

Une sémantique de C
pour un compilateur certifié
en Coq

Sandrine Blazy et Xavier Leroy

INRIA Rocquencourt

Plan

1. Compilation certifiée et langage C.
2. Syntaxe abstraite
3. Sémantique dynamique.
4. Modèle mémoire.
5. Sémantique statique.
6. Évolutions et problèmes rencontrés.

1. Compilation certifiée et langage

Projet Concert

Réalisation d'un compilateur C certifié:

- utilisable dans le domaine de l'embarqué,
- modérément optimisant,
- spécifié en Coq et accompagné d'une preuve de sémantique.

ARC Concert: janvier 2003 à décembre 2004
(INRIA Cristal + Lemme et CNAM Cedric)

Concert: quelques chiffres

- Compilateur C vers assembleur Power PC.
- 5 langages intermédiaires.
- Analyses de flots de données (élimination des sous-expressions communes, propagation de c
- 41200 lignes écrites en Coq (preuves: 55%, sémantiques: 10%, passes du compilateur: 12
- 1900 lignes écrites en Caml.
- Temps de compilation \simeq gcc -O1 (de 50 à 200
- Extraction de 30300 lignes de Caml.

État de l'art

- M.Norrish (1996): sémantique non déterministe (les valeurs portent les types), HOL.
- Papaspyrou: sémantique dénotationnelle.
- les russes (2003): C à la Pascal.
- Fail-safe C (Oiwa et Sekiguchi, 2004).
- CCured, (Necula et al., POPL 2002): système pour C + analyses statiques calculant comme les pointeurs.
- Quelques pratiques de programmation C:
 - Norme ANSI: beaucoup de recommandations
 - MISRA (industrie automobile, 2004).
 - Quelques pratiques industrielle (ex. Dassault) restreint (usage très limité des pointeurs).

Verisoft (W.Paul, SEFM 2005)

- C à la Pascal. Ne sont pas considérés: union, de pointeurs, effets de bord, cast.
- Ajout d'un opérateur & et d'une fonction new.
- Un seul type de boucle (while).
- continue et break ??
- Transformations fondées sur une sémantique à
+ sémantique à petits pas avec preuve d'équi
- Modèle mémoire de bas niveau, mémoire finie

2. Syntaxe

Types et valeurs

```
Inductive signedness : Set :=  
  | Signed: signedness  
  | Unsigned: signedness.
```

```
Inductive intsize : Set :=  
  | I8: intsize  
  | I16: intsize  
  | I32: intsize.
```

```
Inductive type: Set :=  
  | Tvoid: type  
  | Tint: intsize -> signedness -> type  
  | Tfloat: floatsize -> type  
  | Tpointer: type -> type  
  | Tarray: type -> Z -> type  
  | Tfunction: typelist -> type -> type
```

```
Inductive val: Set :=  
  | Vundef: val  
  | Vint: int -> val  
  | Vfloat: float -> val  
  | Vptr: block -> int -> val.
```

Expressions

```
Inductive expr: Set :=
  | Expr: expr_descr -> type -> expr

with expr_descr: Set :=
  | Econst_int: int -> expr_descr
  | Econst_float: float -> expr_descr
  | Evar: ident -> expr_descr
  | Eunop: unary_operation -> expr -> expr_descr
  | Ederef: expr -> expr_descr
  | Eaddrof: expr -> expr_descr
  | Epreincrdecr: incrdecr -> expr -> type -> expr_descr
  | Epostincrdecr: incrdecr -> expr -> type -> expr_descr
  | Ebinop: binary_operation -> expr -> expr -> expr_descr
  | Ecast: type -> expr -> expr_descr
  | Eindex: expr -> expr -> expr_descr
  | Eassign: expr -> expr -> expr_descr
  | Eassignop: binary_operation -> expr -> expr -> type -> expr_descr
  | Ecall: expr -> exprlist -> expr_descr
  | Ecomma: expr -> expr -> expr_descr
  | Eandbool: expr -> expr -> expr_descr
  | Eorbool: expr -> expr -> expr_descr
  | Econdition: expr -> expr -> expr -> expr_descr
  | Esizeof: type -> expr_descr
  ...
```

Instructions

```
Inductive statement: Set :=
  | Sskip : statement
  | Sexpr : expr -> statement
  | Ssequence : statement -> statement -> statement
  | Sifthenelse : expr -> statement -> statement -> statement
  | Swhile : expr -> statement -> statement
  | Sdowhile : expr -> statement -> statement
  | Sfor : option expr -> option expr -> option expr -> statement
  | Sbreak : statement
  | Scontinue : statement
  | Sreturn : option expr -> statement.
```

```
Record function : Set := mkfunction {
  fn_return: type;
  fn_params: list (ident * type);
  fn_vars: list (ident * type);
  fn_body: statement }.
```

```
Record program : Set := mkprogram {
  prog_funct: list (ident * function);
  prog_defs: list (ident * type);
  prog_main: ident }.
```

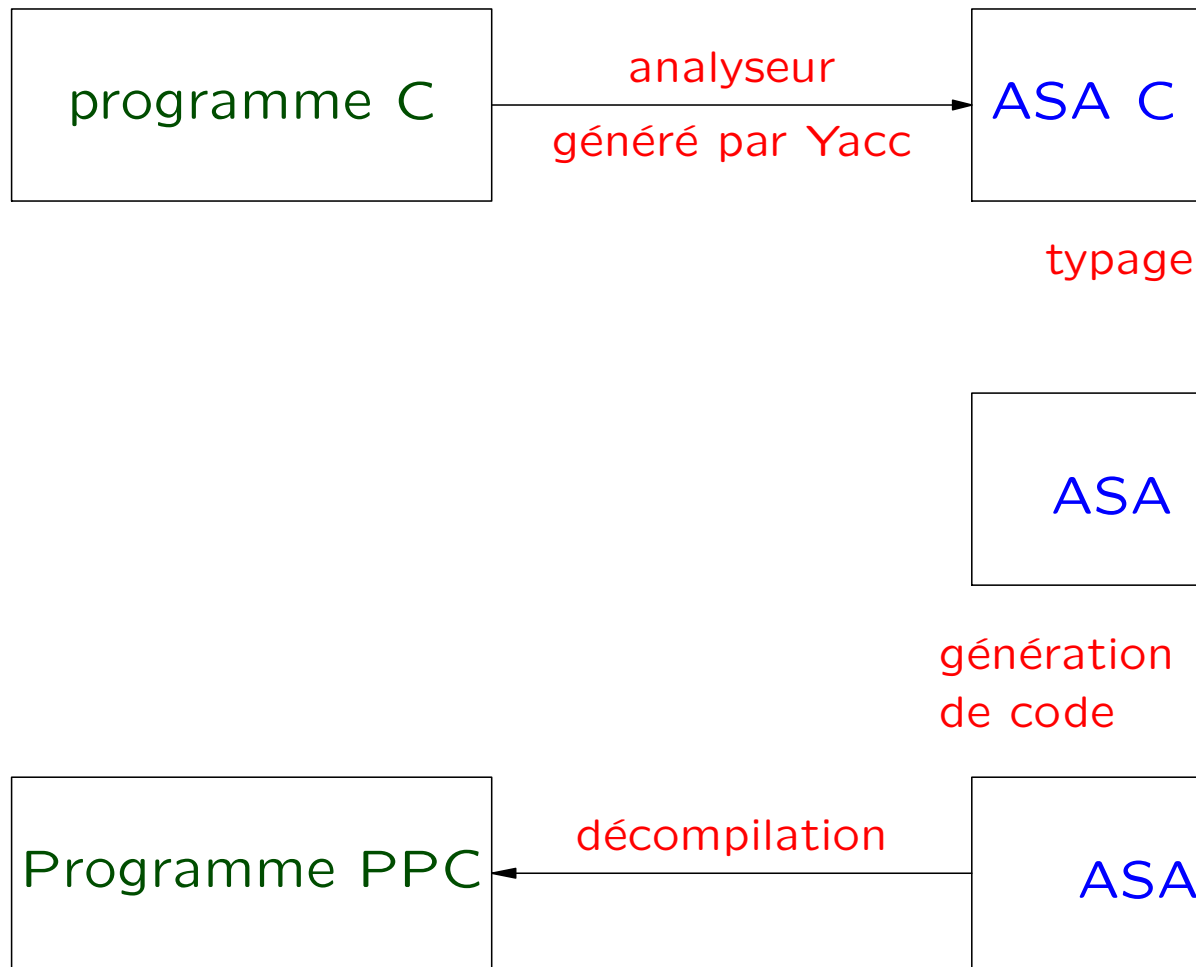
Fragment de C ANSI considéré

- pas d'instructions de saut (goto, setjmp et longjmp)
- nombre fixe d'arguments d'une fonction,
- pas de blocs, switch, union, struct, enum, type initialisations de variables, qualificateurs de type

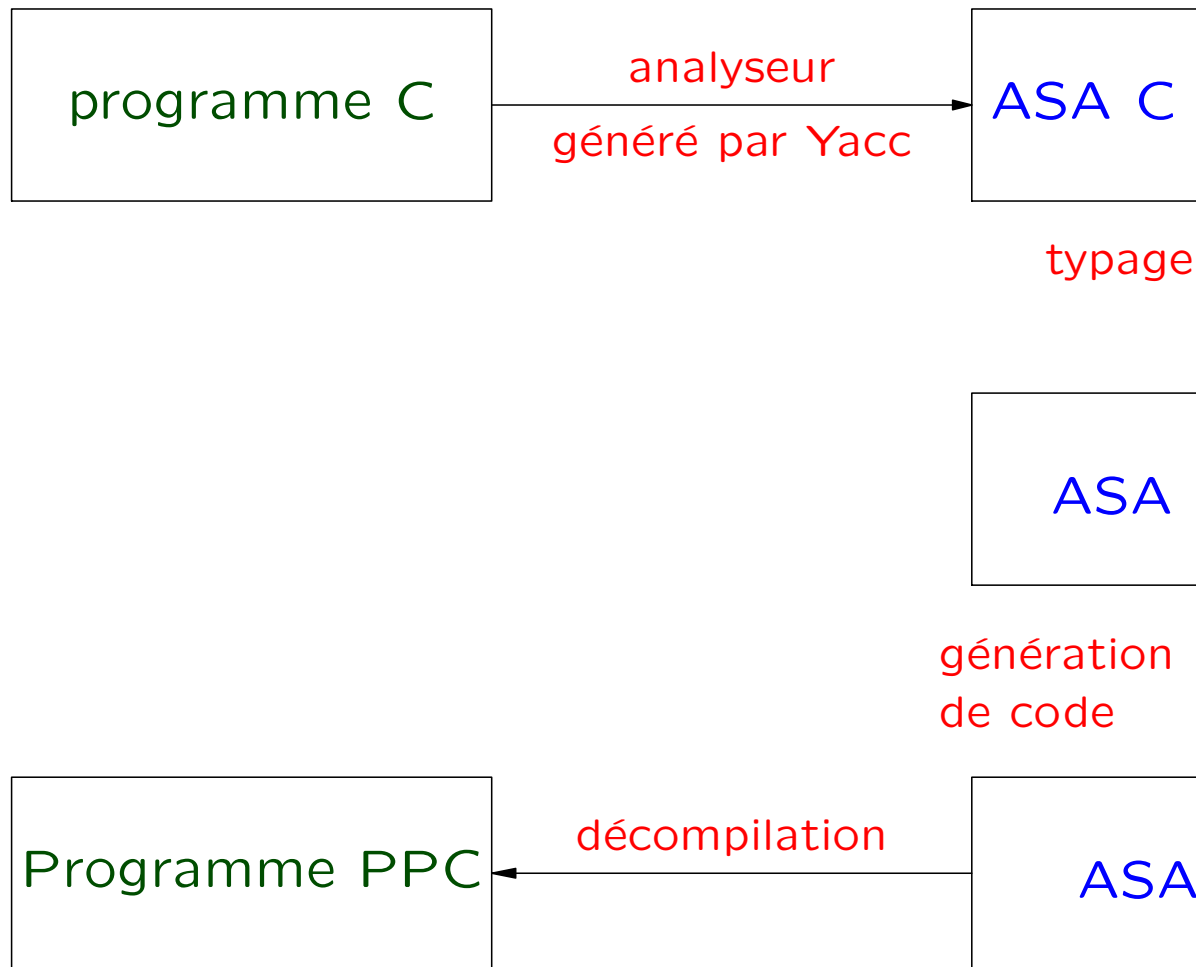
mais

- boucles,
- arithmétique de pointeurs,
- pointeurs sur fonctions.

Architecture du compilateur



Architecture du compilateur



3. Sémantique dynamique de

Sémantique dynamique de C

Le langage C est à la fois:

- de **bas niveau** (l'arithmétique de pointeurs auto-déclarés et le chevauchement de zones mémoire),
- de **haut niveau** (des conditions d'isolation des processus doivent être assurées).

Choix:

- sémantique naturelle à grand pas,
- déterministe (évaluation de gauche à droite de l'expression),
- définie en Coq par des prédicats mutuellement récurrents,
- Plongement profond dans Coq.

Jugements d'évaluation

- $E \vdash a^\tau, M \Rightarrow v, M'$ (expressions)
- $E \vdash [a^\tau], M \Rightarrow v, M'$ (expressions optionnelles)
- $E \vdash \vec{a}^\tau, M \Rightarrow \vec{v}, M'$ (liste d'expressions)
- $E \vdash a^\tau, M \Rightarrow l, M'$ (valeurs gauches)
- $G \vdash f(\vec{v}), M \Rightarrow v, M'$ (appels de fonctions)
- $E \vdash s, M \Rightarrow out, M'$ (instructions)
 $out ::= \text{normal} \mid \text{break} \mid \text{continue} \mid \text{return}[val]$
- $\vdash p \Rightarrow v$ (programmes)

Évaluation des valeurs gauches

$E \vdash a^\tau, M \Rightarrow l, M'$

...

```
with eval_lvalue: env -> mem -> expr ->
    mem -> block -> int -> Prop :=
| eval_Lvar_local: forall e m id l ty,
    e ! id = Some l ->
    eval_lvalue e m (Expr (Evar id) ty) m l Int.zero
| eval_Lvar_global: forall e m id l ofs ty,
    e ! id = None ->
    Genv.find_symbol ge id = Vptr l ofs ->
    eval_lvalue e m (Expr (Evar id) ty) m l ofs
| eval_Lderef: forall e m m1 a ofs ty l,
    eval_expr e m a m1 (Vptr l ofs) ->
    eval_lvalue e m (Expr (Ederef a) ty) m1 l ofs
| eval_Lindex: forall e m a1 m1 v1 a2 m2 v2 l ofs ty,
    eval_expr e m a1 m1 v1 ->
    eval_expr e m1 a2 m2 v2 ->
    addall v1 (typeof a1) v2 (typeof a2) = Some(Vptr l ofs) ->
    eval_lvalue e m (Expr (Eindex a1 a2) ty) m2 l ofs
```

Évaluation des expressions

$E \vdash a^\tau, M \Rightarrow v, M'$

```
Inductive eval_expr: env -> mem -> expr ->
    mem -> val -> Prop :=
| eval_Eindex: forall e m t i m1 loc ty ofs v,
  eval_lvalue e m (Expr (Eindex t i) ty) m1 loc ofs ->
  load_value_of_type ty m1 loc ofs = Some v ->
  eval_expr e m (Expr (Eindex t i) ty) m1 v
| eval_Eassign_op: forall e m vg a m1 m2 v1 v2 v3 op m3
  eval_lvalue e m vg m1 loc ofs ->
  load_value_of_type ty m1 loc ofs = Some v1 ->
  eval_expr e m1 (Expr a ta) m2 v2 ->
  eval_binary_operation op v1 ty v2 ta top m2 = Some v3 ->
  cast_value v3 top ty v ->
  store_value_of_type ty m2 loc ofs v = Some m3 ->
  eval_expr e m (Expr (Eassignop op vg (Expr a ta) top) m3) v
| eval_Epreincrdecr: forall e m a m1 loc ofs ty vbefore :
    vafter m2,
  eval_lvalue e m a m1 loc ofs ->
  load_value_of_type ty m1 loc ofs = Some vbefore ->
  incr_or_decr inc vbefore ty = Some vinc ->
  cast_value vinc tres ty vafter ->
  store_value_of_type ty m1 loc ofs vafter = Some m2 ->
  eval_expr e m (Expr (Epreincrdecr inc a tres) ty) m2
...

```

Évaluation d'une fonction

```
Inductive eval_expr: env -> mem -> expr ->
    mem -> val -> Prop :=
| eval_Ecall: forall e m a bl m1 m2 m3 vf vargs vres f t
    classify_fun (typeof a) = fun_case_f tyargs tyres ->
    eval_expr e m a m1 vf ->
    eval_exprlist e m1 bl tyargs m2 vargs ->
    Genv.find_funct ge vf = Some f ->
    type_of_function f = typeof a ->
    eval_funcall m2 f vargs m3 vres ->
    eval_expr e m (Expr (Ecall a bl) ty) m3 vres
...
with eval_funcall: mem -> function -> list val ->
    mem -> val -> Prop :=
| eval_funcall_intro: forall m f vargs vres e m1 m2 m3 l
    eval_decllist empty_env m (fn_params f ++ fn_vars f) ->
    set_parameters e m1 (fn_params f) vargs m2 ->
    exec_stmt e m2 f.(fn_body) m3 out ->
    outcome_result_value out f.(fn_return) vres ->
    eval_funcall m f vargs (free_list m3 lb) vres.
```

Exécution d'instructions

$E \vdash s, M \Rightarrow out, M'$

Inductive outcome: Set :=

- | Out_break: outcome
- | Out_continue: outcome
- | Out_normal: outcome
- | Out_return: outcome
- | Out_return_val: val -> type -> outcome.

Inductive exec_stmt: env -> mem -> statement ->
mem -> outcome -> Prop :=

- | exec_Sskip: forall e m,
exec_stmt e m Sskip m Out_normal
- | exec_Sexpr: forall e m a m1 v,
eval_expr e m a m1 v ->
exec_stmt e m (Sexpr a) m1 Out_normal
- | exec_Sseq_1: forall e m s1 s2 m1 m2 out,
exec_stmt e m s1 m1 Out_normal ->
exec_stmt e m1 s2 m2 out ->
exec_stmt e m (Ssequence s1 s2) m2 out
- | exec_Sseq_2: forall e m s1 s2 m1 out,
exec_stmt e m s1 m1 out ->
out <> Out_normal ->
exec_stmt e m (Ssequence s1 s2) m1 out

...

Exécution d'un programme

```
Definition exec_program (p: Cabs.program) (r: val) : Prop
  let ge := globalenv p in
  let m0 := init_mem p in
  exists f, exists m1,
  Genv.find_funct ge (Genv.find_symbol ge p.(prog_main)) =
  eval_funcall ge m0 f nil m1 r.
```

4. Modèle mémoire du compilat

Caractéristiques

- Modèle commun à tous les langages du comp
- Propriétés sur:
 - la géographie de la mémoire,
 - les opérations de gestion.
- Compromis entre bas niveau (tableau unique) (nécessité de considérer alias et chevaucheme
- Un bloc de mémoire est composé d'un nombre de cellules élémentaires.
- Une valeur est lue ou écrite en mémoire conna taille, son type et son signe ("memory chunk" déterminent le nombre de cellules élémentaire pour accéder à la valeur.
- La mémoire est infinie, mais la taille d'un bloc lors de la compilation.

Relations entre blocs de mémoire

Définition d'une **relation d'équivalence** et d'une relation (réflexive) de **confinement** entre blocs.

```
Definition block_agree (b: block) (lo hi: Z) (m1 m2: mem)
...

```

```
Definition extends (m1 m2: mem) :=
...

```

Opérations de gestion de la mémoire: si

Mémoire infinie: alloc et free n'échouent jamais.

```
Definition alloc (m: mem) (lo hi: Z) : mem * Z :=
```

```
...
```

```
Definition free (m: mem) (b: block) : mem :=
```

```
...
```

load et store peuvent échouer.

Exemple: stockage dans un bloc d'une valeur trop
ce bloc.

```
Definition load (chunk: memory_chunk) (m: mem) (b: block)  
              : option val :=
```

```
...
```

```
Definition store (chunk: memory_chunk) (m: mem) (b: block)  
               (ofs: Z) (v: val) : option mem :=
```

```
...
```

Opérations de gestion de la mémoire: p

Les opérations préservent la relation d'équivalence

Theorem load_agree:

```
forall (chunk: memory_chunk) (m1 m2: mem)
      (b: block) (lo hi: Z) (ofs: Z) (v1 v2: val),
block_agree b lo hi m1 m2 ->
lo <= ofs -> ofs + size_chunk chunk <= hi ->
load chunk m1 b ofs = Some v1 ->
load chunk m2 b ofs = Some v2 ->
v1 = v2.
```

...

Les opérations préservent la relation de confinement

Theorem load_extends:

```
forall (chunk: memory_chunk) (m1 m2: mem) (b: block) (ofs: Z)
      (v: val)
extends m1 m2 ->
load chunk m1 b ofs = Some v ->
load chunk m2 b ofs = Some v.
```

...

Propriétés de bonne formation des va

Theorem load_store_same:

```
forall (chunk: memory_chunk) (m1 m2: mem) (b: block) (o
store chunk m1 b ofs v = Some m2 ->
load chunk m2 b ofs = Some (Val.load_result chunk v).
```

Theorem load_store_other:

```
forall (chunk1 chunk2: memory_chunk) (m1 m2: mem)
      (b1 b2: block) (ofs1 ofs2: Z) (v: val),
store chunk1 m1 b1 ofs1 v = Some m2 ->
b1 <> b2 \ / ofs2 + size_chunk chunk2 <= ofs1
      \ / ofs1 + size_chunk chunk1 <= ofs2 ->
load chunk2 m2 b2 ofs2 = load chunk2 m1 b2 ofs2.
```

...

Autres propriétés

Theorem `low_bound_store`:

```
forall (chunk: memory_chunk) (m1 m2: mem) (b b': block)
store chunk m1 b ofs v = Some m2 ->
low_bound m2 b' = low_bound m1 b'.
```

Theorem `store_alloc`:

```
forall (chunk: memory_chunk) (m1 m2: mem) (b: block) (o
alloc m1 lo hi = (m2, b) ->
lo <= ofs -> ofs + size_chunk chunk <= hi ->
exists m2', store chunk m2 b ofs v = Some m2'.
```

...

5. Sémantique statique de C

Typage statique

- Environnement de typage E_τ : map $id \mapsto \tau$.
- Fonctions de typage annotant les programmes
Exemple: $E_\tau \vdash a : [a'^\tau]$
Exemple d'erreur: le typage de `*a` échoue si `a` est un pointeur, ni un tableau.
- Utilisation d'un style monadique pour propager

```
Fixpoint type_expr (env: typing_env) (a: expr)
  {struct a} : option expr :=
match a with
| Cabsuntyped.Eassignop op b c =>
  do tb <- type_expr env b;
  do tc <- type_expr env c;
  do te <- type_binop op tb tc;
  make_assign tb te
...

```

Typage des instructions

```
Fixpoint type_statement (env: typing_env) (s: statementu)
  {struct s} : option statement :=
  match s with
  | Cabsuntyped.Sifthenelse e s1 s2 =>
    do te <- type_expr env e;
    if (is_scalar (typeof te)) then
      do ts1 <- type_statement env s1;
      do ts2 <- type_statement env s2;
      Some (Sifthenelse te ts1 ts2)
    else None
  ...
```

Monade d'erreur

Écriture en Coq

```
Definition bind (A B: Set) (f: option A) (g: A -> option B) : option B :=  
  match f with None => None | Some x => g x end.
```

```
Notation "'do' X <- A ; B" := (bind A (fun X => B)).
```

Utilisation intensive d'une tactique dédiée d'inversion

Exemple:

```
Lemma wt_expr_Econst_int:  
  forall (tenv: typing_env) (c: int) (ta: expr)  
    (TYPING: type_expr tenv (Econst_int c) = Some ta),  
  wt_expr tenv ta.
```

Proof.

```
intros until ta; intro TYPING; monadInv TYPING.
```

```
...
```

Difficultés

- Pseudo-hiérarchie entre les types (*cf.* intégral scalaire).
- Différentes notions: conversions implicites (pr **intégrale**, **conversion arithmétique**), compatibilité types.
- Opérateurs surchargés → beaucoup de cas particuliers
Exemple des opérateurs binaires.

```
+   pt + i   ou encore i + pt
-   pt1 - pt2 est de type entier
==  pt1 == pt2 est de type entier   (idem !=)
<   résultat de type entier         (idem >
%   résultat de type entier         (idem S
```

Typage d'un opérateur binaire

```
Definition typing_binop (op:binary_operation) (ta tb:type) : type :=
match op, ta, tb with
| Oshl , Tint ia _ , Tint ib _ => integrale_promotion ta
| o , Tpointer _ , Tint _ _ => if (additive_op o) then Some ta
| Oadd , Tint _ _ , Tpointer _ => Some tb
| Oadd , Tint _ _ , Tarray _ _ => Some tb
| ...
| Osub , Tpointer t1, Tpointer t2 => if (type_eq t1 t2)
                                     then Some (Tint I32 Signed)
                                     else None
| o , Tpointer t1 , Tpointer t2 => if (cmp_op o)
                                     then if (type_eq t1 t2)
                                             then Some (Tint I32 Signed)
                                             else None
                                     else None
| o , Tpointer _ , t => if ((egalitary_op o) && (cmp_to_ptr o))
                         then Some (Tint I32 Signed)
                         else None
| o , Tint _ _ , Tint _ _ => Some (good_type o (arithm_conversion o))
| o , _ , _ => if ((not_bool (integer_only_op o)) && (not_bool (integer_only_op o)))
                (is_arithm ta) && (is_arithm tb))
                then Some (good_type o (arithm_conversion o))
                else None
end.
```

Propriété de bon typage des programmes

Règles de typage vérifiant la cohérence des annotations

- Un nœud d'ASA est bien typé si ses fils le sont
- Quelques règles supplémentaires.

```
Inductive wt_expr: typing_env -> expr -> Prop :=
  | Wt_const_int:  forall tenv c,
    wt_expr tenv (Expr (Econst_int c) (Tint I32 Signed))
  | Wt_const_float:  forall tenv c,
    wt_expr tenv (Expr (Econst_float c) (Tfloat F64))
  | Wt_var:  forall tenv ty id,
    tenv ! id = Some ty ->
    wt_expr tenv (Expr (Evar id) ty)
  | Wt_binop:  forall tenv a1 op a2 ty,
    wt_expr tenv a1 ->
    wt_expr tenv a2 ->
    wt_expr tenv (Expr (Ebinop op a1 a2) ty)
  | Wt_andbool : forall tenv a1 a2,
    wt_expr tenv a1 ->
    wt_expr tenv a2 ->
    wt_expr tenv (Expr (Eandbool a1 a2) (Tint I32 Signed))
  ...
```

Vérifications supplémentaires

- Les paramètres et variables locales d'une fonction sont distincts deux à deux.
- main est du bon type (si elle existe dans le programme).

```
Inductive wt_function: typing_env -> function -> Prop :=
| Wt_fun: forall e1 e2 fargs fbody fret fvars,
  unique_vars (fargs++fvars) ->
  make_types_of_list_vars (fargs++fvars) e1 = e2 ->
  wt_statement e2 fbody ->
  wt_function e1 (mkfunction fret fargs fvars fbody).
```

```
Definition wt_function_list
  (gtenv: typing_env) (functs: list (ident * function))
  forall id f, In (id, f) functs -> wt_function gtenv f.
```

```
Inductive wt_program: program -> Prop :=
| Wt_prog: forall e idmain lf decls,
  make_types_of_list_vars decls empty_typing_env = e ->
  wt_function_list e lf ->
  (forall fmain,
    In (idmain, fmain) lf ->
    type_of_function fmain = Tfunction Tnil (Tint I32)) ->
  wt_program (mkprogram lf decls idmain).
```

Propriétés du typage

- Bonne formation des types

Theorem `typing_expr_wtyped`: forall tenv a ta,
type_expr tenv a = Some ta -> wt_expr tenv ta.

- Pas de préservation du typage lors de l'évaluation
Conversion d'un pointeur en un entier: seul le
(Vptr b ofs) de type (Tpointer t) est converti
(Vptr b ofs) de type (Tint I32 Signed).

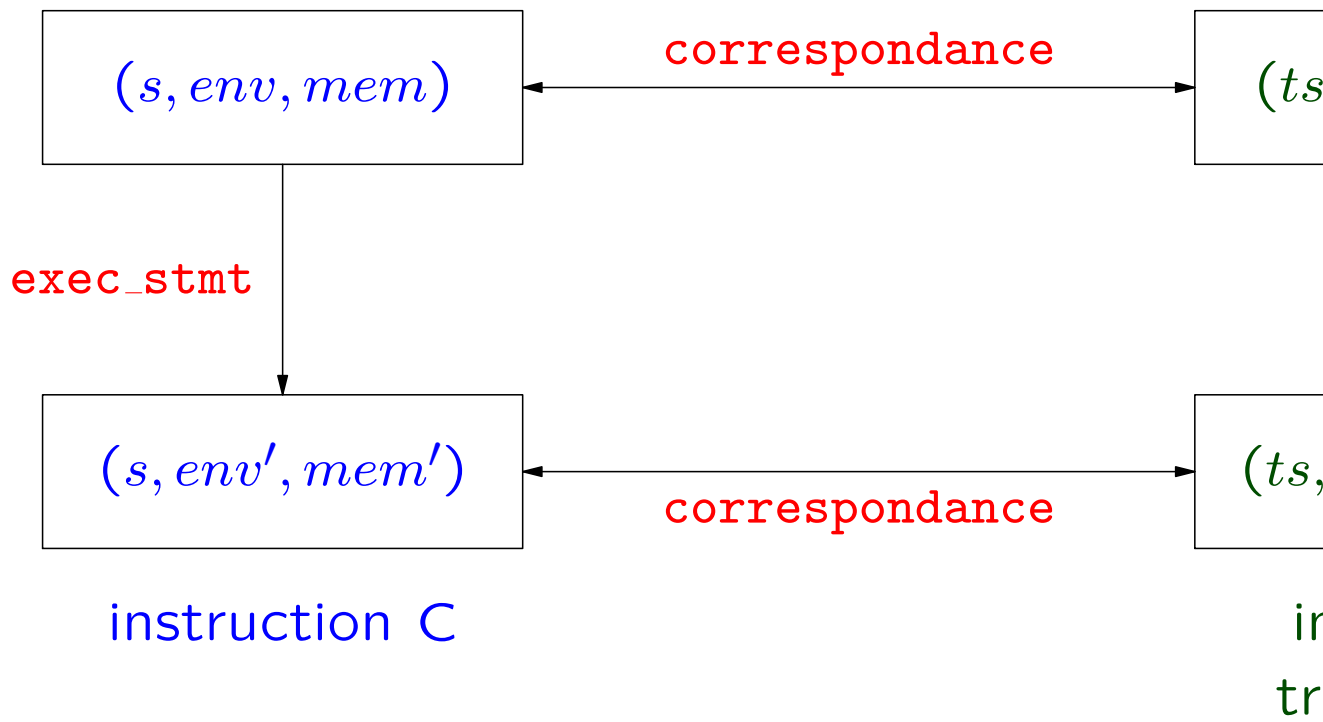
Bilan sur le typage

Théorème de correction sémantique

```
Theorem compiler_correct: forall pu p tp n,  
  type_program pu = Some p ->  
  compile_program p = Some tp ->  
  Csem.exec_program p (Vint n) ->  
  PPC.exec_program tp (Vint n).
```

Décomposition en autant de lemmes que de transformations de programmes dans la chaîne de compilation.
Chacun de ces lemmes est prouvé par induction sur le programme initial p (**lemmes de simulation**).

Exemple de diagramme de simulation



6. Évolutions et problèmes renco