Coq à la conquête des moulins

Laurence Rideau & Bernard Paul Serpette

Inria Sophia-Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
First.Last@inria.fr
http://www.inria.fr/lemme,oasis/First.Last

Résumé

Nous présentons la certification formelle en Coq d'un algorithme utilisé dans les compilateurs: l'affectation parallèle de registres. Nous proposons deux spécifications inductives et une spécification fonctionnelle de l'algorithme ainsi que les preuves de correction de ces spécifications. Un code fonctionnel ML peut être extrait de la spécification fonctionnelle et intégré au code du compilateur.

1. Introduction

Ce travail se place dans le cadre du projet Concert[3], dont l'objectif est la certification formelle d'un compilateur avec l'assistant de preuve Coq[2, 4]. L'idée est de spécifier et de formaliser en Coq toutes les étapes de la compilation[1, 9]: cela comprend la spécification des langages source, cible et des différents langages intermédiaires, la spécification de chaque étape de traduction entre ces langages ainsi que les preuves de correction de ces étapes de traduction. Parmi les différentes tâches d'un compilateur, on trouve aussi un certain nombre d'optimisations correspondant en général à des transformations de programmes au sein du même langage intermédiaire.

L'étude présentée ici correspond à la certification d'un algorithme utilisé dans une phase de traduction dans un langage intermédiaire: l'affectation parallèle de registres [8]. L'étude de cet algorithme est intéressante de plusieurs points de vue: l'algorithme peut être décrit en utilisant une représentation abstraite des registres, indépendamment des langages intermédiaires, et peut donc être réutilisé dans le contexte d'un autre compilateur, son code est raisonnablement court, mais présente des difficultés intéressantes à surmonter pour la spécification et pour les preuves: structure de données non conventionnelle, récursion non structurelle.

Dans un premier temps nous présentons l'affectation parallèle (section 1.1), puis nous décrivons une structure de donnée adaptée: les moulins (section 2.1). Ensuite nous donnons deux spécifications inductives de l'algorithme: une spécification non déterministe (section 3) et une déterministe (section 4), ainsi que les preuves de leur compatibilité et de leur correction. Enfin nous en déduisons une spécification fonctionnelle (section 5) dont nous donnons les preuves de cohérences avec les spécifications précédentes et les preuves de correction et de terminaison.

Les spécifications Coq et les preuves associées sont disponibles à l'adresse suivante :
`http://www-sop.inria.fr/lemme/concert/pmov/`.

1.1. Présentation

L'affectation parallèle est utilisée de manière interne par les compilateurs pour implémenter le protocole d'appel des fonctions. Généralement lors des premières passes, le code intermédiaire généré

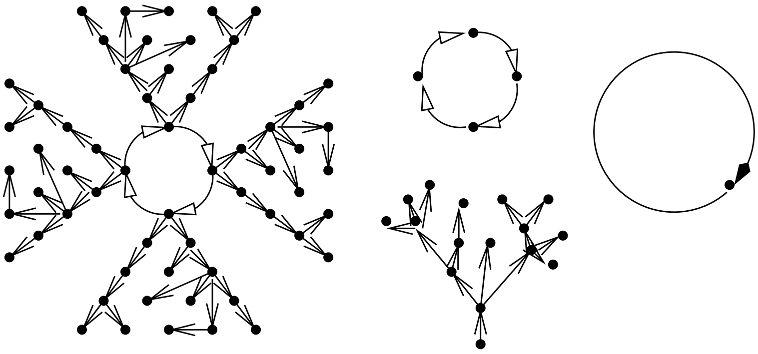

Figure 1: Exemples de moulins

par un compilateur utilise un nombre arbitraire de registres abstraits a_i. À ce stade, un point d'appel d'une fonction f à n arguments prend la forme : $a_r := f(a_1, \ldots, a_n)$. La passe d'allocation des registres va associer, via une fonction ϕ, un registre physique du processeur (élément de $\mathcal{R}$) à chaque registre abstrait. Cette passe peut nécessiter l'adjonction d'instructions de sauvegarde de registre (*spilling*). Après cette passe d'allocation des registres, le noeud d'application a la forme : $\phi(a_r) := f(\phi(a_1), \ldots, \phi(a_n))$. Une passe suivante implémente un protocole d'appel des fonctions en spécifiant, par exemple, que les k premiers arguments soient transmis dans les registres physiques $R_1, \ldots, R_k$. Supposons $k \geq n$, il faut insérer les instructions assurant que, au moment de l'appel de f, chaque registre $\phi(a_i)$ soit bien dans le registre R_i. De manière abstraite on utilise une affectation parallèle que l'on note $(R_1, \ldots, R_n) := (\phi(a_1), \ldots, \phi(a_n))$. Le but de l'algorithme que nous étudierons est de transformer (*sérialiser*) cette affectation parallèle en une suite d'affectation simple ($R_i := R_j$) ayant le même comportement. Cette sérialisation peut faire intervenir un registre temporaire, nommé tmp, par exemple pour $(R_1, R_2) := (R_2, R_1)$.

L'affectation parallèle peut être vue comme une relation de transfert sur l'ensemble des registres : $(\phi(a_1) \mapsto R_1) \bullet \ldots \bullet (\phi(a_n) \mapsto R_n)$. Ici la relation de transfert est spécifiée par la liste de ses arcs.

Nous noterons $\bullet$ l'opérateur de construction des listes, supposé associatif à droite. Pour la lisibilité nous surchagerons cet opérateur, si r est un registre et l une liste de registres, $r \bullet l$ est l'opérateur de construction de listes appelé $cons$, $l \bullet l$ est l'opérateur appelé $append$ et $x \bullet r$ est une contraction de $x \bullet r \bullet \emptyset$ où $\emptyset$ dénote la liste vide.

2. Description CAML de l'algorithme impératif

2.1. Présentation des moulins

La seule propriété que nous assure le protocole d'appel des fonctions est que les registres de destination sont uniques : les arguments différents d'une fonction ne doivent pas se retrouver dans le même registre physique. Ou, de manière équivalente, que les registres ont au plus un prédécesseur pour la relation de transfert. Bien que cette propriété ressemble à une spécification des forêts (ensemble disjoint d'arbres), elle n'interdit pas la présence de cycles : $(r_1 \mapsto r_2) \bullet (r_2 \mapsto r_1)$. Pour avoir un arbre il faut rajouter le fait qu'il existe un unique élément, la racine, n'ayant pas de prédécesseur.

Les relations vérifiant cette propriété d'*au plus un prédécesseur* sont des ensembles disjoints de moulins. Un moulin est un cycle (l'axe) d'où s'échappent des arbres (les pales). La figure 1 présente un ensemble de moulins où l'on retrouve le cas général, un arbre, un simple cycle sur quatre registres et le cas particulier d'une boucle.

2.2. Algorithme intuitif basé sur la topologie

Dans le cas particulier d'un cycle simple, i.e. un axe sans pale, la relation de transfert est de la forme : $(r_n \mapsto r_1) \bullet \ldots \bullet (r_2 \mapsto r_3) \bullet (r_1 \mapsto r_2)$ et la sérialisation se fait à l'aide d'un registre temporaire tmp : $tmp := r_1; r_1 := r_n; \ldots; r_3 := r_2; r_2 := tmp$.

C. May [8] généralise ce cas particulier en proposant un algorithme basé sur la topologie de la relation de transfert (i.e. nécessite une passe pour calculer les successeurs) : *enlever un à un tous les arcs n'ayant pas de successeur et garder cet ordre pour les affectations séquentielles* (i.e. les pales disparaissent petit à petit); *à ce stade il ne reste que des cycles simples pouvant être sérialisés selon l'algorithme introduisant un registre temporaire.*

L'algorithme que nous étudions, proposé par Xavier Leroy, effectue la sérialisation tout en découvrant la topologie (i.e. en une seule passe).

2.3. Version Caml

```
01 type status = To_move | Being_moved | Moved
02
03 let parallel_move src dst tmp =
04   let n = Array.length src in
05   let status = Array.make n To_move in
06   let rec move_one i =
07     if src.(i) <> dst.(i) then begin
08       status.(i) <- Being_moved;
09       for j = 0 to n - 1 do
10         if src.(j) = dst.(i) then
11           match status.(j) with
12             To_move ->
13               move_one j
14           | Being_moved ->
15               printf "%s := %s;\n" tmp src.(j);
16               src.(j) <- tmp
17           | Moved ->
18               ()
19       done;
20       printf "%s := %s;\n" dst.(i) src.(i);
21       status.(i) <- Moved
22     end in
23   for i = 0 to n - 1 do
24     if status.(i) = To_move then move_one i
25   done
```

L'algorithme utilise une variante d'un parcours de graphe en profondeur par la fonction `move_one` prenant en argument un arc $(src_i \mapsto dst_i)$ de la relation de transfert. Les lignes 9 et 10 correspondent à prendre tous les successeurs de dst_i. Le cas de la ligne 17 correspond à un arc déjà analysé. Le cas de la ligne 12 correspond à une récursion simple. Le cas de la ligne 14 correspond à la découverte d'un cycle. À la sortie de cette boucle, ligne 19, tous les nœuds atteignables par dst_i ont été analysés, l'arc $(src_i \mapsto dst_i)$ est sérialisé et est annoté comme analysé.

À la découverte d'un cycle, la pile d'appel est de la forme $((r_n \mapsto r_1), \ldots, (r_2 \mapsto r_3), (r_1 \mapsto r_2))$ et nous analysons à nouveau, à la ligne 14, l'arc $(r_1 \mapsto r_2)$. L'effet de bord de la ligne 16 opère donc nécessairement sur l'arc ayant entamé la récursion à la fonction `move_one`, i.e. l'arc se trouvant au fond de la pile.

La boucle principale, lignes 23 à 25, assure que tous les arcs sont analysés.

3. Algorithme non déterministe

On peut remarquer que tous les arcs de la relation de transfert débutent dans l'état `To_Move` (ligne 5), passent par l'état `Being_moved` (ligne 8), pour finir définitivement dans l'état `Moved` (ligne 21). Plutôt que d'avoir à gérer une liste de statuts, nous utiliserons trois listes disjointes d'arcs. Chacunes de ces listes contenant les arcs ayant le même statut.

L'algorithme va donc extraire un élément de la liste To_move (notée μ) pour l'ajouter dans une seconde $Being\_moved$ (notée σ) ou enlever un élément de cette dernière afin de le mettre dans la dernière liste $Moved$ (notée τ) contenant les résultats.

La liste $Being\_moved$ est utilisée comme une pile en simulant les appels récursifs à la fonction `move_one`, néanmoins on permettra la modification du dernier élément de cette pile lors de la découverte d'un cycle.

La dernière liste, $Moved$, sert à mémoriser les appels à la fonction `printf`, aussi les affectations séquentielles présentent dans cette liste se feront de la droite vers la gauche.

3.1. Règles inductives

Les règles inductives vont décrire les différentes réécritures sur le triplet de ces trois listes : (μ, σ, τ). Le triplet de départ est $(\mu, \emptyset, \emptyset)$ où μ est la relation de transfert.

$$\frac{}{(\mu_1 \bullet (r \mapsto r) \bullet \mu_2, \sigma, \tau) \rhd (\mu_1 \bullet \mu_2, \sigma, \tau)} \quad [Nop]$$

La première règle s'occupe du cas où la relation de transfert possède un arc $(r \mapsto r)$ dont la source et la destination sont identiques. Ceci correspond au test de la ligne 7. On remarquera que dans la version impérative cet arc n'est pas marqué avec `Moved` et gardera le statut de `To_move` jusqu'à la fin de l'algorithme. Néanmoins cet arc ne peut pas provenir d'un appel récursif à la fonction `move_one` car, dans ce cas, l'appelant correspondrait à un arc $(s \mapsto r)$ ce qui violerait le fait que la fonction de transfert soit un moulin: deux arcs différents ayant la même destination. Ainsi il est tout à fait valide de remonter le test de la ligne 7 au niveau de la boucle principale à la ligne 24.

Cette règle Nop, telle qu'elle est écrite, est un cas particulier de la règle suivante ($Start$), ce qui exprime le fait que ce test n'est pas nécessaire à la validité de l'algorithme et peut être considérée comme une optimisation.

$$\frac{}{(\mu_1 \bullet (s \mapsto d) \bullet \mu_2, \emptyset, \tau) \rhd (\mu_1 \bullet \mu_2, (s \mapsto d), \tau)} \quad [Start]$$

La règle $Start$ correspond au premier appel à la fonction `move_one`, i.e. à la ligne 24. On sait que dans ce cas aucun arc n'a le statut `Being_Moved` et cette règle sera donc la seule à considérer le cas où la liste $Being\_moved$ est vide.

$$\frac{}{(\mu_1 \bullet (d \mapsto r) \bullet \mu_2, (s \mapsto d) \bullet \sigma, \tau) \rhd (\mu_1 \bullet \mu_2, (d \mapsto r) \bullet (s \mapsto d) \bullet \sigma, \tau)} \quad [Push]$$

La règle $Push$ correspond à l'appel récursif de la ligne 13. Si l'arc en cours d'analyse est $(s \mapsto d)$ (i.e. le sommet de la liste $Being\_moved$) et s'il existe un successeur $(d \mapsto r)$ dans la liste To_move, elle transfère cet arc successeur au sommet de la pile $Being\_moved$.

$$\frac{}{(\mu, \sigma \bullet (s \mapsto d), \tau) \rhd (\mu, \sigma \bullet (tmp \mapsto d), (s \mapsto tmp) \bullet \tau)} \quad [Loop]$$

La règle *Loop* correspond, dans l'esprit, au cas particulier des lignes 15 et 16 étudiant la découverte d'un cycle. Cette règle est plus générale et considère qu'il n'est pas indispensable de vérifier la présence d'un cycle pour insérer un transfert par le registre temporaire. Ainsi ces lignes 15 et 16 peuvent être aussi remontées au niveau de la boucle principale (ligne 24) et le fait que l'on n'utilise le registre temporaire que lors de la présence d'un cycle doit être considéré comme une optimisation. Le point délicat sera d'imposer l'utilisation de ce registre lors d'un cycle, ceci sera résolu par la règle suivante.

$$\frac{NoRead(\mu, d_n) \ \wedge \ d_n \neq s_0}{(\mu, (s_n \mapsto d_n) \bullet \sigma \bullet (s_0 \mapsto d_0), \tau) \rhd (\mu, \sigma \bullet (s_0 \mapsto d_0), (s_n \mapsto d_n) \bullet \tau)} \quad [Pop]$$

La règle *Pop* correspond au retour d'un appel récursif à la fonction `move_one`. La première prémisse de cette règle, $NoRead(\mu, d_n)$, qui se définit formellement par $(\mu = \mu_1 \bullet (s \mapsto d) \bullet \mu_2 \ \Rightarrow \ s \neq d_n)$, vérifie que l'arc en cours d'étude $(s_n \mapsto d_n)$ n'a plus de successeur dans la liste To_move et empêche ainsi l'utilisation de la règle *Push*. La seconde prémisse, $d_n \neq s_0$ impose l'utilisation de la règle *Loop* dans le cas d'un cycle.

$$\frac{NoRead(\mu, d_n)}{(\mu, (s \mapsto d), \tau) \rhd (\mu, \emptyset, (s \mapsto d) \bullet \tau)} \quad [Last]$$

La règle *Last*, correspondant au retour à la boucle principale de la ligne 24, est un cas particulier de la règle *Pop*.

Pour revenir au cas particulier de la règle *Nop*, on observe qu'une liste To_move de la forme $((r \mapsto r))$ peut, soit s'éliminer par la règle *Nop*, soit se réécrire en elle-même en deux étapes (*Start* puis *Last*), soit utiliser le registre temporaire (*Start*, *Loop* puis *Last*) et se réécrire en $((tmp \mapsto r) \bullet (r \mapsto tmp))$. Ces trois résultats sont valides.

3.2. Preuves

Le théorème concluant cette sous-section exprimera que les règles inductives, introduites précédemment, permettent de transformer une relation de transfert en une liste séquentielle d'affectations ayant le même comportement.

3.2.1. Invariant

Définition 1 (Invariant) *Un triplet* $t = (\mu, \sigma, \tau)$ *est dit* **bien formé**, $Ok(\mu, \sigma, \tau)$, *si et seulement si :*

1. $\mu \bullet \sigma$ *représente un moulin :* $\mu \bullet \sigma = (l_1 \bullet (s_i \mapsto d_i) \bullet l_2 \bullet (s_j \mapsto d_j) \bullet l_3) \ \Rightarrow \ d_i \neq d_j$.
2. *La liste* μ *ne contient pas le registre temporaire :* $\mu = \mu_1 \bullet (s \mapsto d) \bullet \mu_2 \ \Rightarrow \ (s \neq tmp \ \wedge \ d \neq tmp)$.
3. *La liste* σ, *ne peut utiliser le registre temporaire qu'en position source du dernier arc :* $\sigma = \sigma_1 \bullet (s_0 \mapsto d_0) \ \Rightarrow \ d_0 \neq tmp$ *et* $\sigma_1 = \sigma_2 \bullet (s \mapsto d) \bullet \sigma_3 \ \Rightarrow \ (s \neq tmp \ \wedge \ d \neq tmp)$.
4. *La liste* σ *est un chemin :* $\sigma = ((r_{n-1} \mapsto r_n) \bullet \ldots \bullet (r_2 \mapsto r_3) \bullet (r_1 \mapsto r_2))$

On remarque que, si μ est un moulin n'utilisant pas le registre temporaire, alors le triplet $(\mu, \emptyset, \emptyset)$ est bien formé.

Lemme 1 (Conservation de l'invariant) *Les règles de réécriture transforment des triplets bien formés en d'autres triplets bien formés :* $(t_1 \rhd t_2) \ \wedge \ Ok(t_1) \ \Rightarrow \ Ok(t_2)$.

Preuve Par étude de cas sur $t_1 \rhd t_2$. Pour $t_1 = (\mu, \sigma, \tau)$, la seule règle rajoutant un élément à σ est *Push*, mais cet arc continue le chemin présent dans σ.

3.2.2. Sémantiques des affectations

Afin de pouvoir exprimer la validité de la relation de réécriture, nous supposerons l'existence d'un type $Valeur$ correspondant aux contenus des registres. La valeur d'un registre se calcule via un environnement ρ, fonction du domaine des registres vers les valeurs ($\mathcal{R} \rightarrow Valeur$).

Modifier un environnement ρ en affectant la valeur v au registre r s'écrit : $\rho[r := v]$.

L'exécution parallèle d'une relation de transfert dans un environnement ρ est définie par la fonction $Exec_{//}$:

$$Exec_{//}((s_n \mapsto d_n) \bullet \ldots \bullet (s_0 \mapsto d_0), \rho) = \rho[d_0 := \rho(s_0)] \ldots [d_n := \rho(s_n)]$$

Dans cette définition nous omettons les parenthèses car nous supposons que les relations de transfert sont toujours des moulins et utiliserons intensivement le lemme suivant :

Lemme 2 (Indépendance) *Si $\mu = \mu_1 \bullet (s \mapsto d) \bullet \mu_2$ est un moulin alors l'exécution parallèle est indépendante de l'ordre de μ : $Exec_{//}(\mu, \rho) = Exec_{//}((s \mapsto d) \bullet \mu_1 \bullet \mu_2, \rho)$.*

Preuve Par induction sur μ_1. Le cas de base est trivial, tandis que le pas d'induction va nécessiter de prouver qu'il est possible de permuter les deux premiers éléments de μ_1 : $(\rho[d_0 := \rho(s_0)])[d_1 := \rho(s_1)] = (\rho[d_1 := \rho(s_1)])[d_0 := \rho(s_0)]$. Cette égalité est vérifiée dès lors que $d_0 \neq d_1$ ce que nous assure la structure de moulin.

L'effet d'une séquence d'affectations sur un environnement ρ est :

$Exec_S(\emptyset, \rho) = \rho$

$Exec_S((s \mapsto d) \bullet \tau, \rho) = \rho'[d := \rho'(s)] \quad$ *avec* $\rho' = Exec_S(\tau, \rho)$

Définition 2 (Équivalence des environnements) *Deux environnements ρ_1 et ρ_2 sont dit équivalents, $\rho_1 \equiv \rho_2$ si et seulement si les valeurs des registres, autre que le temporaire, ont les mêmes valeurs dans ces deux environnements : $\rho_1 \equiv \rho_2 \Leftrightarrow \forall r \neq tmp,\ \rho_1(r) = \rho_2(r)$.*

Définition 3 (Exécution d'un triplet) *Exécuter un triplet (μ, σ, τ) correspond à exécuter τ d'une manière séquentielle puis $\mu \bullet \sigma$ de manière parallèle : $Exec((\mu, \sigma, \tau), \rho) = Exec_{//}(\mu \bullet \sigma, Exec_S(\tau, \rho))$.*

On remarque que $Exec((\mu, \emptyset, \emptyset), \rho) = Exec_{//}(\mu, \rho)$ et que $Exec((\emptyset, \emptyset, \tau), \rho) = Exec_S(\tau, \rho)$.

3.2.3. Correction

Lemme 3 (Conservation de l'exécution sur un pas) *Les règles de réécriture produisent des triplets ayant la même exécution : $(t_1 \triangleright t_2) \wedge Ok(t_1) \Rightarrow \forall \rho,\ Exec(t_1, \rho) \equiv Exec(t_2, \rho)$.*

Preuve Par étude de cas sur $t_1 \triangleright t_2$ où $t_1 = (\mu, \sigma, \tau)$.

- Nop utilise le fait que $\rho[r := \rho(r)] = \rho$.
- $Start$ et $Push$ utilisent le lemme d'indépendance de l'ordre.
- $Loop$ se fait en sachant que μ et σ n'utilisent pas le registre temporaire.
- Pop revient à prouver que

  $$Exec_{//}((s_n \mapsto d_n) \bullet \mu \bullet \sigma, Exec_S(\tau, \rho)) = Exec_{//}(\mu \bullet \sigma, Exec_S((s_n \mapsto d_n) \bullet \tau, \rho)).$$

  Ceci n'est valide que si le registre d_n n'est pas la source d'un arc de $\mu \bullet \sigma$. Pour μ, ceci est vérifié par la prémisse; pour σ, et par induction, on utilise le fait que σ est un chemin : le cas de base est imposé par la prémisse ($d_n \neq s_0$) et le pas d'induction vient avec la structure de moulin de σ.

- *Last* est un cas particulier de *Pop*.

Ce lemme s'étend facilement à la fermeture reflexive et transitive $\triangleright^*$ de $\triangleright$:

Lemme 4 (Conservation de l'exécution) *Plusieurs pas de règles de réécriture produisent des triplets ayant la même exécution :* $(t_1 \triangleright^* t_2) \ \wedge \ Ok(t_1) \ \Rightarrow \ \forall \rho, \ Exec(t_1, \rho) \equiv Exec(t_2, \rho)$

Preuve Par induction sur le nombre de pas.

Ainsi le théorème de correction devient trivial.

Théorème 1 (Correction de $\triangleright^*$) *Si μ est un moulin n'utilisant pas le registre temporaire, et si le triplet $(\mu, \emptyset, \emptyset)$ peut se réécrire en un autre triplet $(\emptyset, \emptyset, \tau)$ alors l'exécution parallèle de μ est équivalente à l'exécution séquentielle de τ :* $Exec_{//}(\mu, \rho) \equiv Exec_S(\tau, \rho)$.

4. Algorithme inductif déterministe

L'étape suivante consiste à *déterminiser* les règles inductives en imposant un ordre dans le choix des règles et en spécifiant l'arc à extraire de la liste To_move pour les règles Nop, $Push$ et $Start$.

$$\frac{}{((r \mapsto r) \bullet \mu, \emptyset, \tau) \hookrightarrow (\mu, \emptyset, \tau)} \quad [Nop]$$

$$\frac{s \neq d}{((s \mapsto d) \bullet \mu, \emptyset, \tau) \hookrightarrow (\mu, (s \mapsto d), \tau)} \quad [Start]$$

Les deux premières règles s'occupent du cas où la liste $Being\_moved$ est vide et ne s'intèressent qu'à l'élément le plus à gauche de la liste μ.

$$\frac{NoRead(\mu_1, d)}{(\mu_1 \bullet (d \mapsto r) \bullet \mu_2, (s \mapsto d) \bullet \sigma, \tau) \hookrightarrow (\mu_1 \bullet \mu_2, (d \mapsto r) \bullet (s \mapsto d) \bullet \sigma, \tau)} \quad [Push]$$

La règle $Push$ s'occupe du cas où l'arc en cours d'analyse a un sucesseur dans la liste μ. Elle impose de prendre le successeur le plus à gauche.

$$\frac{NoRead(\mu, r_0)}{(\mu, (s \mapsto r_0) \bullet \sigma \bullet (r_0 \mapsto d), \tau) \hookrightarrow (\mu, \sigma \bullet (tmp \mapsto d), (s \mapsto r_0) \bullet (r_0 \mapsto tmp) \bullet \tau)} \quad [LoopPop]$$

$$\frac{NoRead(\mu, d_n) \ \wedge \ d_n \neq s_0}{(\mu, (s_n \mapsto d_n) \bullet \sigma \bullet (s_0 \mapsto d_0), \tau) \hookrightarrow (\mu, \sigma \bullet (s_0 \mapsto d_0), (s_n \mapsto d_n) \bullet \tau)} \quad [Pop]$$

Ces deux règles s'occupent du cas où l'arc en cours d'analyse n'a plus de successeur dans la liste μ et où la liste $Being\_moved$ contient au moins deux éléments. La règle $LoopPop$, pour refléter la dualité avec la règle Pop, impose la présence d'un cycle. On peut remarquer que cette règle, contrairement aux règles non déterministes, en profite pour dépiler l'arc en cours d'analyse. Cette particularité assure que toutes les règles effectuent le mouvement d'un arc, ce trait sera utilisé pour la preuve de terminaison.

$$\frac{NoRead(\mu, d)}{(\mu, (s \mapsto d), \tau) \hookrightarrow (\mu, \emptyset, (s \mapsto d) \bullet \tau)} \quad [Last]$$

La règle *Last* s'occupe du dernier cas où la liste *Being_moved* ne contient plus qu'un seul élément.

Comme nous n'avons fait que des restrictions sur les règles non deterministes, les deux lemmes suivants sont triviaux.

Lemme 5 (Inclusion $\hookrightarrow \subseteq \triangleright^*$) *La relation $\hookrightarrow$ est incluse dans $\triangleright^*$: $t_1 \hookrightarrow t_2 \Rightarrow t_1 \triangleright^* t_2$.*

Preuve Par étude de cas, seule la règle *LoopPop* fait intervenir la transitivité de $\triangleright^*$.

Lemme 6 (Inclusion $\hookrightarrow^* \subseteq \triangleright^*$) *La relation $\hookrightarrow^*$ est incluse dans $\triangleright^*$: $t_1 \hookrightarrow^* t_2 \Rightarrow t_1 \triangleright^* t_2$.*

Preuve Triviale.

Ce qu'il est important de comprendre c'est que ces règles déterministes ne correspondent plus à l'algorithme impératif. En effet, dans l'algorithme impératif, lorsque le registre temporaire est utilisé (découverte d'une boucle) l'arc en cours d'analyse peut avoir encore un successeur, ce qui n'est pas le cas dans la règle *Loop*.

Enfin, même si la preuve du théorème de correction de $\hookrightarrow^*$ est triviale, il reste à prouver le déterminisme et l'existence d'une forme normale (i.e. le triplet $(\emptyset, \emptyset, \tau)$). Ceci sera fait en donnant une version fonctionnelle de ces règles inductives.

5. Version fonctionnelle

À partir des règles inductives déterministes, il est facile de construire la fonction $stepf$, correspondant à une étape de calcul de l'algorithme. La fonction générale *Pmov* a *naturellement* la forme suivante:

$$\forall S, Pmov\ S = match\ S\ with\ ((nil, nil), \_) \Rightarrow S \mid \_ \Rightarrow Pmov\ (stepf\ S)\ end.$$

La fonction ainsi définie n'étant pas structurellement récursive (l'appel récursif se fait sur $stepf(S)$ et non sur un sous-terme de S), une telle définition n'est pas acceptée directement par Coq. Pour définir notre fonction, nous avons utilisé les outils de construction de fonctions récursives générales proposés par Balaa et Bertot [5]. Pour cela nous devons d'abord définir la fonction $stepf$ correspondant à une itération du calcul, puis une relation bien fondée qui sera utilisée pour justifier la terminaison et enfin le théorème montrant que l'appel récursif se fait bien sur un prédécesseur (par la relation bien fondée) de l'argument de la fonction.

5.1. La fonction d'exécution d'un pas de l'algorithme

La fonction `stepf` que nous avons définie s'appuie directement sur les règles inductives déterministes décrites précédemment (associées à la relation $\hookrightarrow$), elle prend en argument un état S (un triplet (t, b, l)) et retourne un nouveau triplet. Le calcul se fait par cas:

Definition *stepf* (S : *State*) : *State* :=
  match S with
    $(\emptyset,\ \emptyset,\ \_) \Rightarrow S$ (* cas d'arrêt *)
    $\mid ((s,\ d) :: tl,\ \emptyset,\ l) \Rightarrow$
      match eq_nat_dec s d with

```
      left _ ⇒ (tl, ∅, l)                                              (* s=d, [Nop] *)
    | right _ ⇒ (tl, (s, d) :: ∅, l)                                   (* s≠d, [Start] *)
    end
| (t, (s, d) :: b, l) ⇒
    match split_move t d with
      Some ((t1, r, t2)) ⇒
        (t1 ++ t2, (d, r) :: ((s, d) :: b), l)                         (* t = t1 • (d, r) • t2, [Push] *)
    | None ⇒
        match b with
          ∅ ⇒ (t, ∅, (s, d) :: l)                                      (* [Last] *)
        | _ ⇒
            match eq_nat_dec d (fst (last b)) with
              left _ ⇒
                (t, replace_last_s b, (s, d) :: ((d, tmp) :: l))       (*b = _ • (d, _), [LoopPop]*)
            | right _ ⇒ (t, b, (s, d) :: l)                            (* [Pop] *)
            end end end end.
```

Le premier cas ne correspond à aucune des règles de la spécification inductive: c'est le cas d'arrêt de la récursion associé au triplet $(\emptyset, \emptyset, \tau)$.
La fonction *eq_nat_dec*, fonction standard de la librairie `Arith` de Coq, permet de choisir entre le cas où les registres donnés en argument sont égaux (cas *left*) ou différents (cas *right*).
La fonction *split_move* prend en argument une liste d'arcs t et un registre d. Si le registre est une source d'un arc de la liste t, la fonction retourne l'arc correspondant $(d, \_)$, et les deux sous-listes entourant l'arc trouvé. Ceci permet de casser la liste t en $t_1 \bullet (d, r) \bullet t_2$ correspond à l'application de la règle `[Push]`.
Enfin, la fonction *replace_last_s* prend en argument une liste b d'arcs et remplace dans b, la source du dernier arc de b par le registre tmp: $hb \bullet (d, r)$ devient $hb \bullet (tmp, r)$ (règle `[LoopPop]`).
La définition de la fonction *stepf* étant très proche de la spécification inductive déterministe de l'algorithme, il est facile de montrer le théorème:

Théorème 2 (Compatibilité de $stepf$ **et** $\hookrightarrow$**)** *Pour tout triplet* $S = (t, b, l)$ *non trivial (i.e.* $S \neq (\emptyset, \emptyset, \tau)$*),* S *est en relation avec* $stepf(S)$ *par* $\hookrightarrow$*:* $\forall \tau, S \neq (\emptyset, \emptyset, \tau) \Rightarrow S \hookrightarrow stepf\ S$

Preuve Par étude de cas sur la composition de S et application de la bonne règle de la relation $\hookrightarrow$ pour chaque cas.

5.2. La fonction récursive générale

La fonction récursive générale se construit par itération de la fonction $stepf$. Pour assurer la terminaison nous devons exhiber une relation $\prec$ bien fondée telle que $stepf(S) \prec S$, ce qui revient à trouver une fonction de mesure sur les triplets qui soit décroissante à chaque appel de $stepf$ (à l'exception du cas d'arrêt $(\emptyset, \emptyset, \tau)$). En étudiant les règles de l'algorithme déterministe, et en regardant plus précisément les tailles des listes To_move et $Being\_move$, on s'aperçoit qu'il y a:

- soit suppression d'un arc de la liste To_move, la liste $Being\_move$ restant inchangée: règle `[Nop]`
- soit transfert d'un arc de la liste To_move vers la liste $Being\_move$, règles `[Start]` et `[Push]`
- soit la liste To_move reste inchangée alors que la liste $Being\_move$ perd un arc par transfert vers la liste $Moved$, règles `[LoopPop]`, `[Pop]` et `[Last]`.

On voit donc que soit l'une des listes To_move ou $Being\_move$ perd un élément, l'autre liste restant inchangée, soit il y a transfert de To_move vers $Being\_move$. On assure la décroissance en accordant un poids plus fort aux arcs de la liste To_move comme dans la fonction $mesure$ suivante:

Définition 4 (La mesure d'un triplet) $mesure(t, b, m) = 2 * length(t) + length(b)$.

On montre ensuite facilement le lemme:

Lemme 7 (Décroissance de la mesure pour $\hookrightarrow$) $S_1 \hookrightarrow S_2 \Rightarrow mesure(S2) < mesure(S1)$.

Preuve Par calcul de la mesure dans tous les cas de la relation $\hookrightarrow$.

En utilisant le théorème **2** de compatibilité de $stepf$ et $\hookrightarrow$, on obtient trivialement le lemme:

Lemme 8 (Décroissance de la mesure pour $stepf$) $S \neq (\emptyset, \emptyset, \tau) \Rightarrow mesure(stepf(S)) < mesure(S)$.

Pour obtenir une relation bien fondée à partir de notre mesure, nous utilisons la librairie `Wf_nat` standard de Coq. Cette librairie fournit la fonction $ltof$ de construction d'une relation à partir d'une fonction quelconque de type $A \to nat$ et la preuve que cette relation est bien fondée. Dans notre cas $(ltof\_\ mesure)$ est la relation qui au couple $(S_1,\ S_2)$ associe la relation d'ordre $mesure(S_1) < mesure(S_2)$, et $(well\_founded\_ltof\ \_\ mesure)$ est la preuve que la relation $(ltof\_\ mesure)$ est bien fondée.
Enfin, pour utiliser l'outil `Recursive Definition` de Balaa&Bertot, nous devons fournir l'équation de point fixe de notre fonction récursive générale; dans notre cas:

$$\forall S, Pmov\ S = match\ S\ with\ ((nil, nil), \_)\ \Rightarrow S \mid \_ \Rightarrow Pmov\ (stepf\ S)\ end.$$

On a alors tous les éléments pour appeler l'outil `Recursive Definition`:

Recursive Definition $Pmov$ $(State {\to} State)$ $(ltof\ \_\ mesure)$ $(well\_founded\_ltof\ \_\ mesure)$ $stepf\_dec$
$(\forall\ S,\ Pmov\ S = match\ S\ with\ ((nil,nil),\_) \Rightarrow S \mid \_ \Rightarrow Pmov\ (stepf\ S)\ end)$.

où $stepf\_dec$ est le nom du lemme **8** dans Coq. Le système génère et prouve alors le théorème de terminaison d'énoncé:
$Pmov\_terminate$:

$$\forall x : State,\ \{v \mid \exists\, p\ ,\ (\forall\, k\ :\ nat,\\ p\ <\ k\ \to\ \forall\, def\ :\ State -> State,\ iter\ (State -> State)\ k\ Pmov\_F\ def\ x\ =\ v)\}.$$

Où $Pmov\_F$ est la fonctionnelle appliquant l'équation de point fixe à son argument (ici def). On remarque que ce théorème donne en prime le résultat du calcul dans v, d'où la fonction récursive générale:

$$Pmov\ x = (let\ (v,\ \_)\ :=\ Pmov\_terminate\ x)\ in\ v.$$

Le système fournit aussi une nouvelle équation de point fixe adaptée au théorème de terminaison, qui sera éventuellement utilisée pour les preuves ultérieures:
$Pmov\_equation$:

```
∀ S, Pmov S =
  (let (p, _) := S in let (m0, m1) := p in                    (*S = (m0, m1, _)*)
   match m0 with
   | nil ⇒ match m1 with
     | nil ⇒ S
     | _ :: _ ⇒ Pmov (stepf S)
     end        | _ :: _ => Pmov (stepf S)
   end)
```

Notons que la nouvelle équation de point fixe a été obtenue par dépliage des cas de l'équation donnée en argument à `Recursive Definition`.
On prouve alors facilement le théorème de correction de la fonction récursive Pmov:

Théorème 3 (Conservation de l'exécution de Pmov) *l'exécution de Pmov sur un triplet bien formé produit un triplet ayant la même exécution:* $Ok(S) \Rightarrow \forall \rho,\ Exec(S, \rho) \equiv Exec(Pmov(S), \rho)$

Preuve par élimination sur S en utilisant la relation bien fondée associée à la fonction $mesure$:($ltof\ \_\ mesure$), (i.e. en supposant le théoreme vrai pour tout y de mesure inférieure à x, en particulier un prédécesseur de x par $stepf$), puis en utilisant l'équation de point fixe $Pmov\_equation$, et en se ramenant à la propriété de conservation de l'exécution par la relation $\hookrightarrow^*$ (par le théoreme **2** de compatibilité de $stepf$ et $\hookrightarrow$).
Enfin, pour conclure, l'étude de la version fonctionnelle de notre algorithme, nous avons utilisé l'extraction de Coq pour extraire en Ocaml le code de notre fonction $Pmov$ et obtenir ainsi un code fonctionnel certifié de l'affectation parallèle, intégrable au compilateur.

6. Conclusion

Nous avons démontré la correction et la terminaison d'un algorithme de sérialisation d'une affectation parallèle. Cet algorithme ne correspond pas *exactement* à l'algorithme impératif proposé par Xavier Leroy.
D'une part, l'algorithme fonctionnel détecte un cycle qu'à partir de l'arc ayant initié la récursion à la fonction `move_one` (i.e. le dernier élément de la liste $Being\_moved$), alors que l'algorithme impératif recherche le cycle à priori n'importe où dans cette liste $Being\_moved$. Le fait qu'il est suffisant de rechercher le cycle à partir du fond de pile est induit dans la preuve de correction de la relation $\triangleright$ (cas de la règle Pop où l'on prouve que tous les éléments de σ outre le dernier ne peuvent pas avoir d_n comme source). C'est cette propriété qui permet de dégager le bon invariant et ainsi de simplifier la preuve. Pour cette affectation parallèle, la difficulté réside plus à trouver la bonne formalisation de l'algorithme que la dans la preuve elle-même.
D'autre part, le coté impératif de l'algorithme (effets de bord) n'est pas abordé. Nous étudions la possibilité d'utiliser Why [7, 6] mais les boucles imbriquées et le nombre de tableaux traités (`src`, `dst` et `status`) impliquent l'usage d'invariants de grosse taille, et les énoncés d'obligations de preuve générés par le système sont très gros et peu manipulables. D'autre part, les effets de bords dans les vecteurs doivent nécessairement faire intervenir des lemmes de *non aliasing*, difficiles à prouver. Mais nous ne désespérons pas de parvenir à prouver l'algorithme impératif en utilisant Why, tout en espérant pouvoir réutiliser les preuves décrites ici.

Remerciements

Nous remercions Xavier Leroy, qui nous a donné l'idée de cette étude et nous a fourni l'algorithme de départ en Caml.

Bibliographie

[1] A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers : principles, techniques and tools.* Addison Wesley, 1986.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[3] L'équipe Concert. L'action de recherche coopérative Concert. `http://www-sop.inria.fr/lemme/concert`.

[4] Coq development team. The *Coq* proof assistant. Documentation, system download. Contact: `http://coq.inria.fr/`.

[5] Antonia Balaa et Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.

[6] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[7] Jean-Christophe Filliâtre. The why software. `http://why.lri.fr/`.

[8] Cathy May. The parallel assignment problem redefined. *IEEE Transactions on Software Engineering*, 15(6):821–824, June 1989.

[9] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.