section 3.2 génèrent systématiquement des problèmes de prédicats mutuellement récursifs qui disparaîtraient par l'introduction des notions définies en 3.2 grâce à la diminution du nombre de jugements à considérer. Ainsi à l'heure actuelle, certaines preuves sont rendues irréalisables par l'apparition de systèmes de trois prédicats mutuellement récursifs.

Même lorsque le recours à *Scheme* n'est pas obligatoire, la multiplication des prédicats due à la définition de nombreux parcours de listes en Typol (pour simuler des ensembles) entraîne dans les preuves la nécessité de réaliser de nombreuses inductions souvent emboîtées les unes dans les autres ce qui trouble beaucoup la compréhension du déroulement de la preuve.

5.2.4 Problèmes liés aux environnements

Tel qu'il est décrit dans [1] le système de typage permet très mal de véhiculer à travers une preuve les informations de validité sur les environnements car ce sont des propriétés qui n'apparaissent qu'à l'appel de variable dans l'environnement. Les modifications apportées à ces règles dans la section 3.1 s'avèrent à nouveau utiles car la donnée initiale de validité de l'environnement de départ complétée par les validités

de chaque enrichissement permet de reconstruire assez facilement la validité de l'environnement courant à chaque étape du raisonnement (en effet, si l'hypothèse d'induction est bien formulée, il n'apparaît jamais plus d'un ou deux enrichissements à gérer).

Le problème de l'α-conversion n'ayant pas été traité ici, il apparaît le besoin de vérifier que les enrichissements d'environnement qui pourront apparaître lors du typage n'amènent pas une redéfinition d'un variable existante (refusée par la validité d'environnement). Pour résoudre ce problème sans ajouter l'α-conversion, il nous faudrait définir les variables d'un terme et vérifier avant tout typage que l'environnement initial ne contient aucune des variables du terme à typer ce qui rend les contraintes sur l'environnement dépendantes du terme étudier, chose très peu naturelle.

L'existence d'un ordre sur les environnements imposé par les listes surchage énormément les preuves par la nécessité de nombreuses manipulations qui disparaîtraient totalement dans le cadre suggéré en 3.3 où l'ordre n'apparaît plus. Sans cela on assiste à une véritable explosion du nombre d'inductions emboîtées les unes dans les autres. Le cas *svar* du théorème 5.1 est extrèmement représentatif: la partie centrale de la démonstration où l'on transfère des propriétés de l'environnement d'exécution dans celui de typage grâce à la définition de la traduction de l'un vers l'autre apporte un système d'inductions emboîtées beaucoup trop complexe par rapport à ce que les notions d'ensemble et d'appartenance permettraient de faire.

Remarque : une preuve réalisée en Coq n'est qu'une suite de tactiques et se trouve donc illisible. Il est possible d'en extraire automatiquement un texte [10] assez joli pour de petites preuves en mathématiques. Malheureusement un tel texte s'avère pour l'instant beaucoup trop long et peu facile à suivre dans les cas qui nous concernent, où apparaissent de grosses hypothèses d'induction elles-mêmes très peu lisibles.

5.3 Stabilité du théorème par rapport aux règles

Une fois le théorème 5.1 démontré, on peut s'intéresser aux conséquences sur ce théorème de certaines modifications des systèmes de règles en vue de corriger différents défauts énoncés plus haut (section 3.1).

5.3.1 Déterminisme

Comme nous l'avons vu à la section 3.1.3, on serait fortement intéressé par l'utilisation d'un système déterministe de règles de typage. Le système $MinOb_{1<:\mu}$ décrit dans [1] s'implémente en Typol de manière assez proche de $Ob_{1<:\mu}$ mais ne permet pas de conserver la propriété de conservation des types mais seulement une propriété plus faible :

Théorème 5.3 *Soient E un environnement d'exécution et e un programme, si $E \vdash e \hookrightarrow v$ et si $E \vdash_{Min} e : T$, alors $E \vdash_{Min} v : T'$ avec $E \vdash T' <: T$.*

Remarque : tous les jugements de typage et d'évaluation qui apparaissent dans l'énoncé de ce théorème sont déterministes.

Dans [1], M. Abadi et L. Cardelli énoncent les propriétés suivantes de $MinOb_{1<:\mu}$:

Proposition 5.4 *Soient E un environnement d'exécution, e un programme et T et T' deux types, on a :*

$$\begin{array}{lll} E \vdash_{Min} e : T & \Rightarrow & E \vdash e : T \qquad (i) \\ E \vdash e : T & \Rightarrow & E \vdash_{Min} e : T' \wedge E \vdash T' <: T \qquad (ii) \end{array}$$

Démonstration (5.3) : A partir de ces énoncés et du théorème 5.1, la démonstration est très facile. En effet si $E \vdash_{Min} e : T$ alors par application de *(i)* on a $E \vdash e : T$ d'où, d'après le théorème 5.1, $E \vdash v : T$. Et enfin *(ii)* nous donne $E \vdash_{Min} v : T'$ avec $E \vdash T' <: T$.

Il reste alors à prouver qu'il existe des cas où cette inégalité peut être stricte pour prouver que les types ne sont effectivement pas toujours conservés par l'évaluation.

Proposition 5.5 *Soient E un environnement d'exécution, e un programme, v une valeur et T un type alors :*

$$E \vdash e \hookrightarrow v \ \wedge \ E \vdash_{Min} e : T \quad \not\Rightarrow \quad E \vdash_{Min} v : T.$$

Remarque : pour des raisons de lisibilité nous utiliserons dans cette démonstration et dans les suivantes la notation $o.l$ pour $sinvoc(o, l)$.

Démonstration : Soient E un environnement et A et A' deux types tels que $E \vdash A' <: A$ avec $A' \neq A$ (par exemple $A' = stint$ et $A = stop$). Soit a un terme tel que $E \vdash a : A'$, on définit le terme ε par $\varepsilon = [l = \zeta(x : T)a]$ où x n'apparaît pas dans a et où $T = [l : A]$. On va montrer que le type de $\varepsilon.l$ diminue strictement lors de l'évaluation. En effet, par l'application de *SVal_ subsumption* et de *SVal_ object*, $E \vdash \varepsilon : T$ donc si l'on applique *SVal_ invoc* on obtient $E \vdash \varepsilon.l : A$. Par ailleurs soit v la valeur d'évaluation de a dans E, raisonnons par l'absurde en supposant que a vérifie la conservation des types alors $E \vdash a : A' \ \Rightarrow \ E \vdash v : A'$ mais dans ce cas *SRed_ invoc* donne $E \vdash \varepsilon.l \hookrightarrow v$ d'où contradiction car $E \vdash \varepsilon.l : A$ et $E \vdash v : A'$ avec $A' \neq A$.

Remarque : le problème vient en fait de la règle *SVal_ object* où l'on masque le véritable type de b par un type plus grand sans véritablement changer son type ce qui entraîne que si, lors de l'exécution, b est extrait de l'objet il retrouve son véritable type. Une solution à ce problème serait de changer effectivement le type de b mais cela nécessiterait d'être fait par induction sur la structure de b ce qui compliquerait énormément le système de règles (surtout dans le cas d'une relation de sous-typage plus complexe que celle présentée ici).

5.3.2 Relation de sous-typage

Revenons au système $Ob_{1<\mu}$, la relation de sous-typage utilisée (voir 3.1.4) peut paraître faible en ce qui concerne les objets. Toutefois la seule modification de la règle *SSub_ Object* en *SSub_ Object'* a des conséquences catastrophiques sur le typage.

Proposition 5.6 *Soient E un environnement d'exécution, e un programme, v une valeur et T un type alors :*

$$E \vdash e \hookrightarrow v \ \wedge \ E \vdash_{<:'} e : T \quad \not\Rightarrow \quad \exists T' \ E \vdash_{<:'} v : T'.$$

Remarque : les jugements $\vdash_{<:'}$ correspondent au système $Ob_{1<:\mu}$ avec la règle *SSub_ Object'*.
Pour des raisons de lisibilité nous utiliserons dans cette démonstration la notation $o.l \leftleftarrows \zeta(x : T)b$ pour $supdate(o, l, \zeta(x : T)b)$.

Démonstration : Soient E un environnement, A et B deux types et a et b deux termes tels que $E \vdash_{<:'} A <: B$ avec $A \neq B$ et $E \vdash_{<:'} a : A$ et $E \vdash_{<:'} b : B$ mais $E \not\vdash_{<:'} b : A$. On définit alors $\varepsilon = [l = \zeta(x : T')a]$ où x n'apparaît ni dans a ni dans b et $T' = [l : A]$. Considérons le terme $\varepsilon.l \leftleftarrows \zeta(x : T)b$ où $T = [l : B]$ vérifie $E \vdash_{<:'} T' <: T$ d'après *SSub_ Object'*. Par *SVal_ object* on a $E \vdash_{<:'} \varepsilon : T'$ donc $E \vdash_{<:'} \varepsilon : T$ par *SVal_ subsumption*. L'application de *SVal_ update* prouve alors que $E \vdash_{<:'} \varepsilon.l \leftleftarrows \zeta(x : T)b : T$. Par ailleurs, l'évaluation de ce terme par *SRed_ update* donne $E \vdash \varepsilon.l \leftleftarrows \zeta(x : T)b \hookrightarrow [l = \zeta(x : T')b]$. Or ce dernier terme n'est pas typable car cela nécessiterait qu'il soit typable par la règle *SVal_ object* et donc que $E \vdash_{<:'} b : A$.

On constate une certaine instabilité du système de règles $Ob_{1<:\mu}$ vis à vis de la propriété de conservation des types. Il faut donc choisir entre une propriété forte de conservation des types et un langage aux propriétés plus intéressantes (déterminisme, sous-typage plus général, ...). Ainsi comme nous l'avons vu le système $MinOb_{1<:\mu}$ permet de prouver la proposition 5.3 et de la même manière une modification des règles plus importantes que la simple modification de la relation de sous-typage doit permettre de prouver une propriété similaire (voir [11]).

6 Vers d'autres preuves

Comme nous l'avons déjà signalé (section 3.1.1), le ζ-calcul $Ob_{1<:\mu}$ ne permet pas l'écriture de vrais programmes pour des raisons de lisibilité. Pour résoudre ce problème, M. Abadi et L. Cardelli ont décrit dans [1] le langage *O -1* plus haut niveau que $Ob_{1<:\mu}$. Nous allons nous intéresser à la possibilité de traduire *O -1* vers $Ob_{1<:\mu}$ et d'en déduire la propriété de conservation des types sur *O -1* à partir de celle que nous avons prouvée pour $Ob_{1<:\mu}$.

6.1 Le langage *O -1*

Nous reprenons ici, pour la description de la syntaxe de *O -1* et des systèmes de règles, la syntaxe utilisée dans la section 3.1 pour $Ob_{1<:\mu}$.

6.1.1 Syntaxe

Pour pouvoir traduire notre langage vers $Ob_{1<:\mu}$ qui ne contient pas de fonctions, nous ne considérons pas le langage *O -1* dans son intégralité mais uniquement les fonctionnalités objets en oubliant les classes. Par contre, pour obtenir un véritable langage, nous l'enrichissons avec les constructeurs *int*, *add* et *let*.

TYPE	::=	*tint*	– type des entiers
		top	– type maximal pour la relation de sous-typage
		X	– variable de type
		tobject$(X)[LT]$	– type d'un objet
TERM	::=	VALUE	– toute "valeur" est un terme
		x	– variable
		add(n_1, n_2)	– addition d'entiers
		let $x : T = e_1$ *in* e_2	– association du nom x au terme e_1 de type T ensuite utilisable dans e_2
		invoc(a, l)	– appel de la méthode de label l de l'objet (terme) a
		upfield$(a : A, l, b)$	– remplacement du champ de label l du terme a par le terme b
		upmethod$(a, l, (x : A)b)$	– remplacement du corps de la méthode de label l du terme a par le terme b
VALUE	::=	*int* n	– entier
		object$(x : T)[L]$	– objet

Remarque : les environnements restent inchangés.

Dans la notation *object*$(x : T)[L]$ associée aux objets, L est la liste des champs et des méthodes de l'objet donc une liste de couples (l, b) où l est un label et b est un terme (soit le corps d'une méthode soit la valeur d'un champ) par ailleurs x est le nom donné à *self*. De la même manière, dans *tobject*$(X)[LT]$, LT est la liste des couples formés par l'association des labels avec les types correspondants et X permet d'utiliser le type global de l'objet à l'intérieur de sa définition (type récursif) en particulier pour typer les utilisations de *self*.

La principale faiblesse de cette syntaxe est l'apparition répétitive de types, souvent complexes à cause des objets. On aimerait donc pouvoir associer un nom à un type comme le constructeur *let* le fait pour les expressions. Une telle construction de la forme :

$$letype\ X = T\ in \ldots$$

apporte de très nombreuses complications pour la définition du typage et de l'exécution. En effet lorsque l'on désire savoir si un type est de la forme *tobject*$(X)[LT]$ par exemple, deux cas se présentent : soit le type est effectivement de cette forme soit le type est un nom de type qui peut être de cette forme (et nécessite une recherche dans l'environnement pour le savoir). Il faut donc définir la clôture réflexive de la relation "*X est défini par T dans l'environnement*" et tester si cette relation est vérifiée entre le type étudié et *tobject*$(X)[LT]$. D'autres problèmes comme ceux liés à la définition des substitutions apparaissent également.

6.1.2 Règles

Les règles d'évaluation du langage se définissent assez aisément et sans surprise par le jugement $E \vdash t \hookrightarrow v$ qui signifie que, dans l'environnement d'évaluation E, le terme t s'évalue en v. On peut citer quelques unes de ces règles à titre d'exemple :

$$Red\_let \ : \ \frac{\{(x = a).E\} \vdash b \hookrightarrow v}{E \vdash let\ x : T = a\ in\ b \hookrightarrow v}$$

$$Red\_object \ : \ E \vdash object(x : T)[L] \hookrightarrow object(x : T)[L]$$

$$Red\_invoc \ : \ \frac{E \vdash a \hookrightarrow object(x : A)[L] \quad E \vdash b[^{a}/_{x}] \hookrightarrow v}{E \vdash invoc(a, l) \hookrightarrow v} \quad (l, b) \in L$$

$$Red\_field \ : \ \frac{E \vdash a \hookrightarrow object(x : A)[L]}{E \vdash upfield(a : T, l, b) \hookrightarrow object(x : A)[L[^{(l,b)}/_{(l,b_0)}]]} \quad (l, b_0) \in L$$

$$Red\_method \ : \ \frac{E \vdash a \hookrightarrow object(x : A)[L]}{E \vdash upmethod(a, l, (y : T)b) \hookrightarrow object(x : A)[L[^{b[^{x}/_{y}]}/_{b_0}]]} \quad (l, b_0) \in L$$

Le système de règles de typage est très proche de celui énoncé dans [1], les quelques modifications sont les mêmes que pour $Ob_{1<:\mu}$ et le jugement de typage reste $E \vdash t : T$, citons :

$$Val\_let \ : \ \frac{E \vdash a : T \quad \{(x : T).E\} \vdash \diamond \quad \{(x : T).E\} \vdash b : B}{E \vdash let\ x : T = a\ in\ b : B}$$

$$Val\_object \ : \ \frac{(\{(x : A).E\} \vdash \diamond \quad \{(x : A).E\} \vdash b_l : B_l[^{A}/_{X}]) \quad \forall (l, b_l) \in L}{E \vdash object(x : A)[L] : A} \quad A = tobject(X)[l : B_l]_{(l,b_l)\in L}$$

$$Val\_invoc \ : \ \frac{E \vdash a : tobject(X)[LT]}{E \vdash invoc(a, l) : B[^{tobject(X)[LT]}/_{X}]} \quad (l, B) \in LT$$

$$Val\_field \ : \ \frac{E \vdash a : tobject(X)[LT] \quad E \vdash b : B[^{tobject(X)[LT]}/_{X}]}{E \vdash upfield(a : T, l, b) : tobject(X)[LT]} \quad (l, B(X)) \in LT$$

$$Val\_method \ : \ \frac{E \vdash a : A \quad \{(x : A).E\} \vdash \diamond \quad \{(x : A).E\} \vdash b : B[^{A}/_{X}]}{E \vdash upmethod(a, l, (x : A)b) : A}(l, B(X)) \in LT \quad A = tobject(X)[LT]$$

$$SVal\_subsumption \ : \ \frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

On constate à nouveau ici que les constructions décrites à la section 3.2 pour simplifier l'écriture des règles s'appliquent très bien. Elles s'appliquent même à des règles plus complexes comme celles concernant les classes du langage *O-1* que nous n'utilisons pas ici (règle *Val_Subclass* en particulier).

6.2 Traduction de *O-1* vers $Ob_{1<:\mu}$

M. Abadi et L. Cardelli ont prouvé l'intéret du ζ-calcul en montrant qu'il est suffisamment puissant pour qu'il existe une traduction de *O-1* vers $Ob_{1<:\mu}$. On désire alors prouver la correction de cette traduction vis-à-vis des typages et des évaluations de *O-1* et $Ob_{1<:\mu}$.

6.2.1 Règles de traduction

Un système de règles définissant une traduction de *O-1* vers $Ob_{1<:\mu}$ est donné dans [1] avec le jugement $\vdash e \rightsquigarrow t$ qui signifie que le terme e de *O-1* est traduit par le terme t de $Ob_{1<:\mu}$. Il nous faut y ajouter les traductions de *let*, *add* et *int*.

$$Trad\_x \ : \ \vdash x \rightsquigarrow x$$

$$Trad\_int \ : \ \vdash int\, n \rightsquigarrow sint\, n$$

$$Trad\_add \ : \ \frac{\vdash e_1 \rightsquigarrow e_1 t \quad \vdash e_2 \rightsquigarrow e_2 t}{\vdash add(e_1, e_2) \rightsquigarrow sadd(e_1 t, e_2 t}$$

$$Trad\_let \ : \ \frac{\vdash e_2[^{e_1}/_x] \rightsquigarrow e_2 t}{\vdash let\, x = e_1 \ in\, e_2 \rightsquigarrow e_2 t}$$

$$Trad\_object \ : \ \frac{\vdash A \rightsquigarrow \mu(X)B(X) \quad (\vdash b_l \rightsquigarrow b_l t) \quad \forall (l = b_l) \in L}{\vdash object(x : A)[L] \rightsquigarrow sfold(\mu(X)B(X), [l = \zeta(x : B[^{\mu(X)B(X)}/_X])b_l t[^{sfold(\mu(X)B(X),x)}/_x]]_{(l=b_l)\in L})}$$

$$Trad\_invoc \ : \ \frac{\vdash a \rightsquigarrow at}{\vdash invoc(a, l) \rightsquigarrow sinvoc(sunfold(at), l)}$$

$$Trad\_Field\_update \ : \ \frac{\vdash a \rightsquigarrow at \quad \vdash b \rightsquigarrow bt \quad \vdash A \rightsquigarrow \mu(X)B(X) \quad x \notin bt}{\vdash upfield(a : A, l, b) \rightsquigarrow sfold(\mu(X)B(X), supdate(sunfold(at), l, \zeta(x : B[^{\mu(X)B(X)}/_X])bt))}$$

$$Trad\_Method\_update \ : \ \frac{\vdash a \rightsquigarrow at \quad \vdash b \rightsquigarrow bt \quad \vdash A \rightsquigarrow \mu(X)B(X)}{\vdash upmethod(a, l, (x : A)b) \rightsquigarrow sfold(\mu(X)B(X), supdate(sunfold(at), l, \zeta(x : B[^{\mu(X)B(X)}/_X])bt[^{sfold(\mu(X)B(X),x)}/_x]))}$$

Au travers de ces règles un peu denses apparaît ici l'importance des constructeurs *sfold* et *sunfold* décrits à la section 3.1.1. Mais on voit également qu'ils participent pour beaucoup à l'illisibilité du ζ-calcul.

La traduction des types de *O-1* en types de $Ob_{1<:\mu}$ ainsi que celle des environnements ne posent aucun problème particulier.

6.2.2 Correction de la traduction

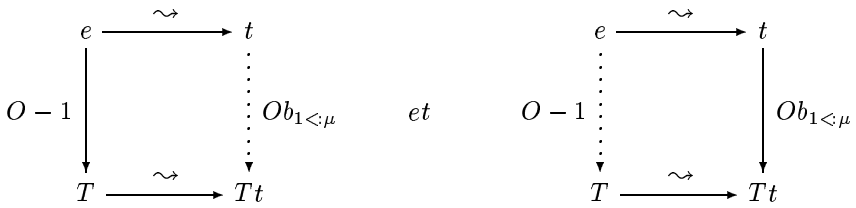
Pour prouver que la traduction définie ci-dessus est correcte par rapport aux systèmes de typage et d'évaluation décrits pour *O -1* et $Ob_{1<:\mu}$, il nous faut (d'après [12, 13]) démontrer les énoncés suivants :

Théorème 6.1 *Validité et complétude de la traduction du typage :*

$$\frac{\vdash E \rightsquigarrow Et \quad E \vdash_{O-1} e : T \quad \vdash e \rightsquigarrow t \quad \vdash T \rightsquigarrow Tt}{Et \vdash_{Ob_{1<:\mu}} t : Tt}$$

$$\frac{\vdash E \rightsquigarrow Et \quad \vdash e \rightsquigarrow t \quad Et \vdash_{Ob_{1<:\mu}} t : Tt \quad \vdash T \rightsquigarrow Tt}{E \vdash_{O-1} e : T}$$

Ce qui peut également se représenter par des diagrammes :

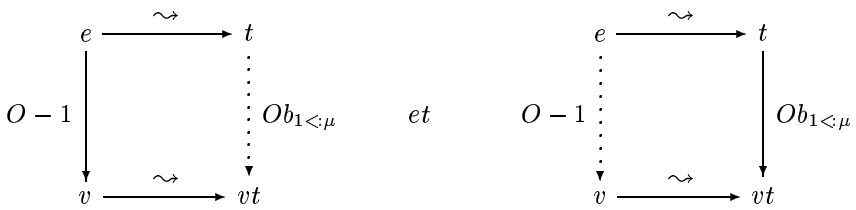


Théorème 6.2 *Validité et complétude de la traduction de l'exécution :*

$$\frac{\vdash E \rightsquigarrow Et \quad E \vdash_{O-1} e \hookrightarrow v \quad \vdash e \rightsquigarrow t \quad \vdash v \rightsquigarrow vt}{Et \vdash_{Ob_{1<:\mu}} t \hookrightarrow vt}$$

$$\frac{\vdash E \rightsquigarrow Et \quad \vdash e \rightsquigarrow t \quad Et \vdash_{Ob_{1<:\mu}} t \hookrightarrow vt \quad \vdash v \rightsquigarrow vt}{E \vdash_{O-1} e \hookrightarrow v}$$

Ce qui peut également se représenter par des diagrammes :



6.2.3 Preuve en Coq

Les premiers pas de la réalisation d'une telle preuve en Coq font ressortir tout particulièrement les problèmes liés à l'utilisation des environnements et des listes en général. En effet, la traduction des environnements est définie par plusieurs parcours de listes et la présence de ce jugement dans tous les énoncés de correction de traduction impose de prouver de nombreuses propriétés de ces parcours de listes ce qui serait rendu nettement plus facile si les environnements étaient implémentés sous une forme plus efficace et associée à des lemmes prédémontrés en Coq.

6.3 Théorème de conservation des types

La correction de la traduction de *O-1* vers $Ob_{1<:\mu}$ nous permet désormais de prouver le théorème de conservation des types pour *O-1* à partir de celui que nous avons démontré pour $Ob_{1<:\mu}$. L'énoncé du théorème est quasiment identique à celui du théorème 5.1 :

Théorème 6.3 *Soient E un environnement d'exécution, e un programme du langage O-1, v une valeur et T un type alors si $E \vdash e \hookrightarrow v$ et si $E \vdash e : T$ on a $E \vdash v : T$.*

Ce qui peut s'écrire sous la forme d'une règle d'inférence :

$$\frac{E \vdash e \hookrightarrow v \quad E \vdash e : T}{E \vdash v : T}$$

Démonstration : Cette preuve est particulièrement appropriée à une présentation sous forme d'arbre de preuve. On note t la traduction de e dans $Ob_{1<:\mu}$, vt celle de v, Tt celle de T et Et celle de E.

$$(i)\ \frac{\vdash E \rightsquigarrow Et \quad E \vdash_{O-1} e \hookrightarrow v \quad \vdash e \rightsquigarrow t \quad \vdash v \rightsquigarrow vt}{Et \vdash_{Ob_{1<:\mu}} t \hookrightarrow vt}\ \textit{Thm. 6.2}$$

$$(ii)\ \frac{\vdash E \rightsquigarrow Et \quad E \vdash_{O-1} e : T \quad \vdash e \rightsquigarrow t \quad \vdash T \rightsquigarrow Tt}{Et \vdash_{Ob_{1<:\mu}} t : Tt}\ \textit{Thm. 6.1}$$

$$\frac{\dfrac{}{\vdash E \rightsquigarrow Et} \quad \dfrac{}{\vdash v \rightsquigarrow vt} \quad \dfrac{\overbrace{Et \vdash_{Ob_{1<:\mu}} t \hookrightarrow vt}^{(i)} \quad \overbrace{Et \vdash_{Ob_{1<:\mu}} t : Tt}^{(ii)}}{Et \vdash_{Ob_{1<:\mu}} vt : Tt}\ \textit{Thm. 5.1} \quad \dfrac{}{\vdash T \rightsquigarrow Tt}}{E \vdash_{0-1} v : T}\ \textit{Thm. 6.1}$$

La possibilité d'écrire une preuve sous cette forme se traduit par une preuve en Coq très proche et donc très simple contrairement à tous les autres cas que nous avons étudiés jusque là et pour lesquels la preuve Coq subit une véritable explosion par rapport à la preuve sur papier. Ainsi la réalisation de la preuve du théorème 5.1 en Coq a nécessité l'application de plus de 2000 tactiques.

7 Conclusion

L'étude de la Sémantique Naturelle des langages à objets, et plus généralement de tout langage de programmation, grâce au système Centaur et, par la traduction automatique, au système Coq nécessite un enrichissement de ces différents outils. Les observations faites sur un mini-langage prouvent qu'il serait déraisonnable d'envisager l'étude d'un véritable langage sans avoir au préalable défini de nouvelles structures en Typol à prendre en compte dans la traduction vers Coq (dont la puissance d'expression reste bien supérieure à celle de Typol). En particulier l'utilisation systématique des listes en Typol pour coder les objets et les environnements entraîne en Coq une surcharge de définitions et de raisonnements inductifs qui pourraient être évités. De plus, les preuves en Sémantique Naturelle font apparaître des formes de raisonnements (par double induction et par inductions mutuelles) que l'on rencontre beaucoup moins ailleurs. Certaines faiblesses de Coq deviennent vraiment gênantes : leur correction devrait permettre de

faciliter ce type de travail en supprimant certaines manipulations techniques actuellement inévitables dans les preuves.

Cependant l'ampleur des améliorations à apporter reste très raisonnable et celles-ci devraient permettre une simplification importante des preuves en Coq. L'expérience décrite ici apparaît très prometteuse, aussi bien en ce qui concerne le pouvoir d'expression de la Sémantique Naturelle, que celui de Coq, en particulier dans le domaine des langages à objets et permet d'envisager un environnement de travail beaucoup plus efficace et s'appliquant à l'étude d'autres langages et d'autres propriétés. Ainsi grâce au nouveau système obtenu l'utilisation des travaux réalisés en Typol pour la spécification et l'exécution de la Sémantique Naturelle de nombreux langages réels (y compris dans le domaine des langages à objets [4, 3]) dans le but de prouver certaines propriétés de ces sémantiques apparaît tout à fait raisonnable.

Remerciements

Je tiens à remercier tout particulièrement Joëlle Despeyroux pour tout son encadrement et sa patience à relire cet article et Yves Bertot pour son point de vue sur l'usage de la *subsumption* mais également les autres membres du projet CROAP de l'INRIA Sophia-Antipolis pour leur accueil lors de mon stage de maîtrise qui a servi de point de départ à cet article *(le rapport de ce stage contenant des annexes plus complètes est disponible à l'adresse : http://www.eleves.ens.fr:8080/home/laurent/inria/rapport.ps)*.

Par ailleurs, je voudrais remercier Roberto Di Cosmo pour m'avoir fait faire mes premiers pas en théorie des langages à objets et m'avoir donné l'envie d'en savoir plus.

Références

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118) and as DEC Systems Research Center Research Report number 62, August 1990.

[3] I. Attali, D. Caromel, and S. O. Ehmety. An operationnal semantics for the Eiffel// language. Research Report 2732, INRIA, November 1995.

[4] I. Attali, D. Caromel, and M. Oudshoorn. A formal definition of the dynamic semantics of the Eiffel language. In *Australian Computer Science Conferenc (ACSC)*, Brisbane, Australie, February 1993.

[5] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CtCoq Proof Environment*. INRIA Sophia Antipolis, February 1996.

[6] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Symposium on Theoretical Aspects Computer Science (STACS)*, Sendai (Japan), December 1993. Also appears as a Cambridge University Research Report (December 1993).

[7] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, February 1989. ACM Press.

[8] D. Clement, J. Despeyroux, Th. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. In *Actes de la conférence int. ACM 'Lisp and Functional Programming' (LFP), MIT, USA*, 1986. also available as Inria Research Report RR-529, May 1986.

[9] C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual, version 6.1. Technical report, Inria-ENS Lyon, December 1996.

[10] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proof. In M. Dezani and G. Plotkin, editors, *Proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902. Springer-Verlag LNCS, April 1995.

[11] P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leqslant}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[12] J. Despeyroux. Proof of translation in Natural Semantics. In IEEE, editor, *Proceedings of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986*, pages 193–205, 1986. also available as a Research Report RR-514, Inria-Sophia-Antipolis, France, April 1986.

[13] J. Despeyroux. Proof of translation in Natural Semantics. polished and extended version of the LICS'86 paper, largely distributed, although never published, November 1987.

[14] J. Despeyroux. Sémantique Naturelle: Spécifications et Preuves. Unpublished lecture notes, 72 pages, in french, May 1997.

[15] T. Despeyroux. TYPOL: A formalism to implement Natural Semantics. Research Report 94, INRIA, Rocquencourt, France, March 1988.

[16] T. Despeyroux. As, for abstract syntax manual - v 1.0. Technical Report 0197, INRIA, September 1996.

[17] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[18] I. Jacobs and L. Rideau. A Centaur tutorial. Technical Report 140, INRIA Sophia Antipolis, July 1992. Revised 22 July 1994 for Centaur 2.0.

[19] G. Kahn. Natural Semantics. In *Proceedings of the Symp. on Theorical Aspects of Computer Science, Passau, Germany*, 1987. also available as a Research Report RR-601, Inria, Sophia-Antipolis, February 1987.

[20] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3(2):151–188, 1983.

[21] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[22] D. Terrasse. Encoding Natural Semantics in Coq. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, volume 936 of *Springer-Verlag LNCS*, pages 230–244, July 1995.

[23] G. Winskel. *The Formal Semantics of Programming Languages, An Introduction.* MIT Press, Cambridge, Massachusetts, 1993.

Nous donnons ici les données Coq relatives à cet article. Tout d'abord la liste des énoncés des lemmes qui nous ont permis de réaliser la preuve de conservation des types pour $Ob_{1<:\mu}$ en commençant par les axiomes que nous n'avons pas démontrés pour des raisons de complexité déraisonnable dans les systèmes actuels. La seconde annexe contient la preuve en Coq, réalisée par Y. Bertot, de l'équivalence entre deux raisonnements possibles pour les systèmes de règles d'inférence avec *Subsumption*.

A Lemmes Coq pour la preuve de SRT

```
Axiom enrich_senv:
(* l'ajout dans l'environnement d'une nouvelle definition compatible avec
   les autres ne modifie pas le typage *)
(* necessite de verifier que les variables de E n'apparaissent pas dans e,
   des permutations sur l'environnement et une preuve avec 3 predicats
   mutuellement recursifs *)
      (E:(l_list STYPING)) (t:STYPING) (e:TERM) (T:STYPE)
      (Typ0_scalc11 (senv E) e T) -> (SEnv (senv (l_cons STYPING t E))) ->
      (Typ0_scalc11 (senv (l_cons STYPING t E)) e T).

Axiom enrich_env_stype:
(* l'ajout dans l'environnement d'une nouvelle definition compatible avec
   les autres ne modifie pas la validite de type *)
      (E1:SENV) (T:STYPE) (SType E1 T) ->
      (E:(l_list STYPING)) E1 =(senv E) ->
      (t:STYPING)(SType (senv (l_cons STYPING t E)) T).

Axiom valid_typing:
(* tout type obtenu par typage est valide *)
      (E:SENV) (e:TERM) (T:STYPE) (Typ0_scalc11 E e T) ->
      (SEnv E) ->(SType E T).

Axiom subst_subtyp:
(* lemme de substitution pour la relation de sous-typage *)
      (E:SENV) (X':STVAR) (B':STYPE) (Bs':STYPE)
      (X:STVAR) (B:STYPE) (Bs:STYPE)
      (Stype_stype B' (mtype X' B') X' Bs') ->
      (Stype_stype B (mtype X B) X Bs) -> (SSub E (mtype X' B') (mtype X B)) ->
      (SSub E Bs' Bs).

Axiom subst_term:
(* lemme de substitution *)
      (E:(l_list STYPING)) (x:SVAR) (T:STYPE) (b:TERM)
      (B:STYPE) (Typ0_scalc11 (senv (l_cons STYPING (styping x T) E)) b B) ->
```

```
      (t:TERM) (Typ0_scalc11 (senv E) t T) ->
      (bs:TERM) (Term_term b t x bs) ->(Typ0_scalc11 (senv E) bs B).

Axiom subtyp_in_env:
(* lemme de substitution avec sous-typage dans l'environnement *)
      (E:(l_list STYPING)) (T1:STYPE) (T2:STYPE) (SSub (senv E) T1 T2) ->
      (x:SVAR) (b:TERM) (T:STYPE)
      (Typ0_scalc11 (senv (l_cons STYPING (styping x T2) E)) b T) ->
      (Typ0_scalc11 (senv (l_cons STYPING (styping x T1) E)) b T).

Lemma uniq_env_svar:
(* unicite du type associe a une variable dans un environnement valide *)
  (E:SENV) (SEnv E) ->
  (x:SVAR) (T:STYPE) (SEnv3 E x T) ->(T1:STYPE) (SEnv3 E x T1) ->T =T1.

Lemma uniq_in_object:
(* unicite du type associe a un label dans un objet valide *)
  (E:SENV) (LB_S:SIGTYP_S) (SType E (stobject LB_S)) ->
  (l:SLBL) (B0:STYPE) (SListOp (sigtyp l B0) LB_S) ->
  (B:STYPE) (SListOp (sigtyp l B) LB_S) ->B0 =B.

Lemma var1:
(* manipulation sur la traduction d'environnement d'execution en
   environnement de typage *)
  (E0:SVENV) (E:SENV) (STypEnv E0 E) ->
  (x:SVAR) (t:TERM) (SVEnv3 E0 x t) ->(Ex [T:STYPE] (SEnv3 E x T)).

Lemma var2:
(* propriete du typage d'une variable *)
  (E:SENV) (x:SVAR) (T:STYPE) (Typ0_scalc11 E x T) ->
  (T':STYPE) (SEnv3 E x T') -> (SEnv E) -> (SType E T') ->(SSub E T' T).

Lemma goodtype_in_senv:
(* validite de tous les types presents dans un environnement *)
  (E:SENV) (SEnv E) ->(x:SVAR) (T:STYPE) (SEnv3 E x T) ->(SType E T).

Lemma cas_var:
(* preuve du cas d'une variable pour le SRT *)
  (E:SVENV) (E':SENV) (STypEnv E E') -> (x:SVAR)
  (T:STYPE) (t:TERM) (SVEnv3 E x t) -> (SEnv E') -> (Typ0_scalc11 E' x T) ->
  (Typ0_scalc11 E' t T).
```

```
Lemma invoc1:
(* propriete du typage d'un objet *)
  (E:SENV) (LB:SIGLBLDECL_S) (LB_S:STYPE) (Typ0_scalc11 E (sobject LB) LB_S) ->
  (Ex [A:SIGTYP_S] (SForall2 E (stobject A) LB A) /\ (SListOp3 LB A)).

Lemma determ_forall:
(* determinisme du typage dans un SForall2 (typage des objets) *)
  (E:SENV) (T1:SIGTYP_S)
  (LB:SIGLBLDECL_S) (SForall2 E (stobject T1) LB T1) -> (SListOp3 LB T1) ->
  (T2:SIGTYP_S) (SForall2 E (stobject T2) LB T2) -> (SListOp3 LB T2) ->T1 =T2.

Lemma validtyp_object:
(* validite des types presents dans un type stobject *)
  (E:SENV) (LB_S:SIGTYP_S) (LB:SIGLBLDECL_S)
  (SForall2 E (stobject LB_S) LB LB_S) -> (SListOp3 LB LB_S) ->
  (SType E (stobject LB_S)).

Lemma invoc2:
(* autre propriete du typage des objets *)
  (E:SENV) (LB:SIGLBLDECL_S) (LB_S:STYPE) (Typ0_scalc11 E (sobject LB) LB_S) ->
  (A:SIGTYP_S) (SForall2 E (stobject A) LB A) -> (SListOp3 LB A) ->
  (SSub E (stobject A) LB_S).

Lemma invoc3_1:
(* autre propriete du typage des objets *)
  (E:SENV) (LB:SIGLBLDECL) (LB_S:SIGTYP_S) (T0:STYPE) (SForall1 E T0 LB LB_S) ->
  (l:SLBL) (b:TERM) (x:SVAR) LB =(siglbldecl l (sigdecl x T0 b)) ->
  (ee:(l_list STYPING)) E =(senv ee) ->
  (Ex [T:STYPE]
  (Typ0_scalc11 (senv (l_cons STYPING (styping x T0) ee)) b T) /\
  (SListOp (sigtyp l T) LB_S)).

Lemma invoc3_2:
(* propriete du type obtenu lors du typage d'un objet *)
  (E:SENV) (LB:SIGLBLDECL) (LB_S:SIGTYP_S) (T:STYPE) (SForall1 E T LB LB_S) ->
  (l:SLBL) (b:TERM) (x:SVAR) (T1:STYPE) LB =(siglbldecl l (sigdecl x T1 b)) ->
  T1 =T.

Lemma invoc3:
(* autre propriete du typage des objets *)
  (E:SENV)
  (LB:SIGLBLDECL_S) (LB_S:SIGTYP_S) (T0:STYPE) (SForall2 E T0 LB LB_S) ->
```

```
  (l:SLBL) (b:TERM) (x:SVAR) (SVListOp LB l b x) ->
  (ee:(l_list STYPING)) E =(senv ee) ->
  (Ex [T:STYPE]
  (Typ0_scalc11 (senv (l_cons STYPING (styping x T0) ee)) b T) /\
  (SListOp (sigtyp l T) LB_S)).

Lemma valid_left_sub:
(* validite des types a gauche des relations de sous-typage *)
(E:SENV) (A:STYPE) (B:STYPE) (SSub E A B) ->(SType E A).

Lemma sub_top:
(* Top est le seul sur-type de Top *)
(E:SENV) (B:STYPE) (SSub E stop B) ->B =stop.

Lemma sub_object:
(* les seuls sur-types d'un objets sont des objets et Top *)
  (E:SENV) (A:STYPE) (B:STYPE) (SSub E A B) ->
  (A1:SIGTYP_S) A =(stobject A1) ->
  (Ex [B1:SIGTYP_S] B =(stobject B1)) \/ B =stop.

Lemma invoc4_3:
(* Propriete de la relation de sous-typage pour les objets *)
  (E:SENV) (A:STYPE) (B:STYPE) (SSub E A B) ->
  (C:STYPE) (SSub E B C) -> (A1:SIGTYP_S) A =(stobject A1) ->
  (C1:SIGTYP_S) C =(stobject C1) ->(Ex [B1:SIGTYP_S] B =(stobject B1)).

Lemma invoc4_2_1:
(* Propriete du sous-typage entre objets *)
  (E:SENV) (A:SIGTYP_S) (B:SIGTYP_S) (SForall4 E A B) ->
  (l:SLBL) (B0:STYPE) (SListOp (sigtyp l B0) B) ->(SListOp (sigtyp l B0) A).

Lemma invoc4_2:
(* Propriete du sous-typage entre objets *)
  (E:SENV) (TA:STYPE) (TB:STYPE) (SSub E TA TB) ->
  (A:SIGTYP_S) (LB_S:SIGTYP_S) TA =(stobject A) -> TB =(stobject LB_S) ->
  (l:SLBL) (B0:STYPE) (SListOp (sigtyp l B0) LB_S) ->(SListOp (sigtyp l B0) A).

Lemma invoc4:
(* unicite du type associe a un label dans un objet valide avec sous-typage *)
  (E:SENV) (TA:STYPE) (TB:STYPE) (SSub E TA TB) ->
  (A:SIGTYP_S) (LB_S:SIGTYP_S)
  TA =(stobject A) -> TB =(stobject LB_S) -> (SType E (stobject LB_S)) ->
```

```
  (l:SLBL) (B0:STYPE) (SListOp (sigtyp l B0) A) ->
  (B:STYPE) (SListOp (sigtyp l B) LB_S) ->B0 =B.

Lemma cas_invoc:
(* cas sinvoc de la preuve *)
  (E:SENV) (LB:SIGLBLDECL_S) (LB_S:SIGTYP_S)
  (l:SLBL) (b:TERM) (bs:TERM) (x:SVAR) (B:STYPE)
  (Typ0_scalc11 E (sobject LB) (stobject LB_S)) ->
  (SVListOp LB l b x) ->
  (Term_term b (sobject LB) x bs) -> (SListOp (sigtyp l B) LB_S) -> (SEnv E) ->
  (Typ0_scalc11 E bs B).

Lemma lbldiff_object:
(* tous les labels d'un objet valide sont distincts *)
  (E:SENV) (T:STYPE) (SType E T) -> (s:SLBL) (s0:STYPE) (R:(l_list SIGTYP))
  T =(stobject (sigtyp_s (l_cons SIGTYP (sigtyp s s0) R))) ->
  (l:SLBL) (B:STYPE) (SListOp (sigtyp l B) (sigtyp_s R)) ->(DIFF_SLBL l s).

Lemma update1:
(* Propriete du typage des objets *)
  (E:(l_list STYPING)) (x:SVAR) (LB_S:SIGTYP_S)
  (SEnv1 (senv (l_cons STYPING (styping x (stobject LB_S)) E))) ->
  (l:SLBL) (B:STYPE) (SListOp (sigtyp l B) LB_S) -> (b:TERM)
  (Typ0_scalc11 (senv (l_cons STYPING (styping x (stobject LB_S)) E)) b B) ->
  (SForall1 (senv E) (stobject LB_S)
     (siglbldecl l (sigdecl x (stobject LB_S) b)) LB_S).

Lemma update2:
(* conservation du cardinal lors de reecritures *)
  (L1:SIGLBLDECL_S) (L:SIGTYP_S) (SListOp3 L1 L) ->
  (l:SLBL) (b:SIGDECL) (L2:SIGLBLDECL_S) (SVListOp1 L1 l b L2) ->
  (SListOp3 L2 L).

Lemma cas_update:
(* cas supdate de la preuve *)
  (E:(l_list STYPING)) (LB:SIGLBLDECL_S) (LB_S:SIGTYP_S)
  (l:SLBL) (b:TERM) (x:SVAR) (B:STYPE) (LBs:SIGLBLDECL_S)
  (Typ0_scalc11 (senv E) (sobject LB) (stobject LB_S)) ->
  (SListOp (sigtyp l B) LB_S) ->
  (SEnv1 (senv (l_cons STYPING (styping x (stobject LB_S)) E))) ->
  (Typ0_scalc11 (senv (l_cons STYPING (styping x (stobject LB_S)) E)) b B) ->
  (SVListOp1 LB l (sigdecl x (stobject LB_S) b) LBs) ->
```

```
  (Typ0_scalc11 (senv E) (sobject LBs) (stobject LB_S)).

Lemma unfold1:
(* propriete du typage de sfold *)
  (E:SENV) (T:STYPE) (b:TERM) (B:STYPE) (Typ0_scalc11 E (sfold T b) B) ->
  (Typ0_scalc11 E (sfold T b) T).

Lemma unfold2:
(* autre propriete du typage de sfold *)
  (E:SENV) (T:STYPE) (b:TERM) (B:STYPE) (Typ0_scalc11 E (sfold T b) B) ->
  (SSub E T B).

Lemma unfold3:
(* autre propriete du typage de sfold *)
  (E:SENV) (T:STYPE) (b:TERM) (B:STYPE) (Typ0_scalc11 E (sfold T b) B) ->
  (Ex [X':STVAR] (Ex [B':STYPE] T =(mtype X' B'))).

Lemma unfold4:
(* propriete du typage de sfold liee au sous-typage et a la substitution *)
  (E:SENV) (B':STYPE) (b:TERM) (Typ0_scalc11 E (sfold B' b) B') ->
  (X:STVAR) (B:STYPE) (SSub E B' (mtype X B)) ->
  (Bs:STYPE) (Stype_stype B (mtype X B) X Bs) ->(Typ0_scalc11 E b Bs).

Lemma cas_unfold:
(* cas sunfold de la preuve *)
  (E:SENV) (T:STYPE) (b:TERM) (X:STVAR) (B:STYPE)
  (Typ0_scalc11 E (sfold T b) (mtype X B)) ->
  (Bs:STYPE) (Stype_stype B (mtype X B) X Bs) ->(Typ0_scalc11 E b Bs).

Theorem subject_reduction:
(* ennonce usuel de SRT *)
  (E:SVENV) (e:TERM) (v:SVALUE) (Eval_scalc1 E e v) ->
  (E':SENV) (STypEnv E E') -> (SEnv E') ->
  (T:STYPE) (Typ0_scalc11 E' e T) ->(Typ0_scalc11 E' v T).

Theorem srt:
(* enonce plus synthetique de SRT *)
  (E:SVENV) (e:TERM) (v:SVALUE) (Eval_scalc1 E e v) ->
  (T:STYPE) (Typ0_scalc1 E e T) ->(Typ0_scalc1 E v T).
```

B Etude de la subsumption *(par Y. Bertot)*

```
Parameter type:Set.

Parameter order:type -> type ->Prop.

Parameter order_reflexive:(t:type)(order t t).

Parameter order_transitive:
          (t1, t2, t3:type) (order t1 t2) -> (order t2 t3) ->(order t1 t3).

(* definition d'un langage avec un type atomique et un constructeur de type
   unaire *)
Inductive lang: Set :=
   atom: nat ->lang
  | f: lang ->lang.

Parameter type_atom:nat ->type.

Parameter g:type ->type.

(* definition du typage fait en Typol *)
Inductive type_lang: lang -> type ->Prop :=
   t_l_atom: (n:nat)(type_lang (atom n) (type_atom n))
  | t_l_f: (e:lang) (t1:type) (type_lang e t1) ->(type_lang (f e) (g t1))
  | t_l_sub:
      (e:lang) (t1, t2:type) (type_lang e t1) -> (order t1 t2) ->
      (type_lang e t2).

Parameter h:lang ->lang.

Parameter h_g_coherent:
          (e:lang) (t:type) (type_lang e t) ->(type_lang (h e) (g t)).

(* definition de l'evaluation du langage *)
Inductive eval_lang: lang -> lang ->Prop :=
   e_l_atom: (n:nat)(eval_lang (atom n) (atom n))
  | e_l_f: (e, v:lang) (eval_lang e v) ->(eval_lang (f e) (h v)).

(* preuve du theoreme SRT comme nous l'avons realisee *)
Theorem srt:
  (e, v:lang) (eval_lang e v) ->(t:type) (type_lang e t) ->(type_lang v t).
```

```
Intros e v H'; Elim H'.
Intros n t H'0; Try Exact H'0.
Intros e0 v0 H'0 H'1 t H'2; Try Exact H'2.
Generalize H'1.
Cut (Ex [e1:lang] e1 =(f e0)).
2:Exists (f e0); Auto.
Intro H'3; Elim H'3; Intros e1 E; Try Exact E; Clear H'3.
Rewrite <- E in H'2.
Generalize E; Clear H'1 H'0; Generalize e0.
Elim H'2.
Intros n e2 H'0; Discriminate H'0.
Intros e2 t1 H'0 H'1 e3 H'3 H'4; Try Assumption.
Apply h_g_coherent.
Apply H'4.
Injection H'3.
Intro H'5; Rewrite <- H'5.
Try Exact H'0.
Intros e2 t1 t2 H'0 H'1 H'3 e3 H'4 H'5; Try Assumption.
Apply t_l_sub with t1 := t1.
Apply H'1 with e0 := e3.
Try Exact H'4.
Try Exact H'5.
Try Exact H'3.
Qed.

(* autre representation du typage *)
Inductive type_lang': lang -> type ->Prop :=
   t_l_atom': (n:nat) (t:type) (order (type_atom n) t) ->(type_lang' (atom n) t)
  | t_l_f':
      (e:lang) (t1, t2:type) (type_lang' e t1) -> (order (g t1) t2) ->
      (type_lang' (f e) t2).

(* preuve de l'equivalence des deux systemes de typage *)

Theorem to_prime: (e:lang) (t:type) (type_lang e t) ->(type_lang' e t).
Intros e t H'; Elim H'.
Intro n; Try Assumption.
Apply t_l_atom'.
Apply order_reflexive.
Intros e0 t1 H'0 H'1; Try Assumption.
Apply t_l_f' with t1 := t1.
Try Exact H'1.
```

```
Apply order_reflexive.
Intros e0 t1 t2 H'0 H'1; Case H'1.
Intros n t0 H'2 H'3; Try Assumption.
Apply t_l_atom'.
Apply order_transitive with t2 := t0; Auto.
Intros e1 t3 t0 H'2 H'3 H'4; Try Assumption.
Apply t_l_f' with t1 := t3; Auto.
Apply order_transitive with t2 := t0; Auto.
Qed.

Theorem prime_to: (e:lang) (t:type) (type_lang' e t) ->(type_lang e t).
Intros e t H'; Elim H'.
Intros n t0 H'0; Try Assumption.
Apply t_l_sub with t1 := (type_atom n).
Apply t_l_atom.
Try Exact H'0.
Intros e0 t1 t2 H'0 H'1 H'2; Try Assumption.
Apply t_l_sub with t1 := (g t1).
Apply t_l_f.
Try Exact H'1.
Try Exact H'2.
Qed.

(* preuve plus naturelle du SRT obtenue grace au nouveau systeme de typage *)
Theorem srt':
  (e, v:lang) (eval_lang e v) ->(t:type) (type_lang e t) ->(type_lang v t).
Intros e v H'; Elim H'.
Auto.
Intros e0 v0 H'0 H'1 t H'2; Try Assumption.
Specialize to_prime with 1 := H'2.
Intro H'3; Inversion H'3.
Apply t_l_sub with t1 := (g t1).
Apply h_g_coherent.
Apply H'1.
Apply prime_to.
Try Exact H0.
Try Exact H1.
Qed.
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399