Démonstration formelle de correction pour un compilateur

Yves Bertot

INRIA Sophia Antipolis
2004 route des Lucioles, BP 93
06902 Sophia Antipolis CEDEX

4 février 2003

Le langage source

- Le fragment impératif commun aux langages Pascal, Fortran, C.
- Pas de structures de données,
- types numériques et booléens,
- Programmation structurée: boucles, séquences, affectations, conditionnelles, pas de branchement,
- Source: le langage *imp* de Winskel[93].

Le langage cible

- Langage assembleur sans structure,
- Un seul type de données,
- Branchements inconditionnels et conditionnels,
- Pile d'opérandes et un registre,
- Les opérations binaires utilisent le registre et le sommet de pile,
- L'interaction avec la mémoire passe par le registre.

Le langage source, Structures de données

- Variables représentées par des indices (pas de table des symboles),
- Déclarations de variables: type et valeur initial (influencé par H. Goguen : « Typed operational semantics »),
- Instructions comme un type inductif,
- Représentation héritée d'un travail sur Centaur (D. Terrasse).

Le langage source, Structures de données *(suite)*

```
Inductive SEQUENCE: Set :=
 sequence: li_list -> SEQUENCE
with INST :Set :=
 coer_SEQUENCE_INST : SEQUENCE->INST | assign: ID->EXP->INST
| l_if : EXP -> INST -> INST -> INST
| while : EXP -> INST -> INST
with li_list:Set :=
  li_nil : li_list | li_cons : INST -> li_list -> li_list.
```

Le langage source, Structures de données *(suite)*

Example de représentation

```
while x > 0 do x := x - 1 done
```

représenté par

```
(while (l_gt (coer_ID_EXP (ident 0))
             (coer_VAL_EXP (coer_INT_VAL (int 0))))
   (assign (ident 0)
           (l_minus (coer_ID_EXP (ident 0))
                    (coer_VAL_EXP (coer_INT_VAL (int (S 0)))))))
```

Sémantique dynamique

– Présentation à la *sémantique naturelle*

$$\rho \vdash i \to \rho'$$

– équivalente à une sémantique opérationnelle structurelle pas-à-pas,
– Exécutable à l'aide de Prolog (mais pas ML ou C).

$$\frac{\rho \vdash e \to l\_true \qquad \rho \vdash i \mapsto \rho' \qquad \rho' \vdash (while\ e\ i) \mapsto \rho''}{\rho \vdash (while\ e\ i) \mapsto \rho''}$$

Sémantique dynamique *(suite)*

$$\frac{\rho \vdash e \rightarrow l\_true \qquad \rho \vdash i \mapsto \rho' \qquad \rho' \vdash (while\ e\ i) \mapsto \rho''}{\rho \vdash (while\ e\ i) \mapsto \rho''}$$

```
Inductive exec : DECLS -> INST -> DECLS -> Prop :=
   ...
exec_while_true: (D',D'',D:DECLS;E:EXP;I:INST)
 (eval D E (coer_BOOL_VAL l_true)) ->
 (exec D I D') -> (exec D' (while E I) D'') ->
 (exec D (while E I) D'')
   ...
```

Autres études sur le même langage

- Avec Ranan Fraer,
  - Subject reduction,
  - Equivalence avec la sémantique pas-à-pas,
  - Transformations de programmes,
  - TAPSOFT'95,
- Bertot&Capretta&Das Barman02: pour une exécution fonctionnelle.

Machine virtuelle et jeu d'instructions

- 4 composants: mémoire (liste de nombres naturels), registre (1 nombre naturel), pile (liste de nombres naturels), position.
- la position dans le programme est donnée par une liste d'instructions,
- *add, sub, and, or, not, eq, gt, immediate*, *push, pop*, *label, branch_if_0, goto*, *load, store*
- Interprétation des valeurs entières comme des booléens (0 pour false).
- Fonctions pour calculer un nouvel état: `new_register`, `new_mem`, `new_stack`, `new_pc`.

Controle de flux

- *label* est une instruction sans effet,
- Tous les branchements se font vers des étiquettes,
- Branchements inconditionnels *goto* et conditionnels *branch_if_0*,
- Le branchement est modélisé à l'aide d'une fonction de recherche linéaire dans le programme (déterministe mais partielle).

Prédicat d'exécution

$$(execute\ pg\ s\ pg'\ s')$$

- pg est le programme complet (ne change pas au cours de l'exécution),
- s est l'état initial,
- pg' est la séquence d'instruction à partir du point courant,
- s' est l'état final.

Théorèmes de structuration

- Accoler deux programmes: quand cela correspond-il à combiner leurs comportements?
- Toutes les étiquettes doivent être utilisées une seule fois,
- Chaque programme doit fournir la destination de tous les branchements,
- Prédicats *unique* et *self_contained* (cohérents).

Isolation

– **Théorème** les fragments cohérents peuvent être étudiés séparément:

$$\forall p_1,p_2,p_3,s,s''.\ (unique\ p_1@p_2@p_3) \wedge (self\_contained\ p_2) \Rightarrow$$
$$\exists s'.\ (execute\ p2,s,p_2,s') \wedge (execute\ p_1@p_2@p_3,s',p_3,s'')$$
$$\Leftrightarrow$$
$$(execute\ p_1@p_2@p_3,s,p_2@p_3,s'')$$

– Théorème démontré dans un cadre plus abstrait (machine abstraite à trois instructions).

Fonctions récursives structurelles

```
Fixpoint compile_inst [i:INST]:nat -> (list Assembly)*nat :=
[n:nat]Cases i of
  (assign (ident x) e) =>
     ((app (compile_exp e) (store x)), n)
| (l_while e i1) =>
  let (pg, n') = (compile_inst i1 (n+2)) in
  ((as_label n)@(compile_exp e)@(as_branch_if_0 (n+1))@
   pg@(goto n)@(as_label (n+1)), n')
  ...
```

Diagrammes de commutation

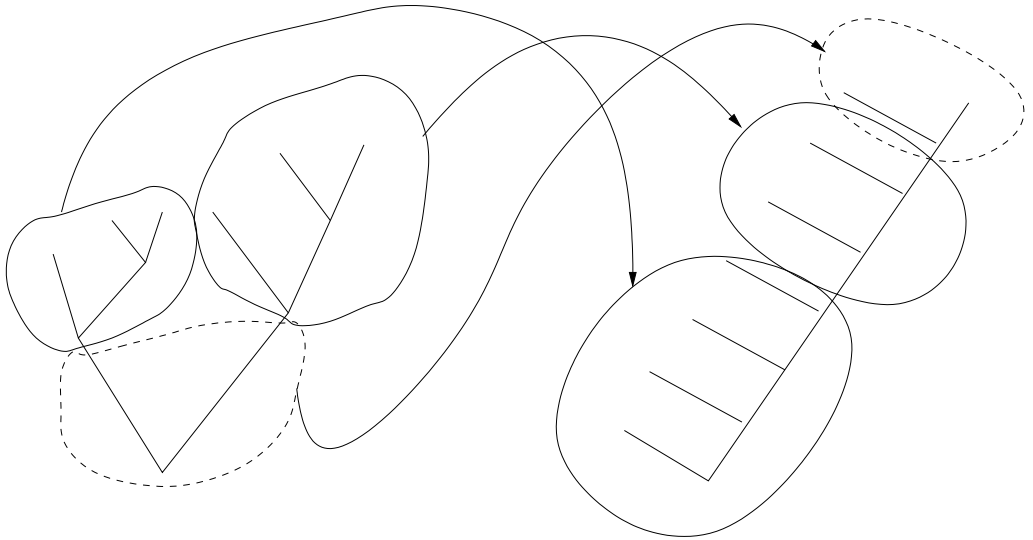
– Le comportement du programme compilé est fidèle au comportement du programme source.

– Lorsque le programme compilé retourne un résultat, celui-ci aurait été obtenu par l'exécution du programme source.

Dérivations et correspondances

- Les règles d'inférence sont les constructeurs d'un type de données arborescent: les dérivations,
- Les dérivations sont des traces de calculs,
- On peut raisonner par récurrence sur ces dérivations (Despeyroux87).

Dérivations et correspondances *(suite)*



Récurrence simple et générale

– Principes de récurrence simple.

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

```
nat_ind:∀P:nat->Prop.(P(O)∧(∀m:nat. P(m)⇒P(m+1)))⇒ ∀n:nat. P(n)
```

– Le principe de récurrence peut aussi être compris comme une fonction récursive qui fabrique des démonstrations.

Récurrence simple et générale *(suite)*

```
Inductive ≤ : nat -> nat -> Prop :=
 le_n : ∀n:nat. n≤n | le_S : ∀n,m:nat. n≤m⇒n≤m+1.
leq_ind : ∀ P :nat->nat->Prop.
∀n:nat. P(n,n)∧(∀n,m:nat. (n≤m∧ P(n,m)) ⇒ P(n,bm+1))⇒
∀n,m:nat. n≤m⇒P(n,m)
```

- Le principe de récurrence n'utilise qu'une sous preuve directe,
- Appel récursif seulement sur le prédecesseur.

Principes de récurrence généraux

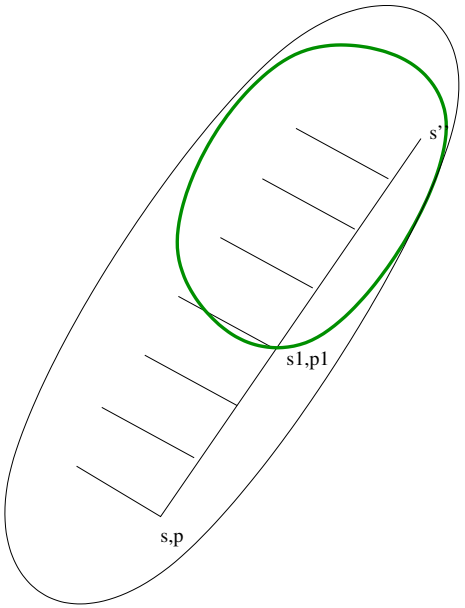
$\forall$P:nat->Prop. ($\forall$x:nat.($\forall$y:nat. y<x$\Rightarrow$ P(y))$\Rightarrow$P(x))$\Rightarrow$ $\forall$z:nat. P(z)

- Appel récursif autorisé sur tous les entiers strictement plus petits.
- fait appel à une relation auxiliaire : x < y $\Leftrightarrow$ x=S($S^k(y)$)

Analogie sur les exécutions de programmes

$(\forall s,p,s'.$
$(\forall s_1,p_1.(execute'\ pg\ s\ p\ s_1\ p_1) \Rightarrow (execute\ pg\ s_1\ p_1\ s') \Rightarrow (P\ s_1\ p_1\ s')) \Rightarrow$
$(P\ s\ p\ s')) \Rightarrow$
$\forall s,p,s'.(execute\ pg\ s\ p\ s')$

Analogie sur les exécutions de programmes *(suite)*



Premier théorème

$\forall\ i,d,d',s,n.$
$(exec\ d\ i\ d') \Rightarrow (check\_inst\ d\ i) \Rightarrow (check\_decls\ d) \Rightarrow (coherent\ d\ s) \Rightarrow$
$\exists\ s'.\ (execute\ (compile\_inst'\ i\ n)\ s\ (compile\_inst'\ i\ n)\ s') \wedge (coherent\ d'\ s')$

Deuxième théorème

$\forall$ *d,d', i,s,s',n. (check_inst d i)*$\Rightarrow$
(execute (compile_inst' i n) s (compile_inst' i n) s')$\Rightarrow$
(types_compatible d d')$\Rightarrow$*(check_decls d')*$\Rightarrow$*(coherent d' s)*$\Rightarrow$
$\exists$ *d". (exec d' i d")*$\wedge$*(coherent d" s')* $\wedge$*(types_ compatible d d").*

Conclusion

- 2 hommes mois, la majeure partie pour le deuxième théorème,
- Une grande partie automatisable,
- Modèles exécutables, compilateur extractible.