CompCert pour les nuls

Une introduction à la vérification formelle de compilateurs en Coq

Sandrine Blazy[1] Xavier Leroy[2]

[1]U. Rennes 1

[2]INRIA Paris-Rocquencourt

2011-12-08

Plan

1. Le langage de spécification de Coq
2. Le langage IMP et ses sémantiques
3. Un compilateur vérifié pour le langage IMP
4. L'assembleur PowerPC en CompCert
5. Le langage CompCert C
6. Le théorème de préservation sémantique de CompCert

Première partie I

Le langage de spécification de Coq

(d'après Yves Bertot et Pierre Castéran)

Qu'est-ce que Coq ?

- Un langage de programmation
- Un outil de développement de preuves

Quel usage de Coq ?
Décrire :

- des données,
- des opérations,
- des propriétés,
- des preuves.

Décrire des données

Énumération

- de toutes les valeurs possibles,
- en distinguant un nombre fini de cas

Structuration

- Chaque cas est constitué de différents sous-cas.
- Similaire à la notion d'enregistrement (« record »).

Description éventuellement récursive

- « diviser pour régner »
- reconnaître une répétition infinie

Décrire des opérations

Programmation fonctionnelle : chaque opération est décrite par une fonction

Des valeurs de sortie sont produites à partir de valeurs en entrée de la fonction. $\rightarrow$ Pas de modification !

Programmation guidée par les cas définissant les structures de données

Éviter les valeurs indéfinies

- Tous les cas doivent être traités.
- La terminaison des calculs est assurée.

Programmation plus sûre

Décrire des propriétés

Langage prédéfini : and, or, forall, exists, ...

Possibilité d'exprimer de manière non ambigue des propriétés relatives à des fonctions

- Exemple
  Chaque fois que $f(x)$ *vaut* `true`, $g(x)$ *est un nombre premier.*

Schéma général pour définir de nouvelles propriétés : prédicats inductifs

- Exemple
  L'ensemble des nombres pairs est le plus petit ensemble E *tel que* $0 \in E$ *et* $x \in E \Rightarrow x + 2 \in E$

Le langage de spécification de Coq

1. Premiers pas en Coq
2. Un peu de logique
3. Types inductifs et fonctions récursives
4. Prédicats inductifs
5. Prédicats coinductifs

Premiers pas en Coq

Nommer une expression

```
Definition trois := 3.
trois is defined
```

Vérifier qu'une expression est bien formée

```
Check trois.
trois : nat
```

Calculer une valeur

```
Compute trois.
= 3 : nat
```

Définir des fonctions, par exemple des expressions dépendant d'une variable

```
Definition add3 (x : nat) := x + 3.
add3 is defined
```

Le type des valeurs

La commande Check est utilisée pour vérifier qu'une expression est bien formée.

- Elle renvoie le type de cette expression.
- Le type indique dans quel contexte l'expression peut être utilisée.

```
Check 2 + 3.
2 + 3 : nat

Check 2.
2 : nat

Check (2 + 3) + 3.
(2 + 3) + 3 : nat
```

Le type des fonctions

La valeur add3 n'est pas un entier naturel.

```
Check add3.
add3 : nat -> nat
```

La valeur add3 est une fonction.

- Elle attend en entrée un entier naturel.
- Elle produit en sortie un entier naturel.

```
Check add3 + 3.
Error the term "add3" has type "nat -> nat"
while it is expected to have type "nat"
```

Appliquer une fonction

Appliquer une fonction se fait par juxtaposition

- Les parenthèses ne sont pas obligatoires.

```
Check add3 2.
add3 2 : nat

Check add3 (add3 2).
add3 (add3 2) : nat
```

Fonctions à plusieurs arguments

Utiliser différentes variables lors de la définition

```
Definition s3 (x y z : nat) := x + y + z.
s3 is defined

Check s3.
s3 : nat -> nat -> nat -> nat
```

On peut appliquer partiellement une telle fonction, obtenant une fonction qui attend le reste des arguments.

```
Check s3 2.
s3 2 : nat -> nat -> nat

Check s3 2 1.
s3 2 1 : nat -> nat
```

Les fonctions sont des valeurs.

- La valeur add3 2 est un entier naturel.
- La valeur s3 2 est une fonction.
- La valeur s3 2 1 est une fonction, comme l'est add3.

Fonctions passées en arguments

Une fonction peut avoir des fonctions comme arguments.

```
Definition rep2 (f : nat -> nat) (x:nat) := f (f x).
rep2 is defined

Check rep2.
rep2 : (nat -> nat) -> nat -> nat

Definition rep2on3 (f : nat -> nat) := rep2 f 3.

Check rep2on3.
rep2on3 : (nat -> nat) -> nat
```

Le langage de spécification de Coq

1. Premiers pas en Coq
2. Un peu de logique
3. Types inductifs et fonctions récursives
4. Prédicats inductifs
5. Prédicats coinductifs

Le Type Prop

Le type prédéfini Prop est celui des propositions logiques.

Par exemple, les propositions vraies et fausses sont des constantes de type Prop.

```
Check True.
True : Prop

Check False.
False : Prop
```

Variables propositionnelles

Une variable propositionnelle est une variable de type `Prop`.
La commande `Variable` permet de déclarer une nouvelle variable propositionnelle.

```
Section logique_prop.
Variable il_pleut : Prop.
Variables P Q R : Prop.

Check P.
P : Prop
End logique_prop.
```

Les connecteurs logiques sont `/\`, `\/`, `~`, `->`, `<->`.

Exemples de propositions :

```
il_pleut \/ ~ il_pleut
P /\ Q -> Q /\ P
~ (P \/ Q) -> ~(P /\ Q)
```

Égalité

Si `t_1` et `t_2` sont des termes du même type, alors `t_1=t_2` est une proposition.

```
Check add3 2 = 5.
add3 2 = 5  :  Prop

Check true = 3.
                ^
Error: The term "3" has type "nat" while it is expected
       to have type "bool".

Check  true <> 3. (* ∼ true = 3 *)
                ^
Error: The term "3" has type "nat" while it is expected
       to have type "bool".
```

Prédicats

Un prédicat est une fonction dont le type du résultat est `Prop`.

```
Variable est_pair : nat -> Prop.

Definition plus_grand_que_10 (n:nat) : Prop := n > 10.


Check plus_grand_que_10 (10 * 2).
plus_grand_que_10 (10 * 2) : Prop
```

Un prédicat peut avoir plusieurs arguments. Par exemple, une relation est un prédicat à 2 arguments.

```
Variable plus_grand_que : nat -> nat -> Prop.
```

Quantificateurs

Soient `P` une proposition et `x` une variable.
$\forall$ `x:A, P` et $\exists$ `x : A, P` sont des propositions.

Notation ASCII : le symbole $\forall$ s'écrit forall, $\exists$ s'écrit exists.

```
Variable A : Type.
Variable R : A -> A -> Prop.
Variable f : A -> A.
Variable a : A.

Check f (f a).
(f (f a)) : A
Check R a (f (f a)).
R a (f (f a)): Prop
Check forall x :A, R a x -> R a (f (f (f x))).
forall x :A, R a x -> R a (f (f (f x))) : Prop.
```

Logique d'ordre supérieur

Il est possible de quantifier sur des types, des fonctions ou des prédicats.

```
Lemma or_comm : forall P Q:Prop, P \/ Q -> Q \/ P.

Lemma not_ex_all_not : forall (A:Type)(P:A->Prop),
  (~ exists a:A, P a) -> forall a, ~ P a.
```

Le langage de spécification de Coq

1. Premiers pas en Coq
2. Un peu de logique
3. Types inductifs et fonctions récursives
4. Prédicats inductifs
5. Prédicats coinductifs

Types énumérés

```
Inductive bool := true | false.

Inductive color : Type :=
| white | black
| red | blue | green | yellow | cyan | magenta .

Check cyan.
cyan : color
```

Filtrage

Analyse par cas

```
Definition negb b :=
 match b with
   | true => false
   | false => true
 end.
```

Le plus souvent : une clause de filtrage par constructeur

Pour `bool`, une autre écriture possible est `if b then false else true`.

```
Compute negb true.
= false : bool

Compute negb (negb true).
= true : bool
```

Filtrage (suite)

```
Definition is_bw (c : color) : bool :=
  match c with
  |  black => true
  |  white => true
  | _ => false
  end.

Compute in (is_bw red).
= false : bool
```

Un exemple de type polymorphe : le type « option »

```
Print option.

Inductive option (A : Type) : Type :=
  Some : A -> option A
| None : option A

Check Some 3.
Some 3 : option nat

Check Some blue.
Some blue : option color
```

Types récursifs

Le type `nat` des entiers naturels

```
Inductive nat :=
| O : nat                    (* zéro *)
| S : nat -> nat.            (* successeur *)

Check S (S (S O)).
3 : nat
```

`nat` est un type inductif ayant 2 constructeurs `O` et `S`.

- Tout entier naturel est soit `O`, soit de la forme `S p`, `p` étant un entier naturel.
- Les expressions de filtrage distinguent ces deux cas.

```
Definition pred (n : nat) := match n with O => O | S p => p end.
```

Fonctions récursives

`Fixpoint` à la place de `Definition` : autorise certains appels récursifs

```
Fixpoint div2 (n : nat) :=
  match n with
  | S (S p) => S (div2 p)
  | _ => 0
  end.
```

`p` est un sous-terme de l'argument inductif `n` (récursion structurelle).
Par exemple, si `n` vaut `S (S (S 0))`, alors `p` vaut `S 0`, qui est un sous-terme du précédent.
Ainsi, la terminaison des calculs est garantie.

D'autres fonctions récursives sur les entiers

```
Fixpoint plus n m :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.

Notation : n + m pour plus n m

Fixpoint minus n m := match n, m with
  | S n', S m' => minus n' m'
  | _, _ => n
 end.
Notation : n - m pour minus n m
```

Extraction

```
Fixpoint plus n m :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.
```

```
Extraction plus.
```

```
(** val plus : nat -> nat -> nat **)
let rec plus n m =
  match n with
  | O -> m
  | S n' -> S (plus n' m)
```

Extraction

```
Fixpoint plus n m :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.

Recursive Extraction plus.
type nat = | O | S of nat

(** val plus : nat -> nat -> nat **)
let rec plus n m =
  match n with
  | O -> m
  | S n' -> S (plus n' m)
```

Vers une récursion plus générale

Obliger à écrire une récursion structurelle peut être trop contraignant.

Parfois, dans une fonction récursive non structurelle, il est facile de voir qu'une certaine quantité décroît.

Solution générale : récursion bien fondée

Solution intermédiaire : utiliser la commande `Function`

- Permet d'écrire des fonctions non récursives structurelles
- Oblige à garantir la terminaison de la fonction par une preuve séparée

Types mutuellement récursifs

Il peut être nécessaire de définir simultanément plusieurs types inductifs, quand ils dépendent mutuellement l'un de l'autre.

```
Inductive t1 ... :=
  ...
with t2 ... :=
  ...
```

Le type des listes

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

Peut aussi s'écrire comme suit :

```
Inductive list (A : Type) : Type :=
| nil
| cons (a : A) (l : list A).

Check nil.
nil : forall A : Type, list A

Check cons color blue (nil color).
cons color blue (nil color) : list color
```

```
Inductive list (A : Type) :=
 | nil : list A
 | cons : A -> list A -> list A.

Implicit Arguments nil [A].
Implicit Arguments cons [A].

Notation "a :: l" := (cons a l).

Check blue :: white :: red :: yellow :: nil.
blue :: white :: red :: yellow :: nil  : list color
```

Le type des listes est défini dans la bibliothèque `List`
(`Require Import List.`)

Types récursifs : bilan

- Un type récursif `T` est décrit par différents constructeurs.
- Chaque constructeur est une fonction.
  - Type du résultat : le type `T`
  - Entrées : différents champs
  - Certains champs peuvent être le type `T` (récursion).
  - En général, l'un des constructeurs n'a pas d'entrée.
- Fonctions opérant sur `T`
  - Utiliser `match ... with ... end`
  - Autant de clauses de filtrage que de constructeurs

Paires & co

Un autre exemple de type polymorphe

```
Print pair.
Inductive prod (A B : Type) : Type :=
    pair : A -> B -> A * B
```

La notation `A * B` représente `(prod A B)`.
La notation `(x, y)` représente `(pair x y)` (argument implicite).

```
Check (2, 4).    : nat * nat
Check (true, 2 :: nil).    : bool * (list nat)
```

Accès aux composantes

```
Compute (fst (0, true)).
 = 0 : nat
Compute (snd (0, true)).
 = true : bool
```

Paires & co

Les paires peuvent être imbriquées.

```
Check (0, 1, true).
  : nat * nat * bool
Compute (fst (0, 1, true)).
 = (0, 1)
                    : nat * nat
```

Généralisation à des n-uplets polymorphes

```
Inductive triple (T1 T2 T3 : Type) :=
  Triple: T1 -> T2 -> T3 -> triple T1 T2 T3.
```

Type enregistrement

```
Record admin_person := MkAdmin {
  id_number : nat;
  date_of_birth : nat * nat * nat;
  place_of_birth : nat;
  gender : bool
}.

Definition MrX := MkAdmin 42 (1,1,2001) 6 true.
```

Un enregistrement est un type inductif à un seul constructeur !

Type enregistrement

Un champ peut être une propriété logique.

```
Record admin_person := MkAdmin {
  id_number : nat;
  range : 0 <= id_number <= 256;
  date_of_birth : nat * nat * nat;
  place_of_birth : nat;
  gender : bool
}.
```

Accès aux champs :

```
Variable t : admin_person.
Check t.(id_number).
  id_number t : nat
Check id_number.
  id_number : admin_person -> nat
Check t.(range).
  range t : 0 <= id_number t <= 256
```

Le langage de spécification de Coq

1. Premiers pas en Coq
2. Un peu de logique
3. Types inductifs et fonctions récursives
4. Prédicats inductifs
5. Prédicats coinductifs

Prédicats inductifs

```
Inductive est_pair : nat -> Prop :=
|  est_pair0 : est_pair 0
|  est_pairS : forall p:nat,
           est_pair p ->
           est_pair (S (S p)).
```

Chaque constructeur est un théorème permettant de conclure `est_pair` :

```
Check est_pair0.
  est_pair0 : est_pair 0
Check est_pairS.
  est_pairS : forall p : nat, est_pair p -> est_pair (S (S p))
```

Il n'y a pas d'autre moyen de conclure `est_pair` que d'appliquer un de ces deux constructeurs. Cela nous permet de prouver p.ex.

```
Lemma un_non_pair : ~ est_pair (S 0).
```

Exemple de prédicat inductif opérant sur des listes

```
Variable A : Type.

Inductive is_tail : list A -> list A -> Prop :=
  | is_tail_refl:
      forall c, is_tail c c
  | is_tail_cons:
      forall i c1 c2, is_tail c1 c2 -> is_tail c1 (i :: c2).

Lemma is_tail_cons_left:
  forall (i: A) c1 c2, is_tail (i :: c1) c2 -> is_tail c1 c2.
```

Un autre exemple de prédicat inductif

La relation $\leq$ sur les entiers naturels est définie par un prédicat inductif.

```
Inductive le (n : nat) : nat -> Prop :=
  | le_n : le n n
  | le_S : forall m : nat, le n m -> le n (S m).
```

La proposition `(le n m)` s'écrit `n <= m`. `n` est un paramètre de `le`.

Comment démontrer la propriété suivante ?

```
Lemma le_SSS : forall n, le n (S (S (S n))).
```

Une autre propriété

```
Lemma le_trans : forall m n p, le m n -> le n p -> le m p.
```

Comment représenter $\leq$?

Le terme `(le n m)` est représenté par `n <= m`.

```
Print le.

Inductive le (n : nat) : nat -> Prop :=
   | le_n : n <= n
   | le_S : forall m : nat, n <= m -> n <= (S m)

Fixpoint leb n m : bool :=
   match n, m with
   | 0, _ => true
   | S i, S j => leb i j
   | _, _ => false
end.
```

le ou leb ?

```
Compute leb 5 45.
 = true : bool

Lemma L5_45 : 5 <= 45.
```

Correnspondance entre les deux définitions

```
Lemma le_leb_iff : forall n p, n <= p <-> leb n p = true.
```

Définitions inductives et fonctions

Il peut être particulièrement difficile de représenter une fonction `f : A -> B` par une fonction Coq, par exemple du fait de la contrainte de terminaison.

L'étude de sémantiques de langages de programmation est un domaine dans lequel cette situation est fréquente.

Dans ce cas, décrire les fonctions par des relations inductives s'avère très efficace.

Que définissent les prédicats suivants ?

```
Inductive mem : nat -> list nat -> Prop :=
  | mem_head: forall hd tl, mem hd (hd :: tl)
  | mem_tail: forall x hd tl, mem x tl -> mem x (hd :: tl).

Inductive sorted : list nat -> Prop :=
  | sorted_nil:
      sorted nil
  | stored_cons: forall hd tl,
      sorted tl ->
      (forall x, mem x tl -> hd <= x) ->
      sorted (hd :: tl).
```

Que définissent les prédicats suivants ?

```
Variable A : Type.
Variable R : A -> A -> Prop.

Inductive star : A -> A -> Prop :=
 | st1: forall a, star a a
 | st2: forall a1 a2 a3, R a1 a2 -> star a2 a3 -> star a1 a3.

Lemma st3: forall a1 a2, R a1 a2 -> star a1 a2.
Lemma st4: forall a1 a2 a3, star a1 a2 -> star a2 a3 -> star a1 a3.
Lemma st5: forall a1 a2 a3, star a1 a2 -> R a2 a3 -> star a1 a3.

Inductive plus : A -> A -> Prop :=
 | plus1: forall a1 a2 a3, R a1 a2 -> star a2 a3 -> plus a1 a3.

Lemma plus2: forall a1 a2, plus a1 a2 ->
    R a1 a2 \/ exists a', R a1 a' /\ plus a' a2.
Lemma plus3: forall a1 a2, star a1 a2 -> a2 = a1 \/ plus a1 a2.
```

Application

Que représentent `(star _ est_successeur_de)` et `(plus _ est_successeur_de)` ?

```
Check star _ est_successeur_de.
  star nat est_successeur_de : nat -> nat -> Prop

Check plus _ est_successeur_de.
  plus nat est_successeur_de : nat -> nat -> Prop
```

Le langage de spécification de Coq

1. Premiers pas en Coq
2. Un peu de logique
3. Types inductifs et fonctions récursives
4. Prédicats inductifs
5. Prédicats coinductifs

Types et prédicats coinductifs

Comment représenter en Coq des structures de données infinies ou potentiellement infinies ?

Coq fournit des moyens de simuler la construction de structures de données infinies, sur lesquelles il est possible de raisonner et d'effectuer des calculs.

Exemples : construire en Coq la séquence infinie des nombres premiers, ou la trace d'exécution d'un processus revient à créer un terme fini dans lequel il est possible d'accéder séquentiellement à n'importe quel élément de la séquence.

Type coinductif

```
CoInductive LList (A: Type) : Type :=
|  LNil : LList A
|  LCons : A -> LList A -> LList A.

Implicit Arguments LNil [A].
```

Les constantes LNil and LCons sont les deux constructeurs du type LList A.

Exemple 1 : Construction d'une liste finie

```
Check (LCons 1 (LCons 2 (LCons 3 LNil))).
LCons 1 (LCons 2 (LCons 3 LNil))
     : LList nat
```

Type coinductif

Comment construire une liste infinie ?

Exemple 2 : la liste des entiers naturels à partir de n

```
Fixpoint from (n:nat) : LList nat
      := LCons n (from (S n)).
```

```
Error: Recursive definition of from is ill-formed.
```

Définition de fonctions à l'aide de CoFixpoint

La commande CoFixpoint permet de construire des objets infinis.

```
CoFixpoint from (n:nat) : LList nat :=
  LCons n (from (S n)).

Definition nat_stream := from 0.

CoFixpoint repeat (A:Type)(a: A) :=  LCons a (repeat a).
```

Fonctions sur listes potentiellement infinies

```
Definition LHead (A:Type) (l:LList A) : option A :=
  match l with
  | LNil => None
  | LCons a l' => Some a
  end.

Definition LTail (A:Type) (l:LList A) : LList A :=
  match l with
  | LNil => LNil
  | LCons a l' => l'
  end.
```

Prédicats inductifs sur un type co-inductif

```
Inductive Finite (A:Type): LList A -> Prop :=
 Finite_LNil : Finite LNil
|Finite_Lcons : forall a l,
                Finite l ->
                Finite (LCons a l).
```

Prédicats co-inductifs

Un prédicat coinductif est :

- un type coinductif,
- un prédicat (fonction dont le type du résultat est `Prop`).

```
CoInductive Infinite(A:Type): LList A -> Prop :=
 Infinite_LCons : forall a l, Infinite l ->
                              Infinite (LCons a l).
```

Ce prédicat a un seul constructeur. Contraitement à la plupart des prédicats inductifs, il n'a pas de cas de base (i.e. constructeur non récursif).

Deuxième partie II

Le langage IMP et ses sémantiques

Le langage IMP et ses sémantiques

6. Le langage IMP
7. Sémantique naturelle
8. Sémantique à transitions et continuations
9. Sémantique à réductions
10. Interprète de référence

Le langage IMP

Un petit langage impératif avec contrôle structuré

Expressions arithmétiques entières :

$a ::= n$	constantes 0, 1, ...
$\mid x$	variables
$\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$	les 4 opérations

Expressions booléennes (conditions) :

$b ::= \texttt{true} \mid \texttt{false}$	constantes booléennes
$\mid a_1 = a_2 \mid a_1 \leq a_2$	comparaisons entières
$\mid \texttt{!}\ b_1$	négation
$\mid b_1\ \texttt{\&}\ b_2$	conjonction

Commandes (*statements*) :

$c ::= \texttt{skip}$	ne rien faire
$\mid x := a$	affectation
$\mid c_1 ; c_2$	séquence
$\mid \texttt{if}(b)\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \}$	conditionnelle
$\mid \texttt{while}(b)\ \{\ c_1\ \}$	boucle

Un programme IMP

Division et reste par l'algorithme d'Euclide :

```
r := x;
q := 0;
while (y <= r) {
    r := r - y;
    q := q + 1
}
```

(En sortie, `q` est le quotient `x/y` et `r` le reste `x` mod `y`.)

Syntaxe concrète, syntaxe abstraite Coq

Syntaxe concrète
(grammaire BNF) :

$$
\begin{aligned}
a ::= {} & n \\
\mid {} & x \\
\mid {} & a_1 + a_2 \\
\mid {} & a_1 - a_2 \\
\mid {} & a_1 * a_2 \\
\mid {} & a_1 / a_2
\end{aligned}
$$

Syntaxe abstraite
(type inductif Coq) :

```
Inductive aexp : Type :=
  | Const: Z -> aexp
  | Var: ident -> aexp
  | Plus: aexp -> aexp -> aexp
  | Minus: aexp -> aexp -> aexp
  | Times: aexp -> aexp -> aexp
  | Div: aexp -> aexp -> aexp.
```

De même pour les expressions booléennes (type `bexp`) et les commandes (type `command`).

Le langage IMP et ses sémantiques

6 Le langage IMP

7 Sémantique naturelle

8 Sémantique à transitions et continuations

9 Sémantique à réductions

10 Interprète de référence

But de la sémantique formelle

Définir mathématiquement la signification de la syntaxe d'un langage de programmation :

- Quelle est la valeur calculée par une expression ?
- Quel est l'effet d'une commande sur les valeurs des variables ?
- Quels sont les comportements indéfinis ? (= erreurs)

La sémantique naturelle (ou : «à grands pas»)

(Gilles Kahn, vers 1985)

Définir des relations entre un bout de syntaxe et ses résultats possibles :

`eval m a n` dans l'état mémoire `m`, l'expression arithmétique `a` s'évalue en l'entier `n`

`beval m be bv` dans l'état mémoire `m`, l'expression booléenne `be` s'évalue en le booléen `bv` (`true` ou `false`)

`exec m c m'` démarrée dans l'état mémoire `m`, la commande `c` termine avec l'état `m'`

États mémoire = fonctions
variable $\mapsto$ `None` (non initialisée) ou `Some`(n) (valeur de la variable).

Plusieurs résultats possibles = non-déterminisme.
Aucun résultat possible = erreur lors de l'exécution (ou : non-terminaison).

La sémantique naturelle (ou : «à grands pas»)

Les relations d'une sémantique naturelle sont définies par récurrence et par cas en suivant la forme de la syntaxe.

Example (Évaluation des expressions de la forme a_1/a_2)

a_1/a_2 s'évalue en l'entier n dans l'état m

ssi

a_1 dans m s'évalue en un entier n_1,

a_2 dans m s'évalue en un entier n_2,

n_2 est différent de 0,

et n est le quotient n_1/n_2

Définition Coq via des prédicats inductifs

(Cf. module `IMP`.)

```
Inductive eval: store -> aexp -> Z -> Prop := ...

Inductive beval: store -> bexp -> bool -> Prop := ...

Inductive exec: store -> command -> store -> Prop := ...
```

À chaque forme possible d'une expression correspond un constructeur des prédicats `eval` ou `beval`.

Pour les commandes, on a 2 constructeurs pour `If` et 2 pour `While`, correspondant aux cas "la condition est vraie" et "la condition est fausse".

Si pour tout n, aucun cas ne permet de conclure `eval` m a n, cela dénote une erreur pendant l'évaluation de a. (Ex : variable non définie, division par zéro.)

Le langage IMP et ses sémantiques

6 Le langage IMP

7 Sémantique naturelle

8 Sémantique à transitions et continuations

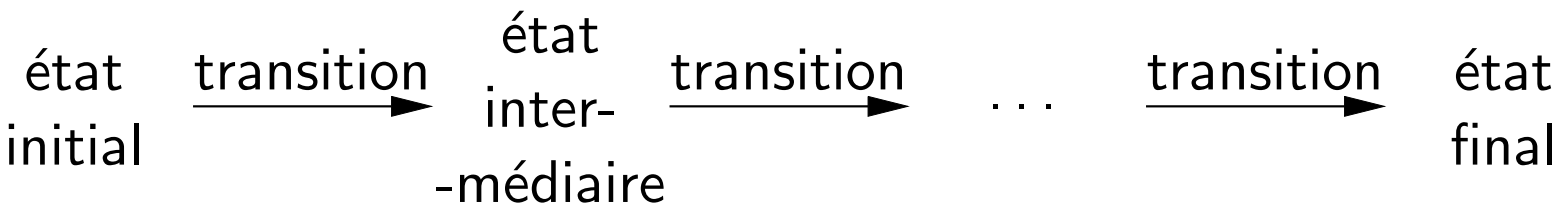
9 Sémantique à réductions

10 Interprète de référence

Un grand pas ou plusieurs petits pas ?

La vision ≪grand pas≫ (sémantique naturelle) :

La vision ≪petits pas≫ (sémantique à transitions) :



Avantage des sémantiques à transitions

En formant des suites de transitions partant de l'état initial, on décrit de manière systématique et uniforme les 3 types d'exécutions :

- Terminaison sans erreurs :

$$\text{état initial} \longrightarrow \cdots \longrightarrow \text{état final}$$

- Divergence (non-terminaison) sans erreurs :

$$\text{état initial} \longrightarrow \cdots \longrightarrow \cdots \qquad \text{(une infinité de transitions)}$$

- Terminaison sur une erreur :

$$\text{état initial} \longrightarrow \cdots \longrightarrow \text{état non final} \not\longrightarrow \qquad \text{(pas de transition)}$$

Vers une sémantique à transitions pour IMP

On choisit de garder une évaluation «grand pas» pour les expressions arithmétiques et booléennes. (Ces évaluations terminent toujours, soit sur un résultat, soit en échouant.)

Par conséquent, une transition correspond à l'exécution d'une commande élémentaire, par exemple :

- Effectuer une affectation $x := a$
- Dans une conditionnelle `if` $(b)\ldots$, calculer la valeur booléenne de b et choisir une des branches du `if`.

Vers une sémantique à transitions pour IMP

En quoi consiste un état de la sémantique ? À vue de nez :

- Un point de programme courant, désignant la prochaine commande élémentaire à exécuter.
- L'état mémoire courant.

Mais comment représenter les points de programmes ?

Points de programmes explicites

Idée naturelle : étiqueter le programme source.

$$l_1 : (l_2 : (x := 1); l_3 : (y := 2) : l_4) : l_5$$

Les transitions successives seraient :

$$\begin{array}{c} (l_1, m) \\ \downarrow \quad \text{focaliser sur la 1ière partie de la séquence} \\ (l_2, m) \\ \downarrow \quad \text{effectuer la 1ière affectation} \\ (l_3, m[x \leftarrow 1]) \\ \downarrow \quad \text{effectuer la 2ème affectation} \\ (l_4, m[x \leftarrow 1, y \leftarrow 2]) \\ \downarrow \quad \text{sortir de la séquence} \\ (l_5, m[x \leftarrow 1, y \leftarrow 2]) \end{array}$$

Problème : difficile à bien formaliser sur machine.

Sous-commandes et continuations

Idée plus commode : considérer des paires d'une sous-commande c et d'un terme de continuation k qui décrit ce qui reste à faire lorsque c a terminé.

Sous-commande	Continuation	État mémoire
$x := 1; y := 2$	«on s'arrête»	m
	$\downarrow$ (focalisation)	
$x := 1$	«on fait $y := 2$ et on s'arrête»	m
	$\downarrow$ (calcul)	
`skip`	«on fait $y := 2$ et on s'arrête»	$m[x \leftarrow 1]$
	$\downarrow$ (reprise)	
$y := 2$	«on s'arrête»	$m[x \leftarrow 1]$
	$\downarrow$ (calcul)	
`skip`	«on s'arrête»	$m[x \leftarrow 1, y \leftarrow 2]$

La sémantique à transitions d'IMP

États = triplets (sous-commande, continuation, état mémoire).

Relation de transition entre états : le prédicat `step`.

États initiaux = (le programme, «on s'arrête», état mémoire initial).

États finaux = (`skip`, «on s'arrête», état mémoire final).

(→ Fichier Coq `IMP`.)

Le langage IMP et ses sémantiques

6 Le langage IMP

7 Sémantique naturelle

8 Sémantique à transitions et continuations

9 Sémantique à réductions

10 Interprète de référence

Sémantiques à réductions

Une autre forme de sémantique ≪à petits pas≫ où on réduit étape par étape le terme syntaxique, en le réécrivant petit à petit.

(Au lieu de se déplacer à l'intérieur du terme d'origine.)

Exemple typique : calculer ≪à la main≫ une expression arithmétique.

$$(1 + 2) \times (3 + 4) \longrightarrow 3 \times (3 + 4) \longrightarrow 3 \times 7 \longrightarrow 21$$

Une sémantique à réductions pour les expressions d'IMP

Des règles de réduction en tête de l'expression, correspondant à une étape élémentaire de calcul :

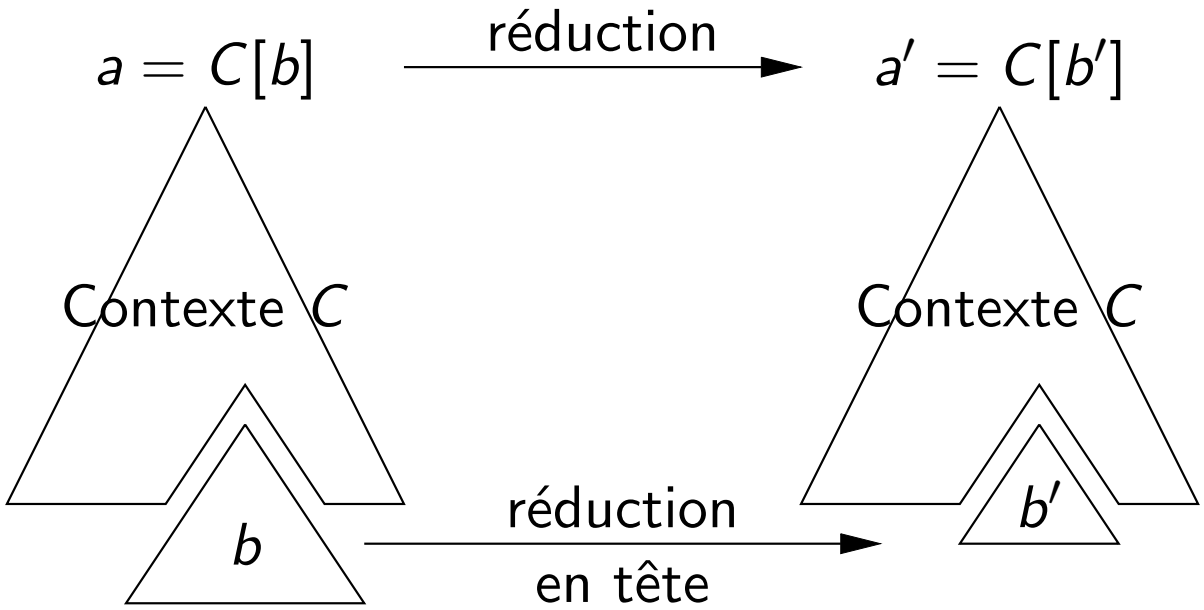
$$\begin{array}{rcll} x & \stackrel{\epsilon}{\rightarrow} & n & \text{si } m(x) = \texttt{Some}(n) \\ n_1 + n_2 & \stackrel{\epsilon}{\rightarrow} & n & \text{avec } n = n_1 + n_2 \\ n_1 - n_2 & \stackrel{\epsilon}{\rightarrow} & n & \text{avec } n = n_1 - n_2 \\ n_1 * n_2 & \stackrel{\epsilon}{\rightarrow} & n & \text{avec } n = n_1.n_2 \\ n_1/n_2 & \stackrel{\epsilon}{\rightarrow} & n & \text{si } n_2 \neq 0, \text{ avec } n = n_1/n_2 \end{array}$$

Plus : une règle générale de réduction sous un contexte :

$$a \rightarrow a' \text{ ssi } a = C[b] \text{ et } b \stackrel{\epsilon}{\rightarrow} b' \text{ et } a' = C[b']$$

Décomposition expression = contexte [redex]

Un contexte = une expression avec un ≪trou≫ noté [ ].



Exemple : $(1 + 2) \times (3 + 4) \rightarrow (1 + 2) \times 7$
avec le contexte $C = (1 + 2) \times [\,]$ et la réduction en tête $3 + 4 \xrightarrow{\epsilon} 7$.

(Cf. fichier Coq `IMP`)

Le langage IMP et ses sémantiques

6 Le langage IMP

7 Sémantique naturelle

8 Sémantique à transitions et continuations

9 Sémantique à réductions

10 Interprète de référence

Exécutabilité

Jusqu'ici, nos sémantiques sont définies par des prédicats logiques :

- est-ce que `Plus (Const 1) (Const 2)` s'évalue en 3 ?
- est-ce que l'état (c, k, m) fait une transition vers (c', k', m') ?

Coq ne fournit pas de moyen commode de calculer le résultat associé à un certain terme :

- en quelle valeur s'évalue `Plus (Const 1) (Const 2)` ?
- quelles sont les transitions possibles à partir de l'état (c, k, m) ?

Ce serait très commode pour pouvoir tester une sémantique sur des exemples de programmes.

Utiliser des fonctions au lieu de prédicats

Solution : réécrire la sémantique sous forme de fonctions Coq, effectivement calculables. (Cf. fichier `IMP.v`.)

- `Inductive eval: store -> aexp -> Z -> Prop := ...`
  devient `Fixpoint eval_f (m: store) (a: aexp) : option Z := ...`
- `Inductive beval: store -> bexp -> bool -> Prop := ...`
  devient `Fixpoint beval_f (m: store) (be: bexp): option bool :=`
- `Inductive step: state -> state -> Prop := ...`
  devient `Definition step_f (st: state) : option state := ...`

Noter l'utilisation de types `option` comme résultats :

- `None` signifie «erreur d'exécution».
- `Some` *r* signifie «pas d'erreur et le résultat est *r*».

Exécuter des programmes depuis Coq

Pour les expressions, on obtient directement leur valeur :

```
Compute (eval_f (update vx 42 initial_store)
                (Plus (Var vx) (Const 2))).
    = Some 44 : option Z

Compute (eval_f initial_store (Div (Const 1) (Const 0))).
     = None : option Z
```

Exécuter des programmes depuis Coq

Pour les commandes, on peut itérer la fonction `step_f`, mais il faut borner a priori le nombre maximal d'itérations (afin de garantir la terminaison de la fonction Coq).

```
Inductive result : Type :=
  | Timeout: result
  | Error: result
  | Termination: store -> result.

Fixpoint steps_f (n: nat) (st: state) : result :=
  match n with
  | O => Timeout
  | S n' =>
      match st with
      | (Skip, Kstop, m) => Termination m
      | _ => match step_f st with None => Error
                                 | Some st' => steps_f n' st' end
      end
  end.
```

En conclusion

Plusieurs styles de sémantique, chacun avec ses forces et ses faiblesses :

- Sémantique naturelle («grands pas») :
  faciles à lire et à écrire ; principes de preuves puissants.
- Sémantique «petits pas» à transitions et continuations :
  traitement unifié de la terminaison et de la divergence ;
  faciles à étendre avec des constructions bizarres comme le `goto`.
- Sémantique «petits pas» à réductions :
  bien adaptées pour décrire le non-déterminisme.
- Interprète de référence : exécutable, se prête au test.

On peut utiliser plusieurs sémantiques pour un même langage à condition de montrer des résultats d'équivalence entre ces sémantiques.
(Cf. annexe A du fichier Coq `IMP`.)

Troisième partie III

Un compilateur vérifié pour le langage IMP

Un compilateur vérifié pour le langage IMP

11 La machine virtuelle IMP

12 Le compilateur

13 Vérifier formellement le compilateur

La machine virtuelle IMP

(Un petit sous-ensemble de la machine virtuelle Java (JVM))

Les composants de la machine :

- Le code C : une liste d'instructions.
- Le compteur de programme pc : un entier donnant la position dans C de la prochaine instruction à exécuter.
- L'état mémoire m : associe des valeurs entières aux noms de variables.
- La pile σ : une liste d'entiers
  (sert à stocker temporairement les résultats intermédiaires).

Le jeu d'instructions

$i ::= \texttt{Iconst}(n)$	empiler n sur la pile
$\mid \texttt{Ivar}(x)$	empiler la valeur de x
$\mid \texttt{Isetvar}(x)$	dépiler une valeur, la mettre dans x
$\mid \texttt{Iadd}$	dépiler 2 valeurs, empiler leur somme
$\mid \texttt{Isub}$	dépiler 2 valeurs, empiler leur différence
$\mid \texttt{Imul}$	dépiler 2 valeurs, empiler leur produit
$\mid \texttt{Idiv}$	dépiler 2 valeurs, empiler leur quotient
$\mid \texttt{Ibranch\_forward}(\delta)$	saut en avant inconditionnel
$\mid \texttt{Ibranch\_backward}(\delta)$	saut en arrière inconditionnel
$\mid \texttt{Ibeq}(\delta)$	dépiler 2 valeurs, sauter si $=$
$\mid \texttt{Ibne}(\delta)$	dépiler 2 valeurs, sauter si $\neq$
$\mid \texttt{Ible}(\delta)$	dépiler 2 valeurs, sauter si $\leq$
$\mid \texttt{Ibgt}(\delta)$	dépiler 2 valeurs, sauter si $>$
$\mid \texttt{Ihalt}$	fin du programme

Les instructions sans branchement incrémentent le *pc* de 1.
Les instructions de branchement l'incrémentent de $1 + \delta$ (en avant) ou $1 - \delta$ (en arrière).

(δ est un offset relatif à l'instruction qui suit le branchement.)

Exemple de code et son exécution

			1		
pile	ϵ	12	12	13	ϵ
mémoire	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
p.c.	0	1	2	3	4
code	`Ivar`(x);	`Iconst`(1);	`Iadd`;	`Isetvar`(x);	`Ibranch_backward`(5)

Sémantique de la machine

En style ≪petits pas≫ : une relation de transition représentant l'exécution d'une instruction.

```
Definition code := list instruction.
Definition stack := list Z.
Definition machine_state := (nat * stack * store)%type.

Inductive transition (C: code):
                  machine_state -> machine_state -> Prop :=
  ...
```

(Cf. fichier `IMPcompiler.v`.)

Exécuter des programmes avec la machine

En itérant la relation de transition :

- État initial : $pc = 0$, état mémoire initial, pile vide.
- État final : pc pointe sur une instruction `halt` ; la pile est vide.

```
Definition mach_terminates (C: code) (final_store: store) :=
  exists pc,
  code_at C pc = Some Ihalt /\
  star (transition C) (0, nil, initial_store) (pc, nil, final_store)

Definition mach_diverges (C: code) :=
  infseq (transition C) (0, nil, initial_store).

Definition mach_runtime_error (C: code) :=
  (* tous les autres cas *)
```

Un compilateur vérifié pour le langage IMP

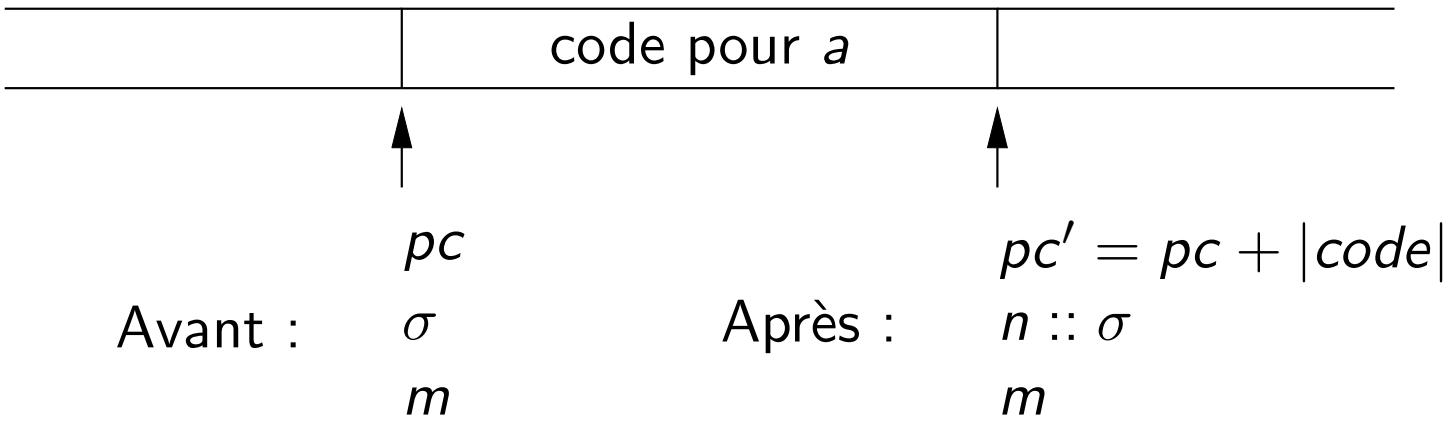
11 La machine virtuelle IMP

12 Le compilateur

13 Vérifier formellement le compilateur

Compilation des expressions arithmétiques

Idée : si a s'évalue en n dans l'état mémoire m,

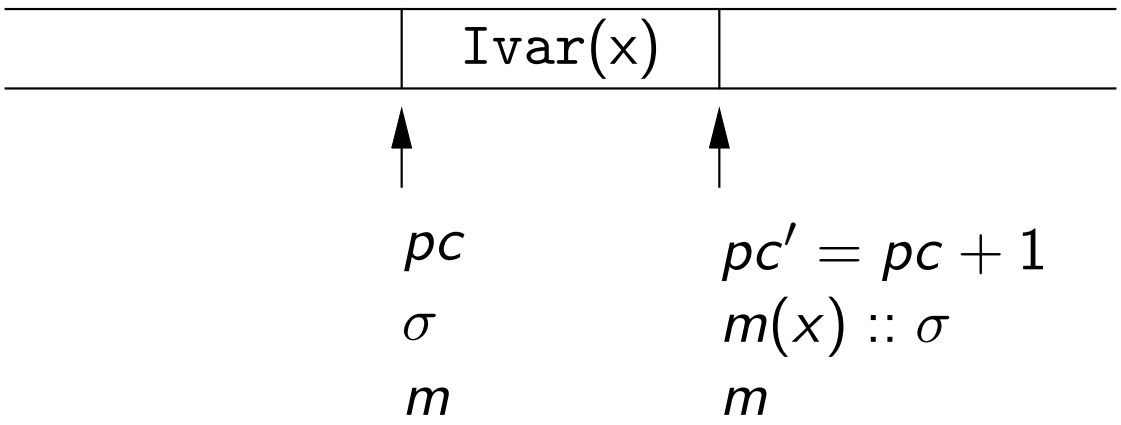


Compilation = traduction en «notation polonaise inverse».

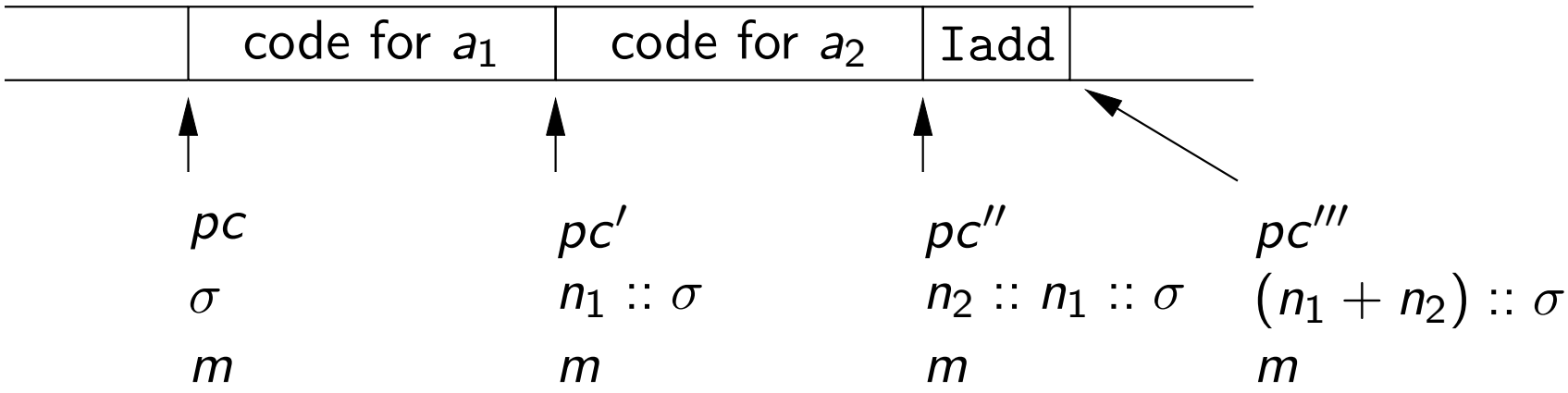
(Voir fonction `compile_aexpr` dans `IMPcompiler.v`)

Compilation des expressions arithmétiques

Un cas de base : $a = x$.



Décomposition récursive : si $a = a_1 + a_2$,

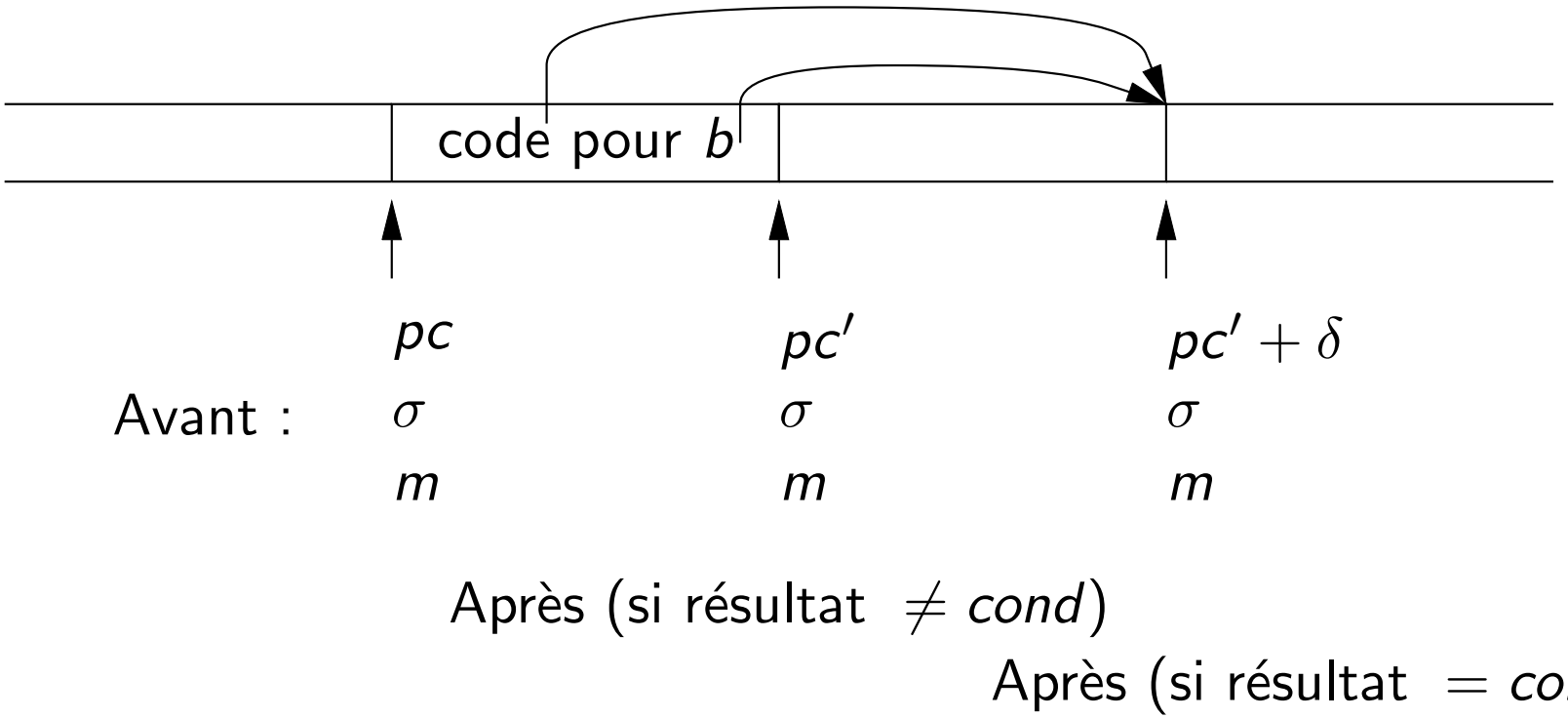


Compilation des expressions booléennes

`compile_bexp` *b cond* δ :

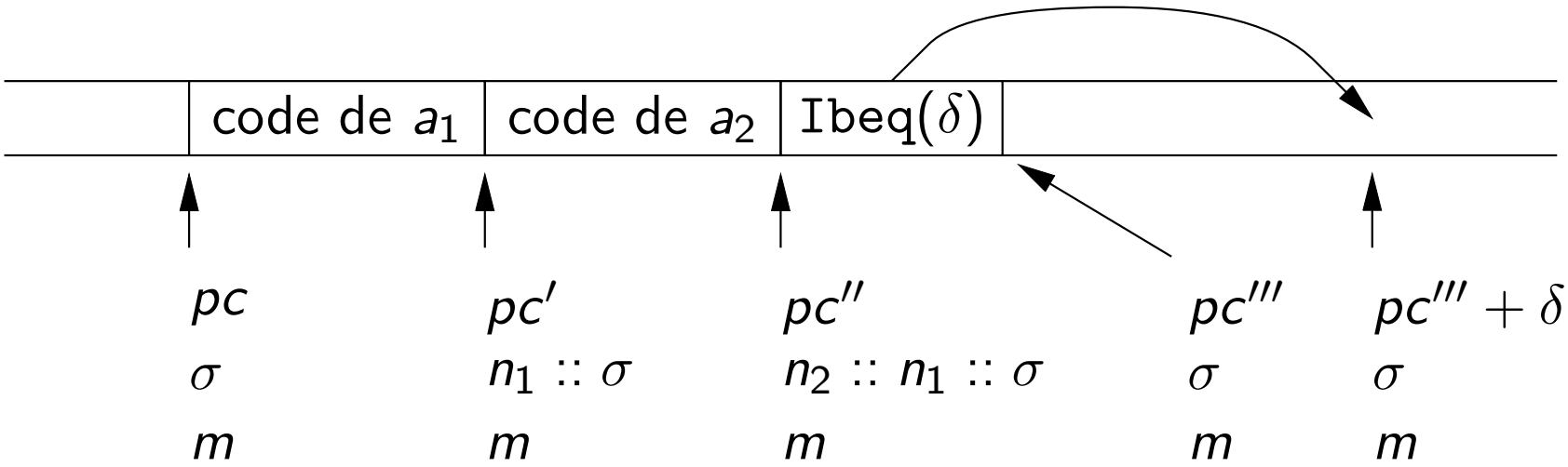
saute δ instructions en avant si *b* s'évalue en la valeur *cond*

continue en séquence si *b* s'évalue en la valeur $\neg cond$.



Compilation des expressions booléennes

Un cas de base : $b = (a_1 = a_2)$ et $cond = \texttt{true}$:



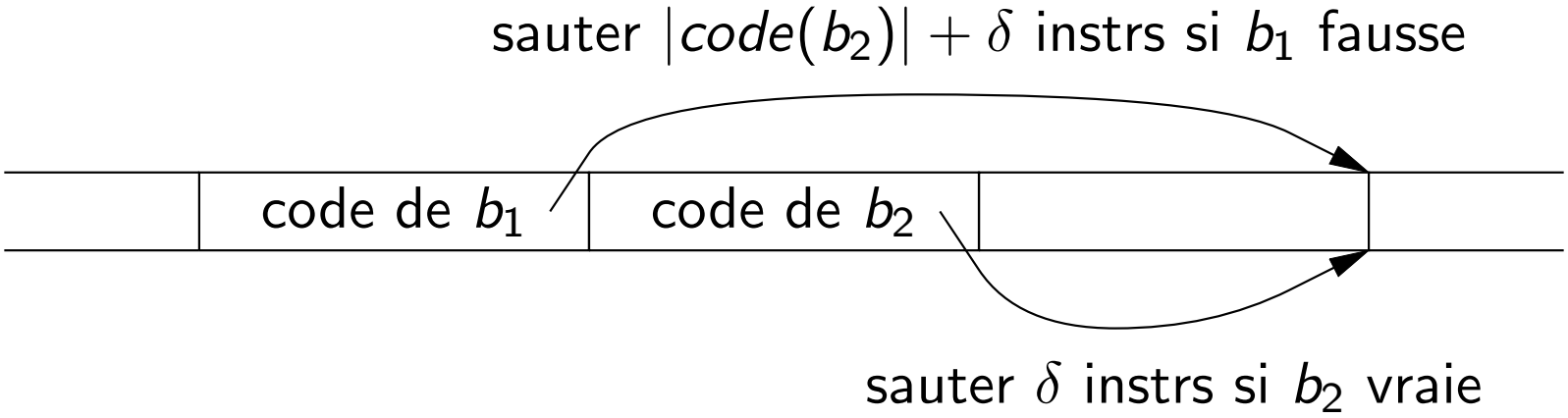
Court-circuiter les conjonctions «&»

Si b_1 s'évalue à `false`, alors b_1 & b_2 s'évalue à `false` ou part en erreur. Dans les deux cas, il est inutile d'évaluer b_2 !

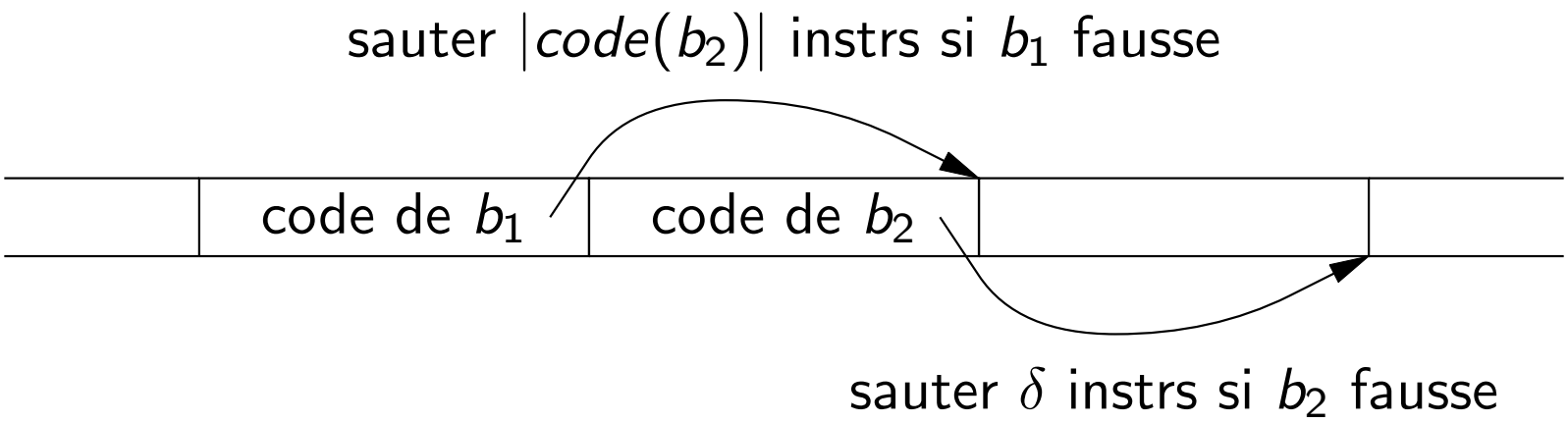
$\rightarrow$ Dans ce cas, le code produit pour b_1 & b_2 devrait sauter par-dessus le code pour b_2 et brancher immédiatement vers la bonne destination.

Court-circuiter les conjonctions «&»

Si *cond* = `false` (brancher si b_1 & b_2 est faux) :

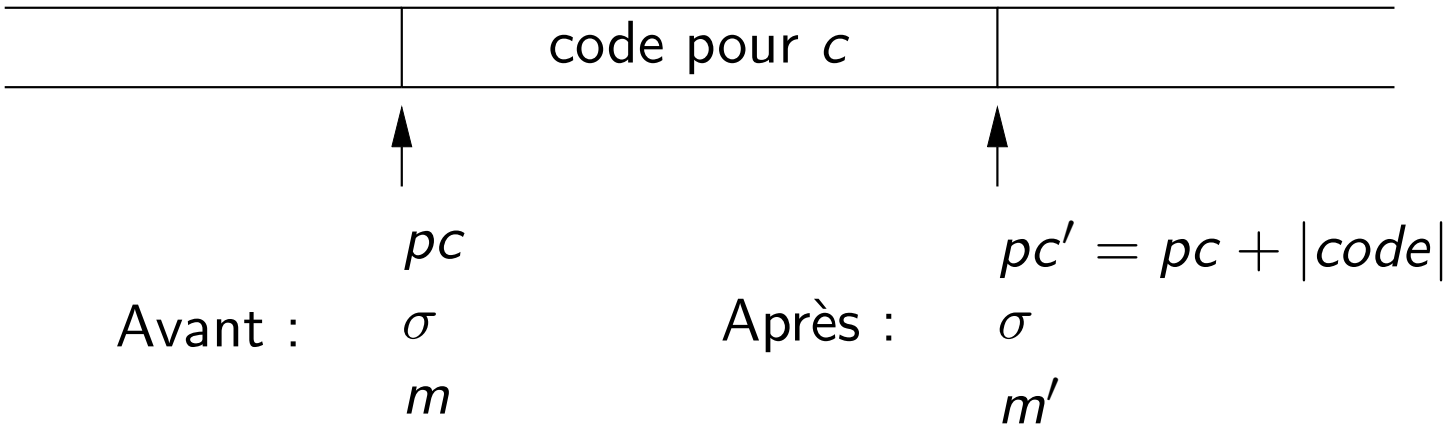


Si *cond* = `true` (brancher si b_1 & b_2 is true) :



Compilation des commandes

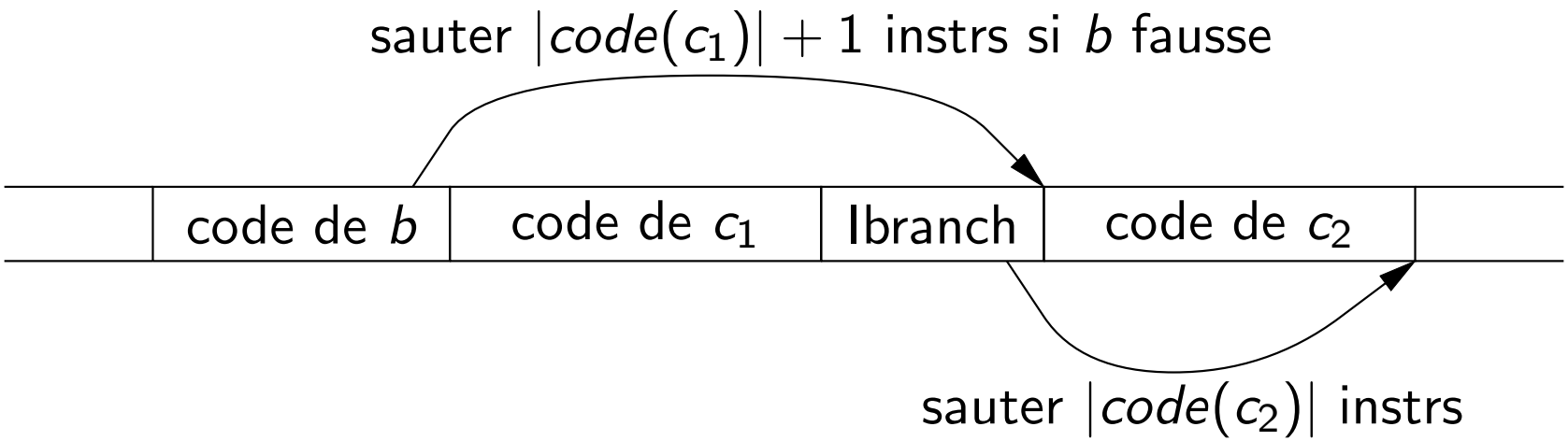
Si la commande c, démarrée dans l'état mémoire m, termine dans l'état mémoire m',



(Voir la fonction `compile_com` dans `IMPCompiler.v`)

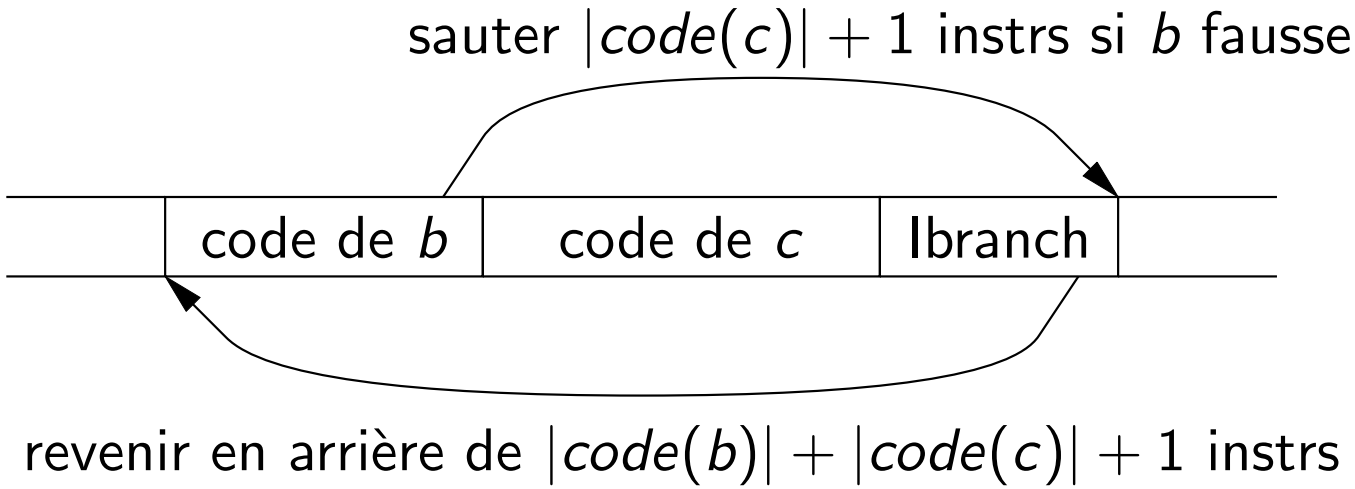
Explication des mystérieux offsets

Code pour `if`(b) { c_1} `else` { c_2 } :



Explication des mystérieux offsets

Code pour `while`(b) $\{$ c $\}$:



Un compilateur vérifié pour le langage IMP

11 La machine virtuelle IMP

12 Le compilateur

13 Vérifier formellement le compilateur

Le compilateur est-il correct ?

Soit c un programme IMP.

Implémentation compilée : faire exécuter le code produit par `compile_program` c par la machine.
$\rightarrow$ comportements terminaison / divergence / erreur.

Spécification : prédire le comportement de c à l'aide d'une des sémantiques formelles d'IMP.
$\rightarrow$ comportements terminaison / divergence / erreur.

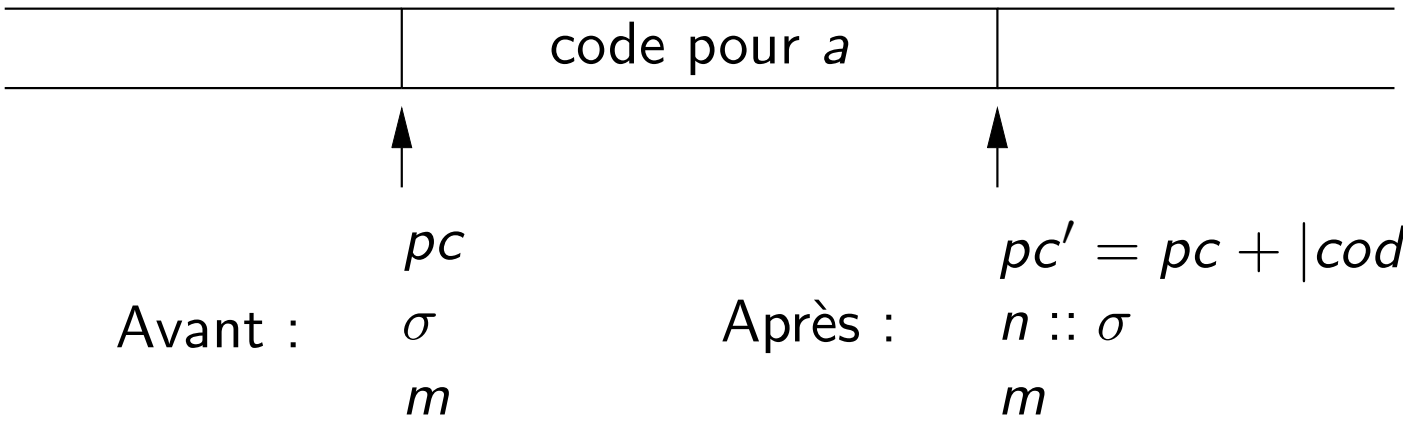
Obtenons-nous les mêmes comportements ?

Est-ce que le code produit par le compilateur se comporte comme prédit par la sémantique du programme source ?

Une première preuve

Essayons de formaliser et de prouver les intuitions que nous avions lorsque nous avons écrit les fonctions de compilation.

Intuition pour les expressions arithmétiques : si a s'évalue en n dans l'état mémoire m,



Un énoncé formel plausible :

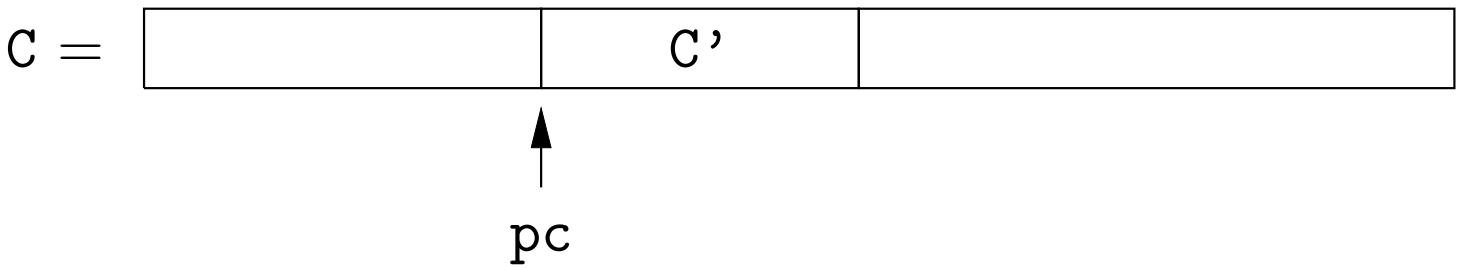
```
Lemma compile_aexp_correct:
  forall m a n pc stk,
  eval m a n ->
  star (transition (compile_aexp a))
       (0, stk, m) (length (compile_aexp a), n :: stk, m).
```

Vérifier la compilation des expressions

Afin de raisonner par récurrence sur l'évaluation de a, il faut généraliser cet énoncé pour que :

- le PC de départ n'est pas forcément 0 ;
- le code `compile_aexp` a apparaît comme un fragment d'un plus grand code C.

Pour ce faire, on définit le prédicat `codeseq_at C pc C'` qui capture la situation suivante :



Vérifier la compilation des expressions

```
Lemma compile_aexp_correct:
  forall C m a n, eval m a n ->
  forall pc stk,
  codeseq_at C pc (compile_aexp a) ->
  star (transition C)
       (pc, stk, m)
       (pc + length (compile_aexp a), n :: stk, m).
```

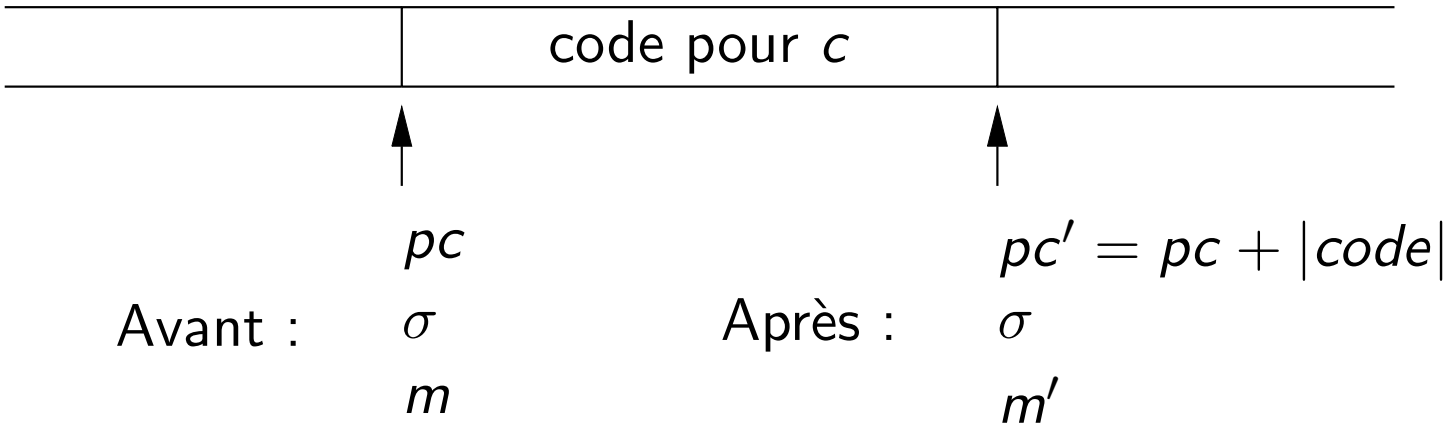
La preuve est une élégante récurrence sur la dérivation de `eval m a n`.

Importance historique :

- Première preuve de compilateur publiée.
  (McCarthy et Painter, 1967).
- Première preuve de compilateur faite ≪sur machine≫
  (Milner et Weyrauch, 1972, utilisant le prouveur Stanford LCF).

Vérifier la compilation des commandes

Intuition : si la commande c, démarrée dans l'état mémoire m, termine dans l'état mémoire m',



Formalisation :

```
Lemma compile_com_correct_terminating:
  forall C m c m', exec m c m' ->
  forall stk pc, codeseq_at C pc (compile_com c) ->
  star (transition C)
       (pc, stk, m)
       (pc + length (compile_com c), stk, m').
```

Point d'étape

En corollaire, on obtient un joli théorème :

```
Theorem compile_program_correct_terminating:
  forall c final_store,
  exec initial_store c final_store ->
  mach_terminates (compile_program c) final_store.
```

Est-ce que cela suffit pour conclure que notre compilateur est correct ?

Pas d'autres comportements indésirables

On a montré que si le programme source IMP termine sans erreurs, un des comportements possibles de la machine, lorsqu'elle exécute le code compilé, est de terminer sans erreur sur le même état mémoire final.

Il nous reste à montrer que la machine ne peut rien faire d'autre, comme par exemple diverger ou bloquer sur une erreur.

C'est une conséquence du fait que la machine IMP est déterministe.

```
Theorem mach_terminates_unique_behavior:
  forall C final_store,
  mach_terminates C final_store ->
  ~mach_diverges C /\
  ~mach_runtime_error C /\
  forall m, mach_terminates C m -> m = final_store.
```

Et si le programme source ne termine pas ?

Si le programme source IMP diverge, on s'attend à ce que la machine, lorsqu'elle exécute le code compilé, fasse une infinité de transitions sans rencontrer d'erreurs.

Ce résultat est vrai, mais demande une preuve complètement différente :

- utilisant la sémantique à transitions et continuations d'IMP au lieu de la sémantique naturelle ;
- raisonnant par simulation : chaque transition IMP de l'exécution du programme source est mise en correspondance avec zéro, une ou plusieurs transitions de la machine exécutant le code compilé.

(Voir fichier `IMPcompiler.v`.)

Quatrième partie IV

L'assembleur PowerPC en CompCert

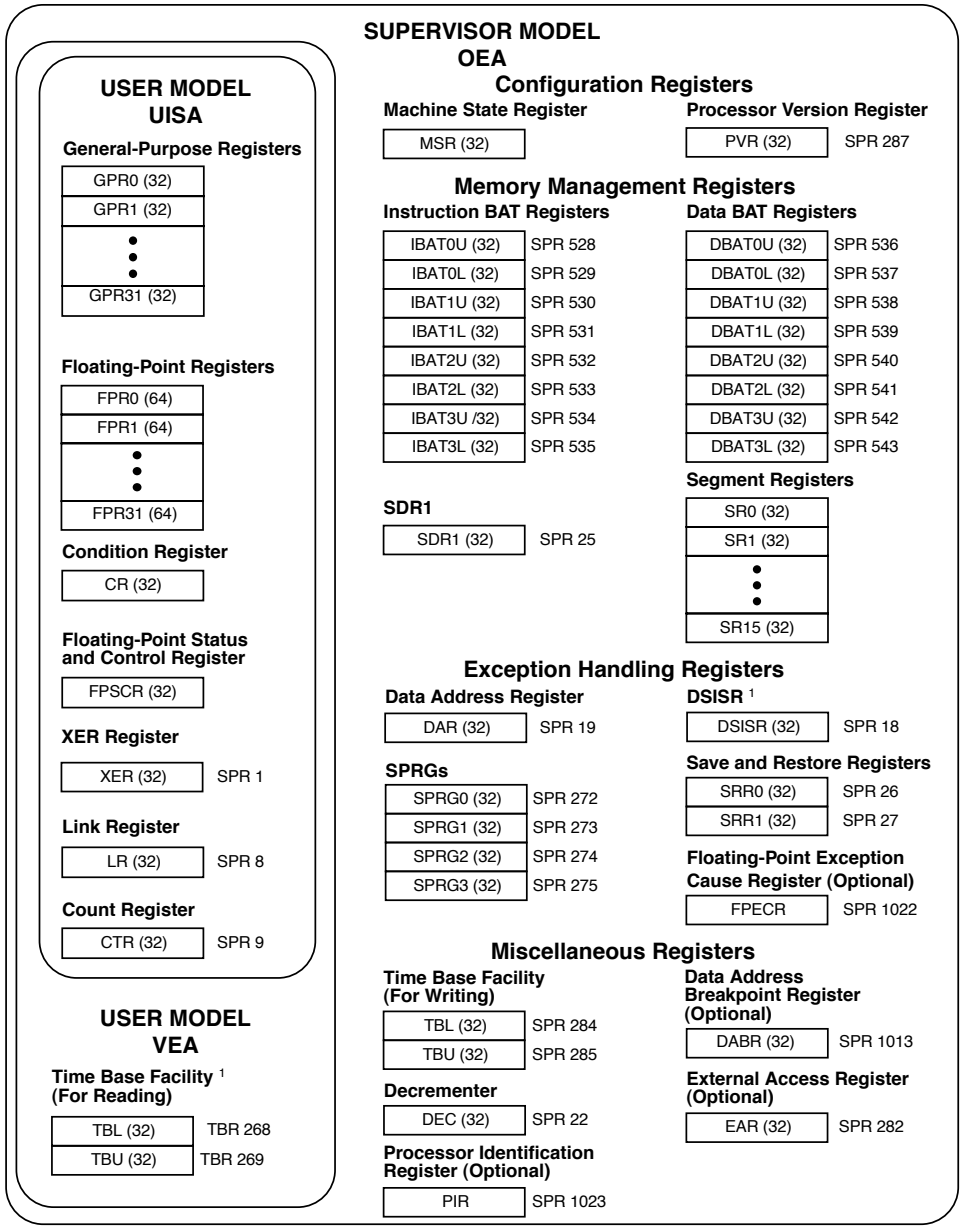
L'assembleur PowerPC en CompCert

14 Modélisation d'un sous-ensemble du PowerPC

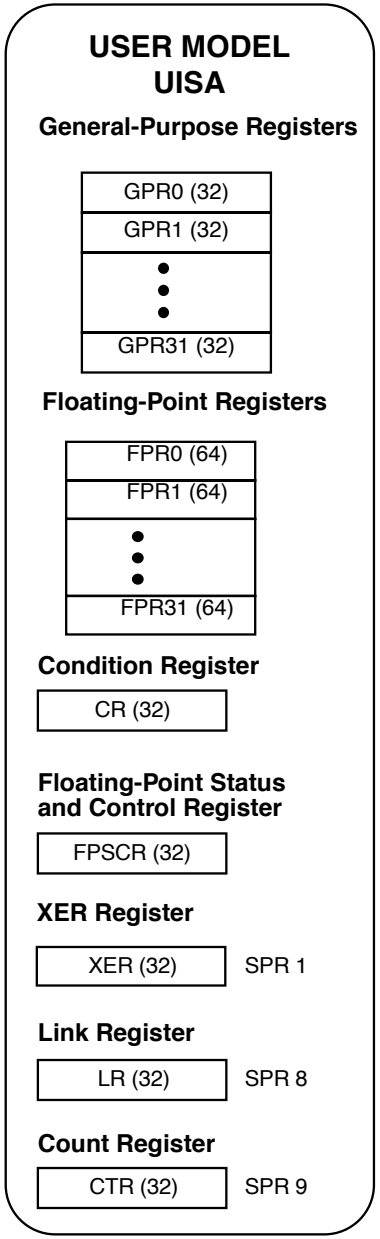
15 Sémantiques : des transitions aux traces

16 Bibliothèques communes à tous les langages de CompCert

Les registres du PowerPC 32 bits



Les registres utilisateurs du PowerPC 32 bits



Registres modélisés dans CompCert :

- Les 32 registres entiers `GPR`
- Les 32 registres flottants `FPR`
- Les registres de branchement `LR` et `CTR`
- Les 4 bits de condition `CR0`...`CR3`
- Le bit `CARRY` du registre XER
- Le compteur de programme `PC`

(Ce sont les seuls registres utilisés par le code compilé.)

États d'exécution du PowerPC

(Module `Asm`)

Partie variable :

1. Valeurs des registres $\mathtt{rs}$ = registre $\mapsto$ valeur
2. État mémoire $\mathtt{m} \approx$ taille&type $\times$ pointeur $\mapsto$ valeur

Partie fixe : un environnement global ge donnant :

1. Symboles globaux $\approx$
   nom de fonction ou de variable globale $\mapsto$ pointeur
2. Contenu du code $\approx$ pointeur de code $\mapsto$ instruction
   (Pas de code auto-modifiant)

Forme générale de la sémantique : une relation de transition

$$\mathrm{ge} \vdash (\mathtt{rs}, \mathtt{m}) \xrightarrow{t} (\mathtt{rs}', \mathtt{m}') \qquad (t = \text{trace d'événements observables})$$

Les valeurs

(Module `Values`)

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

L'union discriminée de

- un entier machine 32 bits
- un flottant IEEE 64 bits
- un pointeur = un bloc mémoire et un offset en octets dans ce bloc
- la valeur `Vundef` dénotant une erreur non fatale.

Que signifie la valeur `Vundef` ?

C'est le résultat d'une opération machine non définie dans notre modèle mais qui ne peut pas ≪planter≫ le processeur. Par exemple :

- Multiplier deux pointeurs.
- Lire dans une zone mémoire non initialisée.

La plupart des opérations propagent `Vundef` sans bloquer.
(Exemple : `Val.add Vundef (Vint 1) = Vundef`.)

D'autres opérations bloquent la sémantique. Exemples :

- Écrire en mémoire à l'adresse `Vundef`
- Branchement conditionnel selon que `Vundef` $= 0$.

Invariants sur les valeurs des registres

Même si le type `val` mélange entiers, flottants et pointeurs, la sémantique du PowerPC garantit des propriétés sur les valeurs des registres :

Registres	Valeurs possibles
`PC`	pointeur de code
`GPR`*i*, `LR`, `CTR`	entier ou pointeur ou `Vundef`
`FPR`*i*	flottant ou `Vundef`
`CR`*i*, `CARRY`	entier 0 ou entier 1 ou `Vundef`

Le jeu d'instructions PowerPC

Sur les $\approx$ 180 instructions du PowerPC 32 bits, on en modélise 90, plus 6 macro-instructions (= séquences standard de vraies instructions).

On modélise au niveau assembleur, pas au niveau langage machine :

- Représentation ≪textuelle≫ des instructions ; pas de codage en mots de 32 bits.
  (P.ex. `Paddi GPR2 GPR2 (Vint 1)` au lieu de `0x38420001`.)
- Utilisation d'étiquettes symboliques pour les branchements au lieu d'offsets relatifs.
  (P.ex. ≪brancher vers `L110`≫ au lieu de ≪brancher +24≫.)

Exemple : l'instruction add

Syntaxe abstraite :

```
Inductive instruction : Type :=
  | Padd: ireg -> ireg -> ireg -> instruction
                                    (**r integer addition *)
```

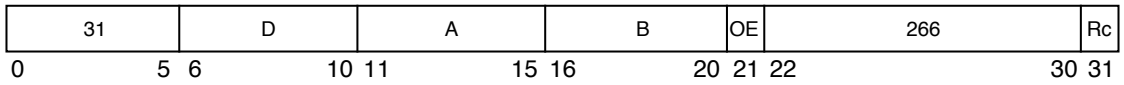
Exemple : l'instruction add

On consulte le manuel de référence du PowerPC :

add*x* **add*x***

Add (x'7C00 0214')

add	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
add.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
addo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
addo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)



rD ← (**r**A) + (**r**B)

The sum (**r**A) + (**r**B) is placed into **r**D.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (If Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see next bullet item.

- XER:

  Affected: SO, OV (If OE = 1)

  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register."

Exemple : l'instruction add

Sémantique de l'instruction :

```
Inductive outcome: Type :=
  | OK: regset -> mem -> outcome
  | Error: outcome.

Definition exec_instr (c: code) (i: instruction)
                      (rs: regset) (m: mem) : outcome :=
  match i with
  | Padd rd r1 r2 =>
      OK (nextinstr (rs#rd <- (Val.add rs#r1 rs#r2))) m
  | ...
```

`Val.add` : traite les cas int+int, int+ptr, ptr+int ; sinon, `Vundef`.

`rs#rd <- ...` : mise à jour de la valeur du registre `rd`.

`nextinstr` : incrémente le PC de 1.

Travaux pratiques

Examinons ensemble les instructions suivantes :

- `cmpw`
- `bt`
- `lbz`

Comment faire pour ajouter l'instruction `crnor` ?

L'assembleur PowerPC en CompCert

14 Modélisation d'un sous-ensemble du PowerPC

15 Sémantiques : des transitions aux traces

16 Bibliothèques communes à tous les langages de CompCert

Transitions élémentaires

Correspondent à l'exécution de l'instruction au PC courant.

```
Variable ge: genv.

Inductive state: Type :=
  | State: regset -> mem -> state.

Inductive step: state -> trace -> state -> Prop :=
  ...
```

Transitions élémentaires

Une règle qui couvre la plupart des instructions :

```
| exec_step_internal:
    forall b ofs c i rs m rs' m',
    rs PC = Vptr b ofs ->
    Genv.find_funct_ptr ge b = Some (Internal c) ->
    find_instr (Int.unsigned ofs) c = Some i ->
    exec_instr c i rs m = OK rs' m' ->
    step (State rs m) E0 (State rs' m')
```

Si le `PC` courant est `Vptr b ofs`,
si `b` est l'adresse d'une fonction interne de code `c`
si en position `ofs` dans le code `c` on a l'instruction `i`
si cette instruction s'exécute sans erreur de $(\texttt{rs}, \texttt{m})$ vers $(\texttt{rs}', \texttt{m}')$
alors on peut faire une transition de `State rs m` vers `State rs' m'`
qui ne produit aucun événement observable (trace vide `E0`).

Transitions élémentaires

Une règle particulière pour les opérations «built-in» :
des opérations spéciales qui peuvent produire des événements observables.

(Par exemple : lecture volatile, écriture volatile.)

```
| exec_step_builtin:
    forall b ofs c ef args res rs m t v m',
    rs PC = Vptr b ofs ->
    Genv.find_funct_ptr ge b = Some (Internal c) ->
    find_instr (Int.unsigned ofs) c = Some (Pbuiltin ef args res)
    external_call ef ge (map rs args) m t v m' ->
    step (State rs m) t
         (State (nextinstr(rs #GPR11 <- Vundef #GPR12 <- Vundef
                             #FPR12 <- Vundef #FPR13 <- Vundef
                             #FPR0 <- Vundef #CTR <- Vundef
                             #res <- v)) m')
```

(Voir plus loin pour une explication de `external_call`.)

Transitions élémentaires

Deux autres règles pour deux autres cas particuliers :

- L'instruction courante est `Pannot` (annotations !) :
  c'est comme une opération built-in `Pbuiltin`,
  sauf que les arguments sont ou bien des registres ou bien des emplacements de pile.

- Le `PC` pointe sur une fonction externe (non définie dans le programme) :
  on fait comme un `Pbuiltin`, avec ajout d'un événement dans la trace, et on continue à l'adresse de retour (dans le registre `LR`).

États initiaux, états finaux

États initiaux :

- Mémoire : contient juste les variables globales après initialisation de leur contenu.
- Registres :
  - `PC` pointe sur la 1ière instruction de la fonction `main`
  - `LR` contient l'adresse de retour invalide `Vint 0`
  - les autres registres sont `Vundef`

États finaux :

- `PC` contient l'adresse invalide `Vint 0` (= on vient de retourner de la fonction `main`)
- `GPR3` contient un entier r (le code de retour)

Suites de transitions et comportements

Comme d'habitude, les comportements possibles d'un programme correspondent aux suites de transitions partant de l'état initial.

Nouveauté : les événements observables associés à ces transitions définissent des traces représentant les entrées/sorties qui ont eu lieu pendant cette exécution.

Suites de transitions et comportements

(Module `Behaviors`)

Premier comportement possible : terminaison `Terminates`(t, r)

$$\text{état initial} \xrightarrow{t_1} \cdots \xrightarrow{t_n} \text{état final (code retour } r)$$

et $t = t_1 \ldots t_n$ (concaténation des traces élémentaires).

Example

$$\begin{aligned} t \;=\;& \texttt{Event\_vload Mint32 "x" (EVint 12)} \\ .\;& \texttt{Event\_annot "message" [ ]} \\ .\;& \texttt{Event\_vstore Mint32 "x" (EVint 13)} \end{aligned}$$

Le programme lit un `int` à l'adresse volatile `x`.
Le monde extérieur lui donne la valeur entière 12.
Le programme exécute `__builtin_annotation("message");`
Le programme écrit l'`int` 13 à l'adresse volatile `x`.
Le programme s'arrête.

Suites de transitions et comportements

(Module `Behaviors`)

Second comportement possible : divergence silencieuse `Diverges`(t).

$$\text{état initial} \xrightarrow{t_1} \cdots \xrightarrow{t_n} \xrightarrow{\texttt{E0}} \xrightarrow{\texttt{E0}} \xrightarrow{\texttt{E0}} \xrightarrow{\texttt{E0}} \cdots$$

Après avoir produit les interactions $t = t_1 \ldots t_n$, le programme «boucle» sans aucune interaction.

Troisième comportement : divergence réactive `Reacts`(T).

$$\text{état initial} \xrightarrow{t_1 \neq \texttt{E0}}{}_+ \xrightarrow{t_2 \neq \texttt{E0}}{}_+ \xrightarrow{t_3 \neq \texttt{E0}}{}_+ \cdots$$

Le programme ne termine jamais mais interagit infiniment souvent avec le monde extérieur, produisant la trace infinie $T = t_1.t_2.t_3 \ldots$.

Suites de transitions et comportements

(Module `Behaviors`)

Quatrième comportement : le plantage `Goeswrong`(t).

$$\text{état initial} \xrightarrow{t_1} \cdots \xrightarrow{t_n} \text{état non final} \not\longrightarrow$$

Après avoir fait les interactions $t = t_1 \ldots t_n$, le programme plante.

Cinquième comportement : le plantage immédiat `Goeswrong`(`E0`) si l'état initial n'est pas défini.

Non-déterminisme induit par le monde extérieur

Les instructions «standard» du PowerPC sont déterministes : il y a au plus une transition possible à partir d'un état donné. Les instructions built-in comme la lecture volatile peuvent introduire du non-déterminisme.

Example

Soit un programme PowerPC correspondant au code C suivant :

```
volatile int x;
int main() { while (100 / x < 10) /*skip*/; return 0; }
```

On peut avoir plusieurs comportements possibles suivant les valeurs successives que le monde extérieur met dans `x` :

20, 30, 40, 50, 60, 60, 60 ...	divergence réactive
20, 10	terminaison
100, 0	plantage

L'assembleur PowerPC en CompCert

14 Modélisation d'un sous-ensemble du PowerPC

15 Sémantiques : des transitions aux traces

16 Bibliothèques communes à tous les langages de CompCert

Bibliothèques communes

La sémantique du PowerPC fait appel à un certain nombre de bibliothèques qui sont aussi utilisées pour la sémantique de CompCert C et des autres langages intermédiaires de CompCert :

- `Integers` : entiers machine
- `Floats` : flottants
- `AST` : éléments de syntaxe abstraite
- `Values` : le type des valeurs et ses opérations
- `Memtype`, `common/Memory` : le modèle mémoire
- `Events` : événements, traces, appels externes
- `Globalenvs` : environnements globaux
- `Smallstep` : outils pour les relations de transition
- `Behaviors` : comportements observables des programmes.

Les entiers machine

(Module `Integers`)

Un ≪foncteur≫ générique, paramétré par le nombre $w \geq 1$ de bits.

Représentation principale d'un entier machine w bits : un entier mathématique (type `Z`) dans l'intervalle $[0, 2^w - 1]$.

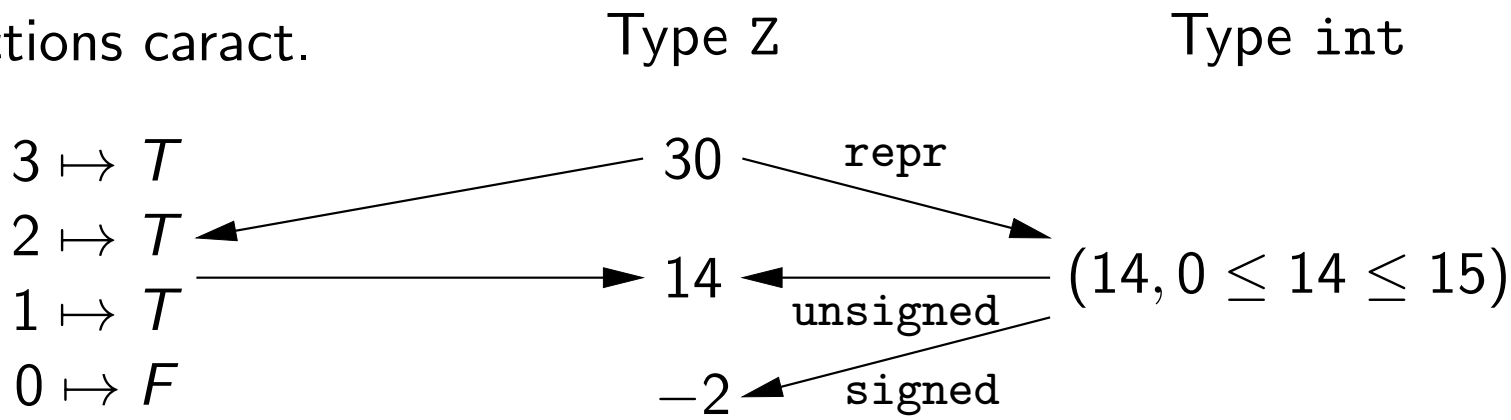
```
Definition modulus : Z := two_power_nat wordsize.
Record int: Type :=
   mkint { intval: Z; intrange: 0 <= intval < modulus }.
```

Représentation auxiliaire : une fonction caractéristique

$$\text{numéro de bit} \mapsto \text{valeur de ce bit}$$

Les entiers machine

Des fonctions de conversion (exemple dans le cas $w = 4$) :



Opérateurs usuels sur le type `int` définis

- soit par arithmétique modulo 2^w, p.ex.
  `add x y = repr(unsigned x + unsigned y)`
- soit bit-à-bit via des fonctions caractéristiques, p.ex.
  `and x y = from_bits (fun i => to_bits x i && to_bits y i)`

Les flottants IEEE 64 bits

(Module `Floats`)

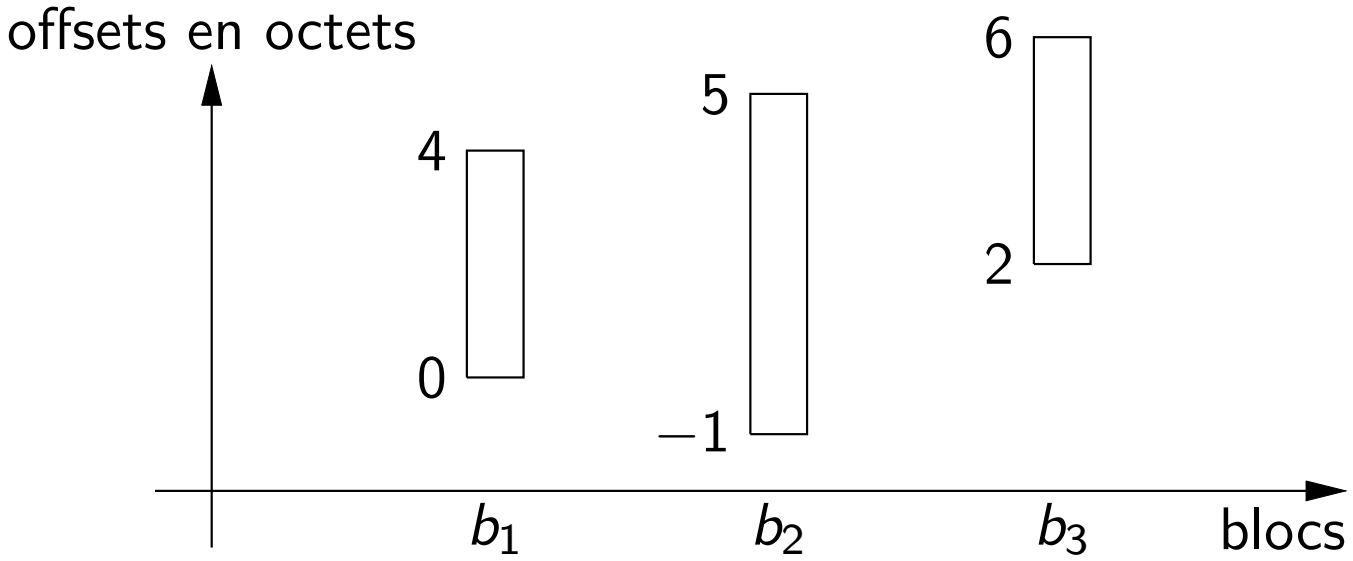
Pour le moment, les flottants sont simplement axiomatisés :

- un type abstrait `float`
- les opérations arithmétiques usuelles : `neg`, `abs`, `add`, `sub`, `mul`, `div`, comparaisons
- une opération `singleoffloat` qui arrondit un flottant 64 bits au flottant 32 bits le plus proche
- des conversions entre `float` et `int64` représentant l'encodage et le décodage au format IEEE.

Les états mémoire

(Modules `Memtype` et `Memory`)

Une vision ≪C bas niveau≫ de la mémoire comme une collection de blocs disjoints, chaque bloc étant un tableau d'octets.

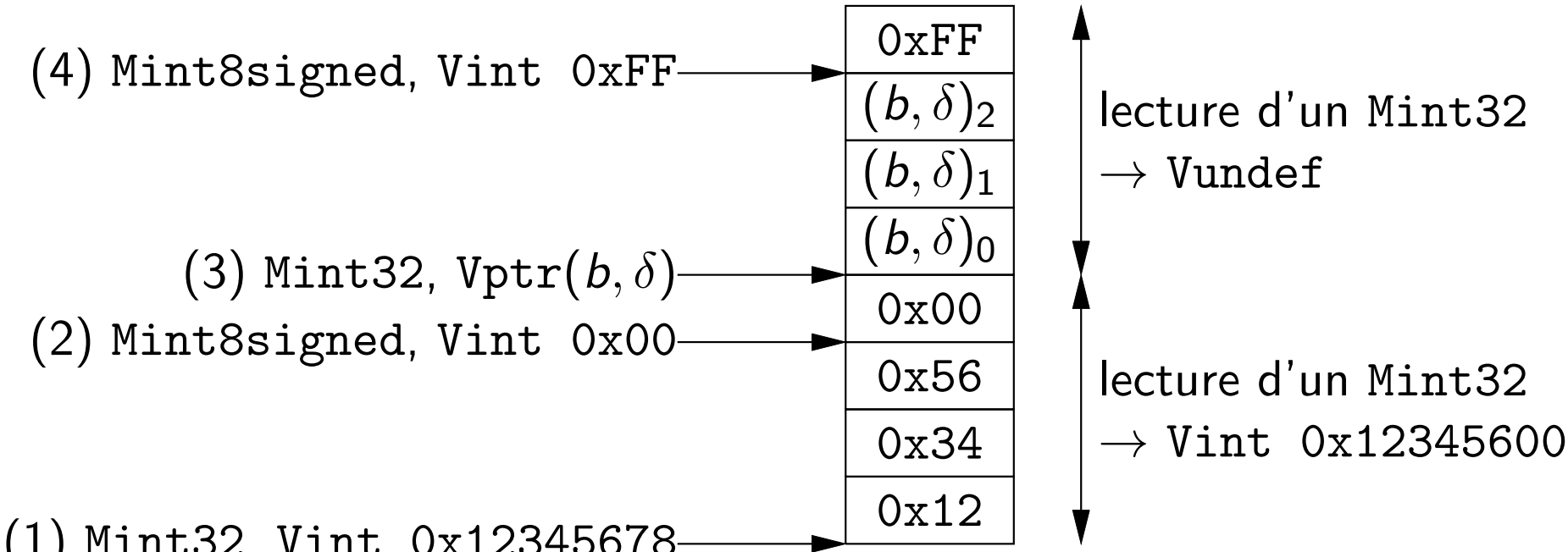


Chaque bloc a une borne basse et une borne haute.
Chaque octet de chaque bloc a une permission
(lecture/écriture, ou lecture seule, ou inaccessible car libéré).

Contenu des blocs

Une case d'un bloc peut contenir :

- `undef` (non initialisé)
- un octet (entier dans $[0, 255]$)
- un fragment de pointeur $(b, \delta)_i$ pour $i = 0, \ldots, 3$



Les principales opérations sur la mémoire

- `alloc` *m lo hi* : `mem` × `block`

  Alloue un nouveau bloc, de bornes *lo* et *hi*, de taille $hi - lo$ octets.
  Renvoie l'identifiant du nouveau bloc et la mémoire modifiée.
  Le contenu du nouveau bloc est non initialisé.

  Note 1 : on ne réutilise jamais un identifiant de bloc.
  Note 2 : `alloc` n'échoue jamais $\rightarrow$ mémoire infinie.

- `free` *m b lo hi* : `option mem`

  Libère (marque comme inaccessibles) les octets *lo* à $hi - 1$ du bloc *b*.

  Note 1 : l'identifiant *b* ne devient pas réutilisable pour autant.
  Note 2 : `free` peut échouer, si ≪double `free`≫ p.ex.

Les principales opérations sur la mémoire

- `load` *chunk m b ofs* : `option val`
- `store` *chunk m b ofs v* : `option mem`

Lit ou écrit une valeur de type&taille *chunk* dans le bloc *b* à la position *ofs*.

Échoue si :

- Accès hors des bornes
- Permissions insuffisantes (p.ex. le bloc a été libéré)
- *ofs* n'est pas bien aligné.

chunk décrit à la fois le nombre d'octets accédés et comment interpréter ces octets :

`Mint8unsigned`	`Mint32` (inclut les pointeurs)
`Mint8signed`	`Mfloat32`
`Mint16unsigned`	`Mfloat64`
`Mint16signed`	

Syntaxe des programmes

(Module `AST`)

Dans tous les langages CompCert, un programme complet se compose

- D'une liste de fonctions : (nom de la fonction, définition)
  La définition est soit `Internal` f (f = corps de la fonction)
  soit `External` ef (ef = built-in ou déclaration d'une fonction externe)
- D'une liste de variables globales :
  (nom, flag «readonly», flag «volatile», initialiseur, infos suppl.)
- Du nom de la fonction `main` (le point d'entrée).

Syntaxe des programmes

Le type des corps de fonctions f et des infos supplémentaires vi sur les variables dépend du langage considéré :

Langage	Fonctions f	Infos suppl. vi
Asm PowerPC	Listes d'instructions	Aucune
CompCert C	Type de retour + paramètres formels + variables locales + corps de la fonction	Type C
RTL	Signature de type + paramètres formels + control-flow graph	Aucune

Les environnements globaux

(Module `Globalenvs`)

Tous les langages de CompCert partagent également une notion d'environnement global d'exécution, paramétré par

- le type F des définitions de fonctions
- le type V des infos supplémentaires sur les variables

Un environnement global se compose de 3 dictionnaires :

1. nom de fonction ou de variable globale $\mapsto$ bloc
2. bloc de fonction $\mapsto F$
3. bloc de variable globale $\mapsto V$

Les blocs de fonctions sont des entiers négatifs (-1, -2, ...)
qui n'existent pas dans l'état mémoire.
Les blocs de variables globales sont des entiers positifs (1, 2, ...)
et désignent des blocs valides de l'état mémoire.

Les environnements globaux

(Module `Globalenvs`)

Principales opérations sur les environnements globaux :

- `Genv.find_symbol` *ge ident* : `option block`
  Trouve le bloc associé à un nom global (variable ou fonction)
- `Genv.find_funct_ptr` *ge b* : `option` *F*
  Trouve la définition associée à un bloc de fonction
- `Genv.find_var_info` *ge b* : `option` *V*
  Trouve les infos associées à un bloc de variable
- `Genv.globalenv` *p* : `Genv.t` *F V*
  Construit l'environnement global pour le programme *p*
- `Genv.initmem` *p* : `option mem`
  Construit l'état mémoire initial pour le programme *p*

Environnement global et état mémoire initial

Par exemple, pour le programme C suivant :

```
int x = 42;
char y;
void f(...) { ... }
int main()  { ... }
```

Table des symboles :

$\texttt{x} \mapsto 1; \texttt{y} \mapsto 2;$

$\texttt{f} \mapsto -1; \texttt{main} \mapsto -2$

Table des fonctions :

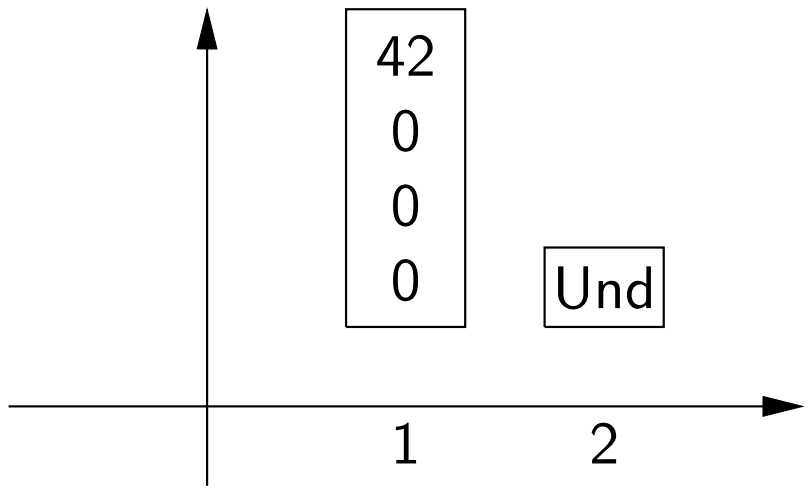
$-1 \mapsto$ `void f(...) {...}`

$-2 \mapsto$ `int main()  {...}`

Table des infos de variables :

$1 \mapsto \texttt{int}; 2 \mapsto \texttt{char}$

État mémoire initial :



Fonctions externes et événements observables

(Module `Events`)

Une notion ≪fourre-tout≫ qui permet d'unifier le traitement de :

- Les opérateurs spéciaux connus du compilateur :
  lectures et écritures volatiles ; `memcopy` ; `malloc` et `free`.
- Les fonctions ≪built-in≫ spécifiques au processeur :
  p.ex. `__builtin_fabs` pour le PowerPC.
- Les fonctions déclarées `extern` mais non définies dans le programme :
  appels systèmes, fonctions de bibliothèque.

Certaines fonctions externes sont ≪inlinées≫ par le compilateur ; les autres restent sous forme d'appels de fonctions.

Fonctions externes et événements observables

(Module `Events`)

Souvent (mais pas toujours), l'exécution d'une fonction externe produit un événement observable qui va s'ajouter à la trace d'observables du programme.

Le rôle de l'événement est double :

1. Informer le monde extérieur d'une action «visible» du programme.
   J'appelle `putchar('x')`
   J'écris 42 dans la variable volatile `y`

2. Demander au monde extérieur le résultat de cette action.
   J'ai appelé `getchar()` *; quel est le caractère lu ?*
   Je lis un `int32` *depuis la variable volatile* `x` *; quel est le résultat ?*

Sémantique des fonctions externes

(Module `Events`)

Définie par un prédicat

$$\texttt{external\_call}\ ef\ ge\ vargs\ m\ t\ vres\ m'$$

qui relie :

- la fonction externe ou opérateur spécial *ef*
- l'environnement global *ge*
- la liste des valeurs arguments *vargs* fournis par le programme
- l'état mémoire ≪avant≫ *m*
- la trace d'événements observables *t*
- la valeur du résultat *vres*
- l'état mémoire ≪après≫ m'

Voir dans `Events` la définition par cas suivant le genre de *ef*.

Cinquième partie V

Le langage CompCert C

Le langage CompCert C

17 Syntaxe

18 Sémantique

19 L'interprète de référence

Les types

(Module `Csyntax`)

Cf. déclaration du type Coq «`type`» et ses commentaires.

Points à noter :

- Les entiers et les flottants portent une taille explicite :
  `char` est `I8`, `short` est `I16`, `int` et `long` sont `I32`.
  `float` est `F32`, `double` est `F64`.
- Pas de type entier 64 bits (`long long`)
  ni de flottant $>$ 64 bits (`long double`).
- Pas de types nommés par `typedef` : ils sont expansés plus tôt.
- Traitement des `struct` et des `union`.

Traitement de `struct` et `union`

Les types `struct` et `union` sont «expansés». Exemple en C :

```
struct list { int hd; struct list * tl; };
struct msg  { char * name; struct list * queue; };
struct msg mymessage;
```

Équivalent en CompCert C :

```
mymessage :
  struct msg {
    char * name;
    struct list {
      int hd;
      comp_ptr(list) tl;
    } * queue;
  }
```

`comp_ptr(list)` signifie «pointeur vers la `struct` ou l'`union` englobante nommée `list`».

Les expressions

Cf. déclaration du type Coq `expr` et ses commentaires.

Points à noter :

- Chaque expression (et ses sous-expressions) est annotée par son type.
- Tous les opérateurs de C sont disponibles.
- Quelques-uns sont des formes dérivées :

```
   a[i]   ≡   *(a + i)
    ++a   ≡   a += 1
a && b    ≡   a? (b? 1 : 0) : 0
a || b    ≡   a? 1 : (b? 1 : 0)
```

- L'opérateur `valof` marque explicitement l'utilisation d'une l-value en tant que r-value.
- Certaines formes d'expressions n'apparaissent que pendant l'évaluation : `val(Vptr...)`, `val(Vundef)`, `loc` *b ofs*, `paren` *e*

Pour mémoire : l-values et r-values

l-value : expression qui dénote un emplacement mémoire.
On peut affecter dedans. (*Left-hand side of an assignment.*)

r-value : expression qui dénote une valeur.
(*Right-hand side of an assignment.*)

l-value :	$l ::= x \mid l.field \mid *r$
r-value en C :	$r ::= l \mid cst \mid r + r \mid \&l \mid l = r \mid \ldots$
r-value en CompCert :	$r ::= \texttt{valof}(l) \mid cst \mid r + r \mid \&l \mid l = r \mid \ldots$

Commandes (*statements*)

Cf. déclaration du type Coq `statement` et ses commentaires.

Points à noter :

- Les trois boucles `while`, `do...while` et `for` + `break` et `continue`
- `if`/`else` et `switch` structuré façon Misra C et Java.
  Pas de `switch` mal structuré façon Duff's device.
- Pas de déclarations locales à un bloc. Les variables sont soit globales soit déclarées au début de la fonction.

Pour mémoire : Duff's device

Un `switch` et une boucle qui se chevauchent sans être inclus l'un dans l'autre.

```
switch(count % 8) {
case 0:      do {    *to = *from++;
case 7:              *to = *from++;
case 6:              *to = *from++;
case 5:              *to = *from++;
case 4:              *to = *from++;
case 3:              *to = *from++;
case 2:              *to = *from++;
case 1:              *to = *from++;
        } while(--n > 0);
}
```

Malheureusement toujours permis en ISO C.

Fonctions et programmes

Une fonction = type de retour + paramètres + variables locales + corps.

Un programme = variables globales + fonctions + point d'entrée `main`.

Le langage CompCert C

17 Syntaxe

18 Sémantique

19 L'interprète de référence

La sémantique de CompCert C

(Module `Csem`)

Sémantique à transitions, de type ≪petits pas≫ :

- avec des continuations pour l'exécution des commandes et des appels de fonctions
- et des réductions sous contextes pour l'évaluation des expressions.

Deux sources de non-déterminisme :

- Les appels de fonctions externes (comme en Asm PowerPC)
- Plusieurs ordres d'évaluation sont possibles pour les expressions.

Les états de la sémantique

Contrairement à IMP et à l'Asm PowerPC, nous avons quatre sortes d'états.

Sorte 1 : `State` f s k e m

Exécution de la commande s dans la fonction f.

k est la continuation

m est l'état mémoire.

e est l'environnement local : nom de variable locale $\mapsto$ bloc mémoire.

(Et non pas nom variable $\mapsto$ valeur comme en IMP ou en Asm, car en C on peut prendre un pointeur sur une variable via l'opérateur &, et donc les variables doivent résider en mémoire.)

Les états de la sémantique

Sorte 2 : `ExprState` *f r k e m*

Évaluation de l'expression *r* (une r-value).

Les autres composantes sont comme pour les états `State` :

f : fonction courante
k : continuation
e : environnement local
m : état mémoire

Les états de la sémantique

Sorte 3 : `Callstate` *fd args k m*

Lors de l'appel de fonction, marque la transition entre la fonction appelante et la fonction appelée *fd*.

Les arguments *args* sont une liste de valeurs.

Sorte 4 : `Returnstate` *res k m*

Lors du retour de fonction, marque la transition entre la fonction qui retourne et la fonction qui l'a appelée.

res est la valeur retournée (`Vundef` si pas de valeur de retour).

Les continuations

(Cf. type `cont` dans module `Csem`.)

Encodent à la fois :

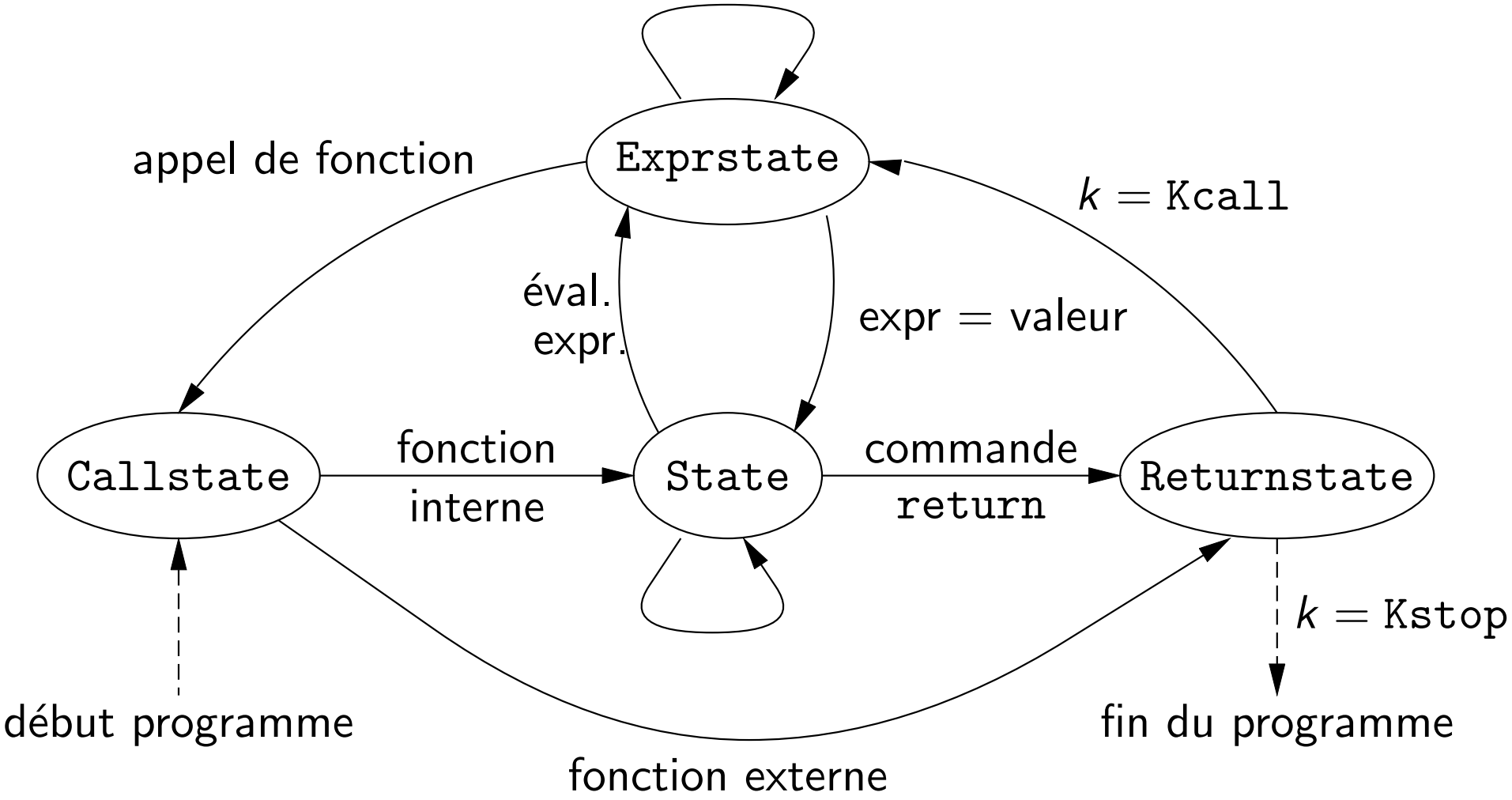
1. Une position d'une commande dans le corps de la fonction courante (comme en IMP).
2. La pile des appels de fonctions en attente.

Exemple : si la fonction principale g a appelé la fonction f, on est dans un état `State` f s k e m avec k de la forme

$$\underbrace{\ldots\ldots\ldots\ldots\ldots}_{\text{position dans } f}\ (\texttt{Kcall}\ g\ (\underbrace{\ldots\ldots\ldots\ldots\ldots}_{\text{position dans } g}\ \texttt{Kstop}))$$

L'automate des états

(Familles de transitions entre sortes d'états)



La relation de transition

(Module `Csem`)

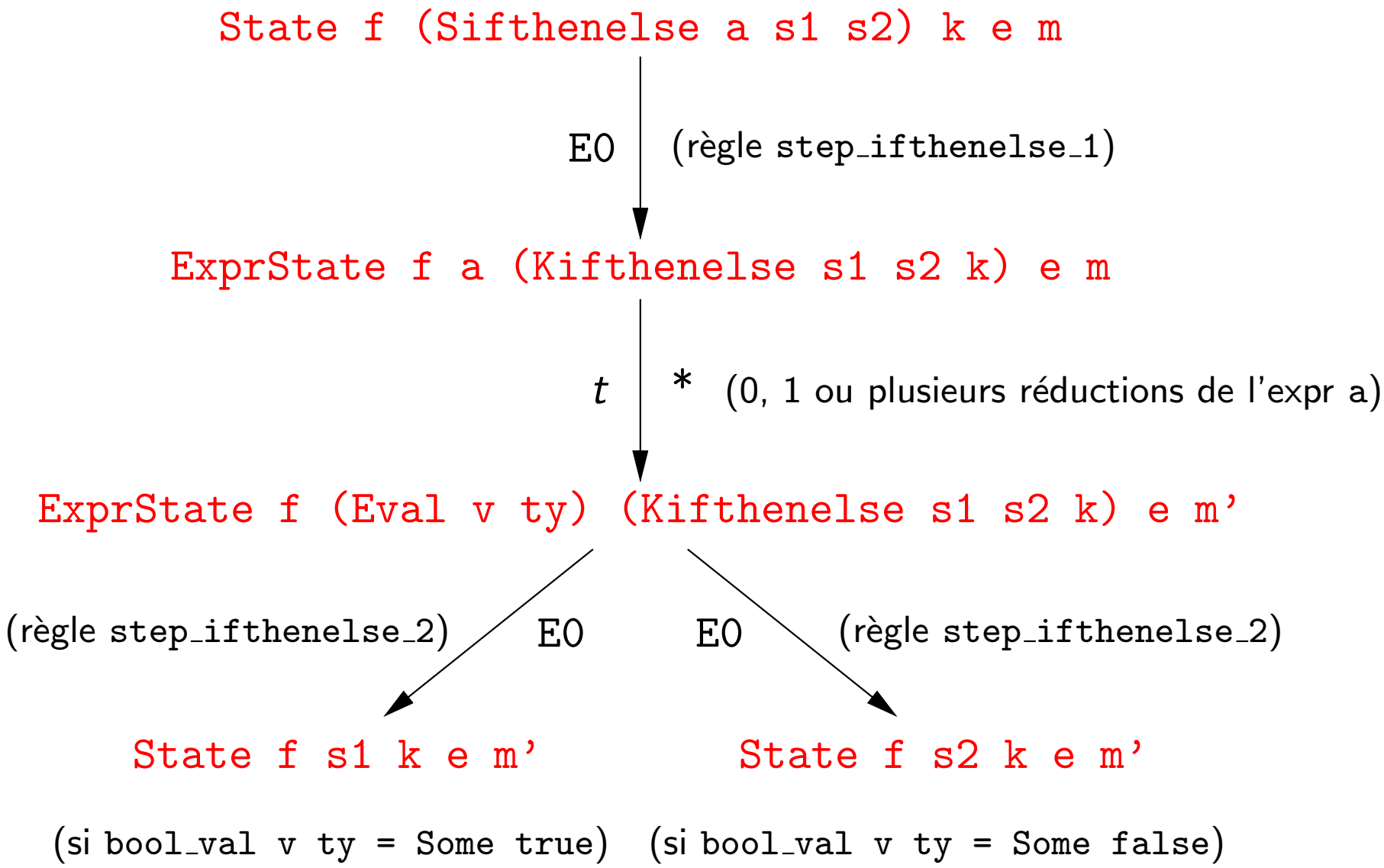
```
Variable ge: genv.

Definition step (S: state) (t: trace) (S': state) : Prop :=
  estep S t S' \/ sstep S t S'.
```

`step` est l'union de deux relations de transition (disjointes) :

1. `estep` pour les transitions liées aux expressions.
   Procède par réductions sous contexte, dans le style de la relation `red` pour IMP.
2. `sstep` pour les transitions liées aux commandes et aux appels de fonctions. Procède par focalisation + continuations, dans le style de la relation `step` pour IMP.

Exemple de transitions : `if...else`



Prise en compte des types statiques

La plupart des opérations du langage C sont surchargées : elles se comportent de plusieurs manières différentes selon les types de leurs arguments.

Donc : les fonctions de la sémantique qui opèrent sur des valeurs prennent aussi leur type en compte.

Exemple : `bool_val v t : option bool`, qui détermine la valeur booléenne (vrai/faux) d'une valeur `v`.

Prise en compte des types statiques

```
Function bool_val (v: val) (t: type) : option bool :=
  match v, t with
  | Vint n, Tint sz sg => Some (negb (Int.eq n Int.zero))
  | Vint n, Tpointer t' => Some (negb (Int.eq n Int.zero))
  | Vptr b ofs, Tint sz sg => Some true
  | Vptr b ofs, Tpointer t' => Some true
  | Vfloat f, Tfloat sz => Some (negb(Float.cmp Ceq f Float.zero))
  | _, _ => None
  end.
```

Points à noter :

- `bool_val` renvoie `None` si la valeur n'est ≪pas d'accord≫ avec le type,
- ... et aussi si $v = \texttt{Vundef}$.
- Le pointeur nul est représenté par `Vint 0`, donc tout `Vptr` est `true`.

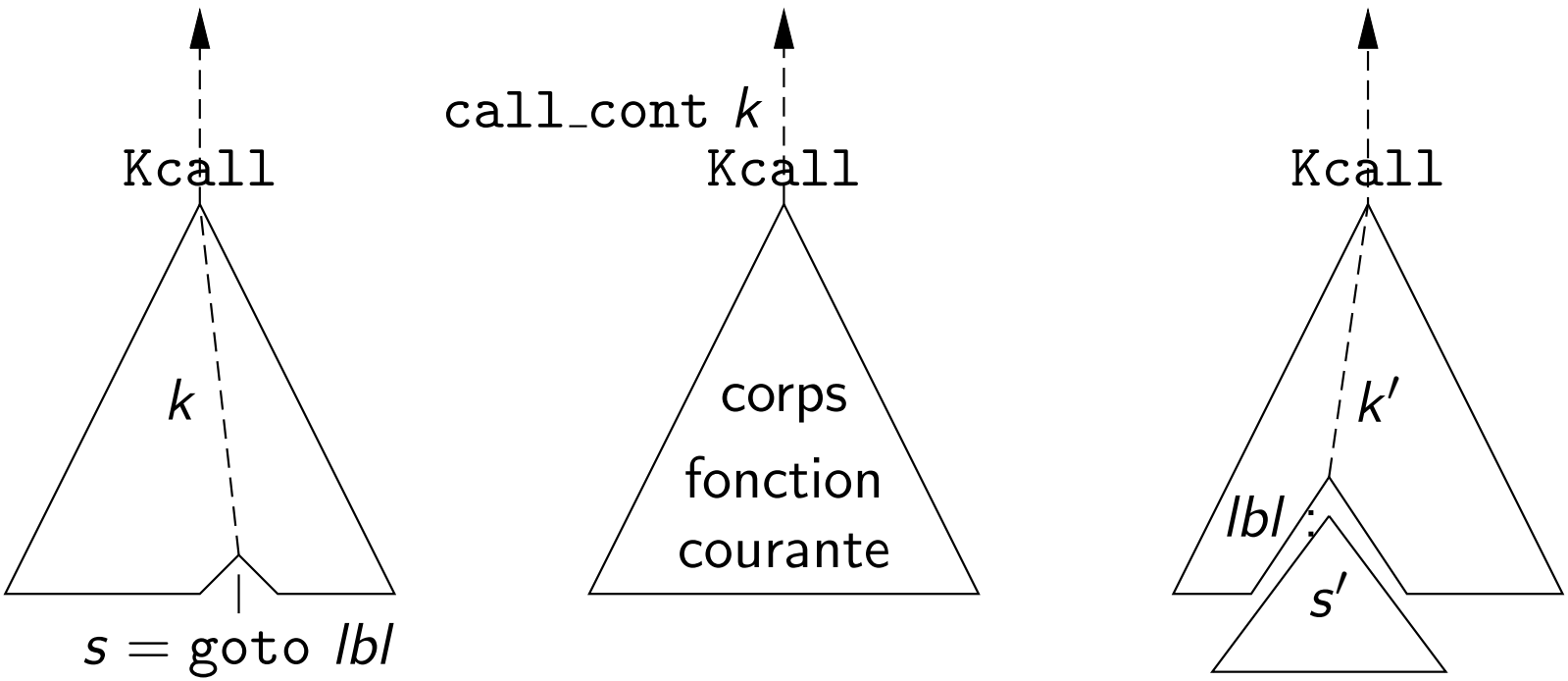
Travaux pratiques

Examiner et expliquer les règles de transition pour :

1. les séquences (règles `step_seq` et suivantes)
2. les boucles `while` (règles `step_while` et suivantes)

Le goto (règle step_goto)

1. Focaliser sur le corps de la fonction tout entier (`call_cont`).
2. Y chercher une sous-commande s' étiquetée *lbl* tout en construisant la continuation k' correspondant à la position de s' (`find_label`).



Entrée dans une fonction (règle `step_internal_function`)

On appelle une fonction interne f avec les arguments $v_1, \ldots, v_k$.

$$\begin{array}{l}\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \{ \\ \qquad \tau'_1\ y_1; \ldots; \tau'_m\ y_m; \\ \qquad cmd; \\ \}\end{array}$$

1. Allouer des blocs mémoire pour les paramètres x_i et les variables locales y_i ; construire l'environnement local correspondant. (Prédicat `alloc_variables`.)
2. Initialiser les blocs des paramètres x_i par les valeurs des arguments v_i. Vérifier que $k = n$. (Prédicat `bind_parameters`.)
3. Exécuter le corps de la fonction, cmd.

Sortie d'une fonction

Trois cas où la fonction retourne :

1. Le contrôle atteint la fin du corps de la fonction : commande courante = `skip`, continuation = `Kstop` ou `Kcall`. (Règle `step_skip_call`.)
2. Un `return;` explicite, sans argument. (Règle `step_return_0`.)
3. Un `return` *e*; explicite, avec un argument *e*, une fois que *e* a été évalué en une valeur. (Règle `step_return_2`.)

Dans les 3 cas, on libère (= rend invalides) les blocs mémoire correspondant aux variables locales.

Dans les cas 2 et 3, on utilise `call_cont` pour "sauter" toutes les commandes restant à faire dans la fonction courante.

La valeur de retour est `Vundef` dans les cas 1 et 2, et le ≪cast≫ de la valeur de *e* dans le case 3.

Les «casts»

Fonction `sem_cast` du module `Csem`

`sem_cast` v t_1 t_2 : `option val` convertit
une valeur v du type t_1 vers le type t_2, en deux temps :

1. Analyser t_1 et t_2 pour reconnaître le type de conversion
   (e.g. entier vers flottant, ou pointeur vers pointeur, ou . . .)
   (Fonction `classify_cast` du module `Csyntax`).
2. Vérifier que v est bien du type attendu et faire la conversion.

Travaux pratiques : examiner les différents cas de `sem_cast`.
Quels sont ceux où la sémantique de CompCert est plus définie que celle du standard ISO C ?

L'évaluation des expressions

Par une suite de réductions :

$$\texttt{estep} \;:\; (\texttt{ExprState}\ f\ a\ k\ e\ m) \;\stackrel{\texttt{E0}}{\rightarrow}\; \begin{array}{l}(\texttt{ExprState}\ f\ a'\ k\ e\ m') \\ \text{ou}\quad (\texttt{CallState}\ \ldots)\end{array}$$

Seuls l'expression a et l'état mémoire m' changent
(sauf si on exécute un appel de fonction).

On itère les réductions jusqu'à ce que $a = \texttt{Eval}\ v\ ty$.

L'ordre d'évaluation des expressions

Le standard ISO C 99 dit :

> *6.5[3] Except as specified later (for the function-call* `()`, `&&`, `||`, `?:`, *and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.*

Un point délicat de la sémantique :

	Ordre d'évaluation largement non-spécifié
$+$	effets de bord dans les expressions (p.ex. `x++`)
$=$	sémantique non-déterminisme

L'ordre d'évaluation

```
int a() { printf("a"); return 1; }
int b() { printf("b"); return 2; }
int c() { printf("c"); return 3; }

... a() + (b() + c()) ...
```

On a 6 ordres d'évaluation possibles, et observables :

`abc` `acb` `bac` `bca` `cab` `cba`

Les points de séquence

Quelques opérateurs donnent des garanties sur l'ordre d'évaluation. C'est spécifié indirectement via la notion de point de séquencement (*sequence point*).

$$e_1 \overset{\downarrow}{,}\, e_2 \qquad e_1 \overset{\downarrow}{\&\&}\, e_2 \qquad e_1 \overset{\downarrow}{||}\, e_2 \qquad e_1 \overset{\downarrow}{?}\, e_2 : e_3$$

(Plus : à la fin d'une expression top-level ; à l'entrée dans une fonction.)

Lorsque l'évaluation franchit un point de séquencement :

- tous les effets apparaissant avant le point ont été exécutés ;
- les calculs apparaissant après le point n'ont pas commencé.

Exemples de comportements garantis

En supposant `x == 0` initialement :

`x = 1, x` $\rightarrow$ `1` ✔

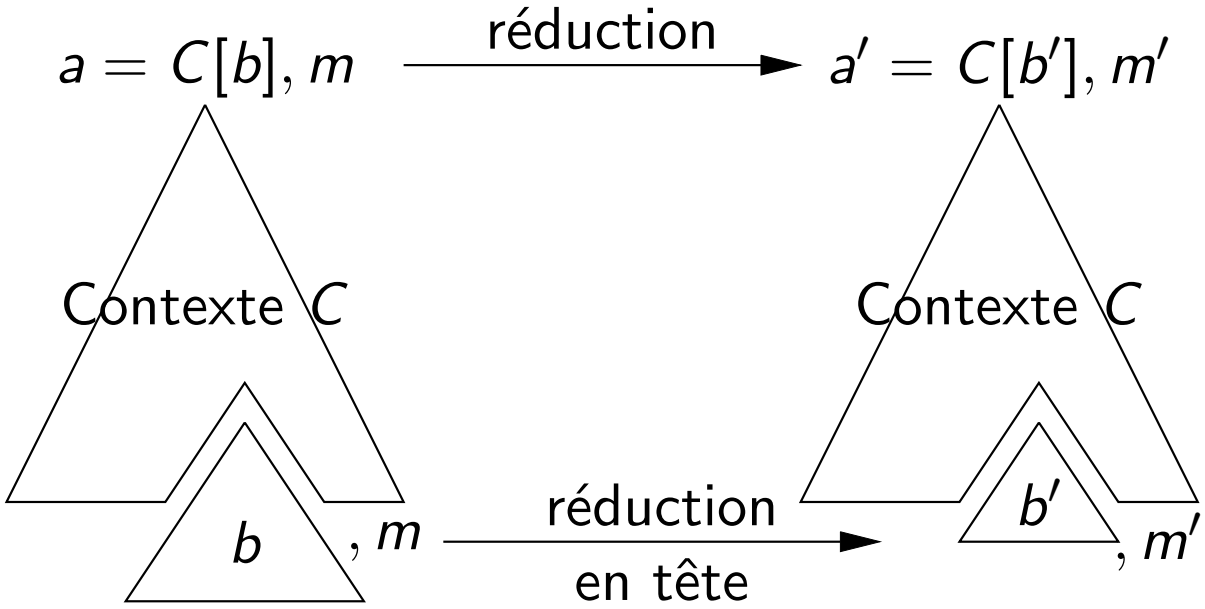
`x = 1, x` $\rightarrow$ `0` ✘

`x == 0 ? 1 : (10 / x)` $\rightarrow$ `1` ✔

`x == 0 ? 1 : (10 / x)` $\rightarrow$ `division par zéro` ✘

Comment formaliser tout cela ?

Utilisation d'une sémantique par réductions sous contextes (cf. section 9) :



Restreindre les contextes C pour interdire de réduire «après» un point de séquencement, tout en permettant plusieurs décompositions $a = C_1[b_1] = C_2[b_2] = \ldots$ correspondant à tous les ordres d'évaluation permis.

Réductions en tête

Deux relations de réduction en tête, une pour les l-values, l'autre pour les r-values :

$$\begin{aligned} \texttt{lred} &: (l, m) \to (l', m') \\ \texttt{rred} &: (r, m) \to (r', m') \end{aligned}$$

Plus : un prédicat spécial qui reconnaît les appels de fonctions :

$$\texttt{callred} : r \to (\text{fonction à appeler}, \text{arguments}, \text{type de retour})$$

Réductions en tête de l-values

```
| red_var_local: forall x ty m b,
    e!x = Some(b, ty) ->
    lred (Evar x ty) m
         (Eloc b Int.zero ty) m
| red_var_global: forall x ty m b,
    e!x = None ->
    Genv.find_symbol ge x = Some b ->
    type_of_global b = Some ty ->
    lred (Evar x ty) m
         (Eloc b Int.zero ty) m
```

Une variable `x` se réduit en un emplacement `Eloc b ofs` où `b` est le bloc associé à `x` dans l'env. local ou l'env. global l'offset `ofs` est toujours zéro.

Réductions en tête de l-values

```
| red_deref: forall b ofs ty1 ty m,
    lred (Ederef (Eval (Vptr b ofs) ty1) ty) m
         (Eloc b ofs ty) m
```

Un déréférencement `*r` où `r` est déjà réduit en une valeur de pointeur `Eval (Vptr b ofs)` produit l'emplacement `Eloc b ofs`.

Réductions en tête de l-values

```
| red_field_struct: forall b ofs id fList f ty m delta,
    field_offset f fList = OK delta ->
    lred (Efield (Eloc b ofs (Tstruct id fList)) f ty) m
         (Eloc b (Int.add ofs (Int.repr delta)) ty) m
| red_field_union: forall b ofs id fList f ty m,
    lred (Efield (Eloc b ofs (Tunion id fList)) f ty) m
         (Eloc b ofs ty) m.
```

Un accès à un champ `l.f`
où `l` est déjà réduit en l'emplacement `Eloc b ofs`
se réduit en l'emplacement `Eloc b` (`ofs` + `delta`).

Si on accède dans une struct, `delta` est la position du champ `f` dans cette struct.

Si on accède dans une union, `delta` = 0.

Réductions en tête de r-values

Conversion d'une l-value déjà réduite `Eloc b ofs` en une r-value :

```
| red_rvalof: forall b ofs ty m v,
    load_value_of_type ty m b ofs = Some v ->
    rred (Evalof (Eloc b ofs ty) ty) m
         (Eval v ty) m
```

`load_value_of_type` discrimine suivant le type accédé :

- Si type scalaire (entier, flottant, pointeur) : lecture en mémoire à l'adresse (`b`, `ofs`)
- Si type tableau ou fonction : on renvoie le pointeur `Vptr b ofs`.
- Si type struct ou union : non défini en CompCert C.

Réductions en tête de r-values

Réduction d'un opérateur binaire dont les deux arguments ont déjà été réduits en des valeurs :

```
| red_binop: forall op v1 ty1 v2 ty2 ty m v,
    sem_binary_operation op v1 ty1 v2 ty2 m = Some v ->
    rred (Ebinop op (Eval v1 ty1) (Eval v2 ty2) ty) m
         (Eval v ty) m
```

Tout le travail est fait par `sem_binary_operation`, qui est définie par cas sur l'opérateur op, puis par cas sur les valeurs et les types des deux arguments.

Travaux pratiques : examiner le cas `op = Oadd`, c.à.d. la fonction `sem_add`.

Réductions en tête de r-values

Affectation `l = r` où `l` est `Eloc b ofs` et `r` est `Eval v2` :

```
| red_assign: forall b ofs ty1 v2 ty2 m v m',
    sem_cast v2 ty2 ty1 = Some v ->
    store_value_of_type ty1 m b ofs v = Some m' ->
    rred (Eassign (Eloc b ofs ty1) (Eval v2 ty2) ty1) m
         (Eval v ty1) m'
```

Note 1 : ≪cast≫ implicite de la valeur de `r` vers le type de `l`.

Note 2 : `store_value_of_type` effectue une écriture mémoire si `l` est de type scalaire, et fait une erreur sinon.

Note 3 : la valeur de l'affectation `l = r` est la valeur écrite (après cast).

Travaux pratiques

Déchiffrer les règles

- `red_assignop` pour la construction `l += r`
- `red_postincr` pour la construction `l++`

Les contextes de réduction

Comme pour IMP, un contexte est une expression avec un ≪trou≫ [] représenté par une fonction Coq `expr -> expr`.

Le prédicat `context from to C` vérifie que la fonction `C` est un contexte valide :

- On interdit les contextes de la forme `e, C` ou `e ? C : e'` ou `e ? e' : C` car ils permettraient de réduire après un point de séquencement.
- On vérifie que `C` préserve les ≪sortes≫ (l-value ou r-value) : si on remplace le trou par une expression `e` de sorte `from`, on obtient bien une expression `C e` de sorte `to`.

Les réductions sous contexte

Trois règles de réduction sous contexte qui définissent la transition `estep` : `step_lred`, `step_rred` et `step_call`.

Regardons `step_rred` :

```
| step_rred: forall C f a k e m a' m',
    rred a m a' m' ->
    not_stuck e (C a) m ->
    context RV RV C ->
    estep (ExprState f (C a) k e m)
       E0 (ExprState f (C a') k e m')
```

Tout semble normal sauf cette hypothèse `not_stuck`...

Non-déterminisme et erreurs à l'exécution

Lorsque plusieurs ordres d'évaluation sont possibles, certains peuvent faire ≪planter≫ le programme et pas d'autres.

Exemple 1 : si `x` vaut 0 initialement,

```
(x = 1) + (100 / x)
```

fait une division par zéro si on évalue `100 / x` en premier,
mais renvoie 101 si on évalue `x = 1` en premier.

Exemple 2 : si `f` est une fonction qui boucle,

```
(1 / 0) + f()
```

fait une division par zéro si on évalue `1 / 0` en premier,
mais diverge sans planter si on évalue `f()` en premier.

Non-déterminisme et erreurs à l'exécution

Le standard ISO C laisse entendre qu'un programme est indéfini dès qu'il existe un ordre d'évaluation qui produit une erreur.

(Ceci laisse toute liberté au compilateur pour choisir un ordre d'évaluation, sans se soucier d'introduire des comportements erronés.)

Problème : comment refléter cela dans la sémantique ? Avec notre convention «erreur à l'exécution = aucune réduction n'est possible», on obtient naturellement qu'un programme est erroné seulement si tous les ordres d'évaluation produisent une erreur.

Non-déterminisme et erreurs à l'exécution

```
| step_rred: forall C f a k e m a' m',
    rred a m a' m' ->
    not_stuck e (C a) m ->
    context RV RV C ->
    estep (ExprState f (C a) k e m)
       E0 (ExprState f (C a') k e m')
```

Le rôle de la condition `not_stuck` est de garantir qu'aucune autre réduction de `C a` ne peut produire une erreur.

Si oui, la transition `estep` est possible.

Si non, aucune transition `estep` n'est possible, et le programme bloque comme il le doit.

Caractériser l'absence de réductions erronées

≪Une réduction de l'expression `a` peut faire une erreur≫ $\Leftrightarrow$
on peut écrire `a = C[b]` où la sous-expression `b` n'est pas une valeur et ne peut pas se réduire.

Exemple : `a` est `(1 / 0) + f()`. On peut l'écrire comme `a = C[1/0]` et la sous-expression `1/0` n'est pas une valeur et ne peut pas se réduire.

En prenant la contraposée :
≪Aucune réduction de l'expression `a` ne cause une erreur≫ $\Leftrightarrow$
pour toute décomposition `a = C[b]`, soit la sous-expression `b` est une valeur, soit elle peut se réduire.

Cf. définition de `not_stuck` dans `Csem`.

Le langage CompCert C

17 Syntaxe

18 Sémantique

19 L'interprète de référence

Une sémantique exécutable pour CompCert C

(Module `Cexec`)

Comment rendre exécutable les relations inductives définissant la sémantique de CompCert C ? Comme nous l'avons fait pour IMP (section 10), nous pouvons réécrire la relation de transition

```
step : state -> trace -> state -> Prop
```

sous forme d'une fonction calculable

```
do_step : world -> state -> list (trace * state)
```

Trois différences par rapport à `IMP` :

- Non-déterminisme $\rightarrow$ une liste de résultats possibles au lieu d'une option. Liste vide de résultats = erreur.
- Un résultat est une paire (trace d'observables, nouvel état).
- Le paramètre `world` modélise le monde extérieur et sert à exécuter les interactions avec ce dernier.

Équivalence entre les écritures inductive et fonctionnelle

Correction de l'interprète vis-à-vis de la sémantique

```
Lemma do_step_sound:
  forall w S t S',
  In (t, S') (do_step w S) -> Csem.step ge S t S'.
```

Complétude de l'interprète vis-à-vis de la sémantique

```
Lemma do_step_complete:
  forall w S t S' w',
  possible_trace w t w' -> Csem.step ge S t S' ->
  In (t, S') (do_step w S).
```

Interprète vs. sémantique définie inductivement

Avantages

- Animer la sémantique de C est très utile pour comprendre le comportement d'un programme.
  - L'interprète montre tous les comportements possibles ;
  - il permet de tester la sémantique.
  - Il permet de comprendre pourquoi certaines valeurs ne sont pas définies.
- L'interprète renforce la confiance en la sémantique.

Inconvénients

- L'écriture de la fonction est plus compliquée que celle de la relation.
- Raisonner sur une fonction est plus compliqué que raisonner par induction sur une relation.

L'interprète de référence

L'interprète de référence `ccomp -interp` s'obtient en itérant la fonction `do_step` à partir de l'état initial, en suivant toutes ou seulement certaines branches.

Exploration exhaustive

Avec `ccomp -interp -all` :

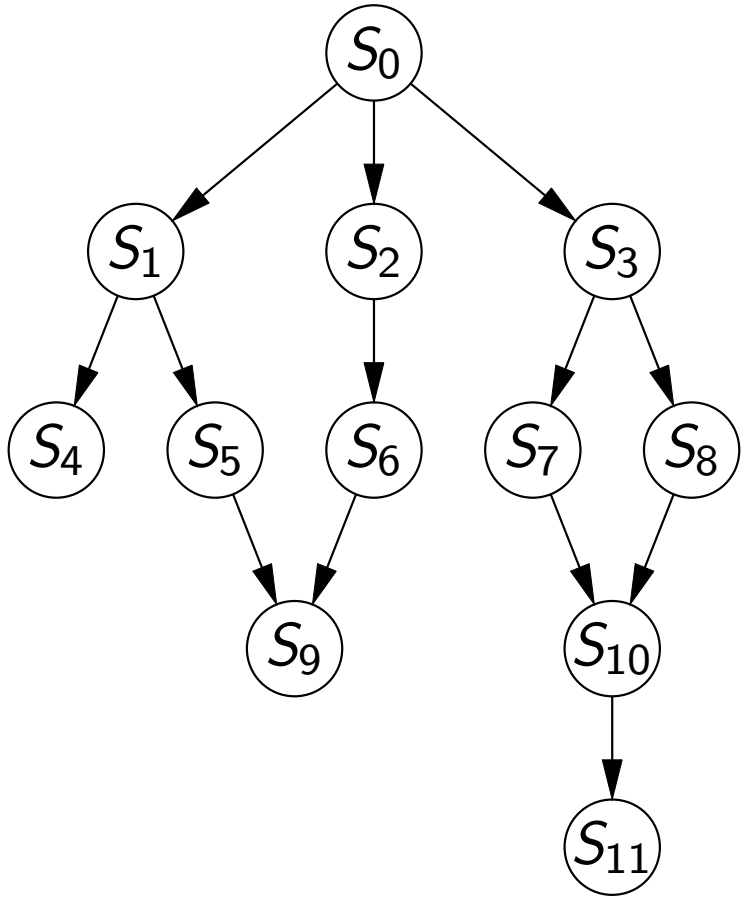
$T = 0$:

$T = 1$:

$T = 2$:

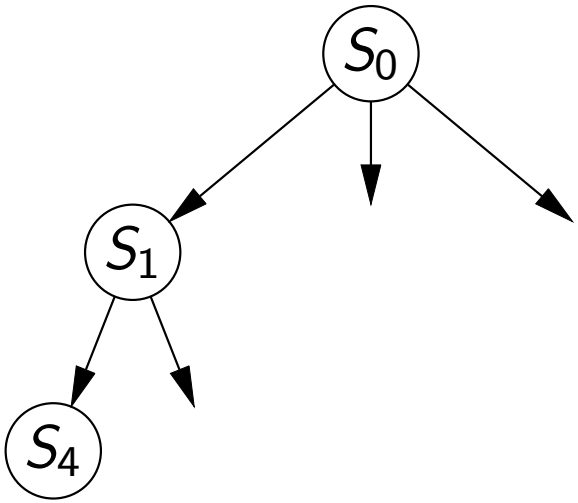
$T = 3$:

$T = 4$:

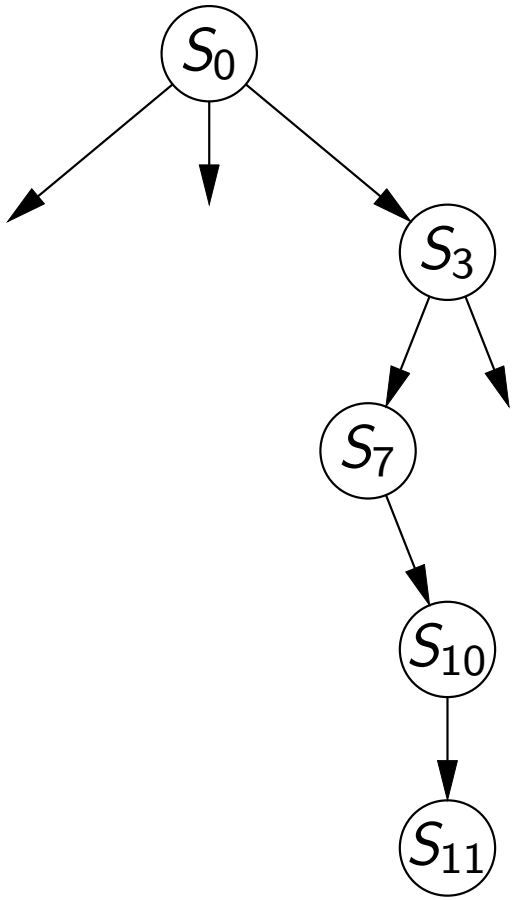


Exploration partielle rapide

Premier choix
(`ccomp -interp`) :



Randomisée
(`ccomp -interp -random`) :



DÉMO !

En pratique...

Limitations de l'interprète de référence :

- Programmes complets avec un `int main() { ... }`
- Inefficacités algorithmiques au-delà de 1000 écritures mémoire.
- Seule fonction externe disponible : `printf`
- Variables `volatile` traitées comme non `volatiles`
  (pas d'événement observable à la lecture ou à l'écriture).

Quelques utilisations :

- Vérifier qu'un (petit) programme complet s'exécute bien sans erreurs d'après la sémantique de CompCert C.
- Test aléatoire avec l'outil `Csmith` de l'U. Utah.

Sixième partie VI

Le théorème de préservation sémantique de CompCert et ses conséquences

Structure de la preuve de préservation sémantique

1. Pour chacune des 18 passes de CompCert, on prouve un théorème de simulation entre le programme en entrée P_1 et le programme transformé P_2 :
   - Simulation en avant : chaque transition de P_1 est simulée par zéro, une ou plusieurs transitions de P_2.
   - Simulation en arrière : chaque transition de P_2 est simulée par zéro, une ou plusieurs transitions de P_1.

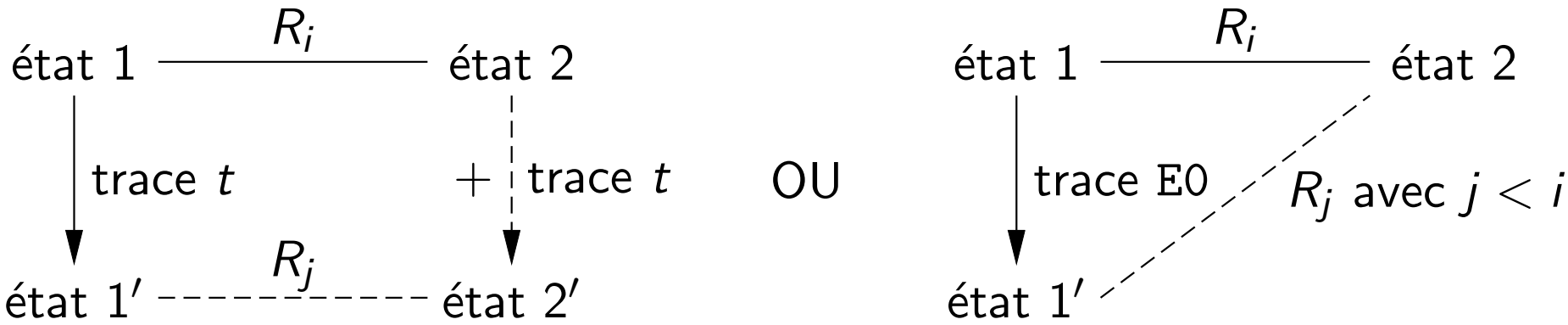
2. En composant ces 18 simulations, on obtient une simulation en arrière entre CompCert C et l'Asm PowerPC/ARM/x86.

3. Cette simulation arrière implique que les comportements observables (terminaison/divergence/etc + traces d'événements visibles) sont soit préservés, soit améliorés (l'Asm plante ≪moins souvent≫ ou ≪plus tard≫ que le source C).

Forme d'une simulation en avant

(Module `Smallstep`)

Il existe une famille de relations R_i telles que :



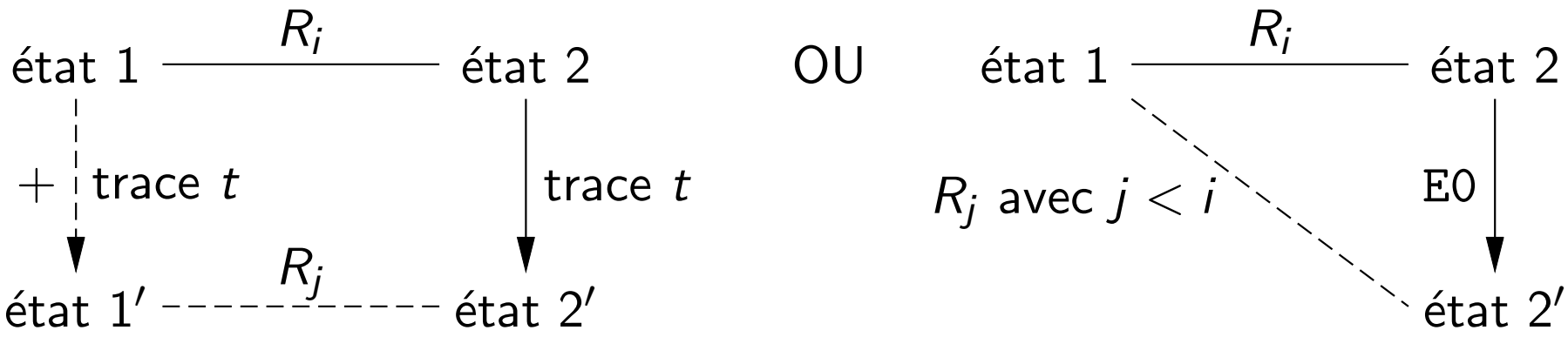
(Trait plein = hypothèse ; trait pointillé = conclusion).

Plus : conditions sur les états initiaux et les états finaux.

Forme d'une simulation en arrière

(Module `Smallstep`)

Il existe une famille de relations R_i telles que :



sous l'hypothèse que «état 1» n'est pas bloqué.

Plus difficile à démontrer qu'une simulation avant, mais plus puissante : permet au programme transformé P_2 d'avoir moins de comportements que le programme source P_1 (déterminisation lors de la compilation).

Que dit l'existence d'une simulation ?

Que les comportements des 2 programmes sont identiques ou ≪améliorés≫.

```
Definition behavior_improves (beh1 beh2: program_behavior): Prop :=
  beh1 = beh2
  \/ exists t, beh1 = Goes_wrong t /\ behavior_prefix t beh2.
```

Simulation avant : tout comportement du programme source est aussi un comportement (identique ou amélioré) du programme transformé.

```
Theorem forward_simulation_behavior_improves:
  forall beh1, program_behaves L1 beh1 ->
  exists beh2, program_behaves L2 beh2 /\ behavior_improves beh1 beh2.
```

Simulation arrière : tout comportement du programme transformé est identique à ou améliore un comportement du programme source.

```
Theorem backward_simulation_behavior_improves:
  forall beh2, program_behaves L2 beh2 ->
  exists beh1, program_behaves L1 beh1 /\ behavior_improves beh1 beh2.
```

Propriétés algébriques des simulations

(Module `Smallstep`)

Les simulations se composent entre elles :

- Si S_{12} : simulation avant de P_1 vers P_2
  et S_{23} : simulation avant de P_2 vers P_3,
  il existe S_{13} : simulation avant de P_1 vers P_3,
- Si S_{12} : simulation arrière de P_1 vers P_2
  et S_{23} : simulation arrière de P_2 vers P_3,
  il existe S_{13} : simulation arrière de P_1 vers P_3,

Sous certaines hypothèses, la simulation avant implique la simulation arrière :

- Si S : simulation avant de P_1 vers P_2
  et P_1 est *receptive* et P_2 est *determinate*,
  il existe S' : simulation arrière de P_1 vers P_2

Le théorème principal de CompCert

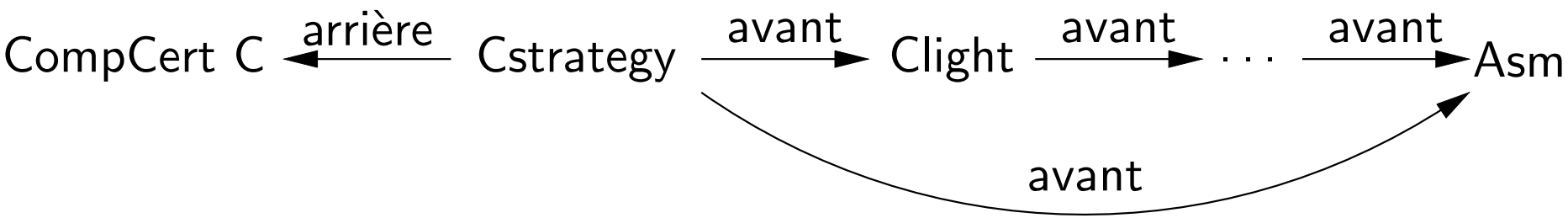
(Module `Compiler`)

Si la compilation réussit, il existe une simulation arrière entre le source CompCert C et l'assembleur PowerPC/ARM/x86 produit par le compilateur.

Le théorème principal de CompCert

(Module `Compiler`)

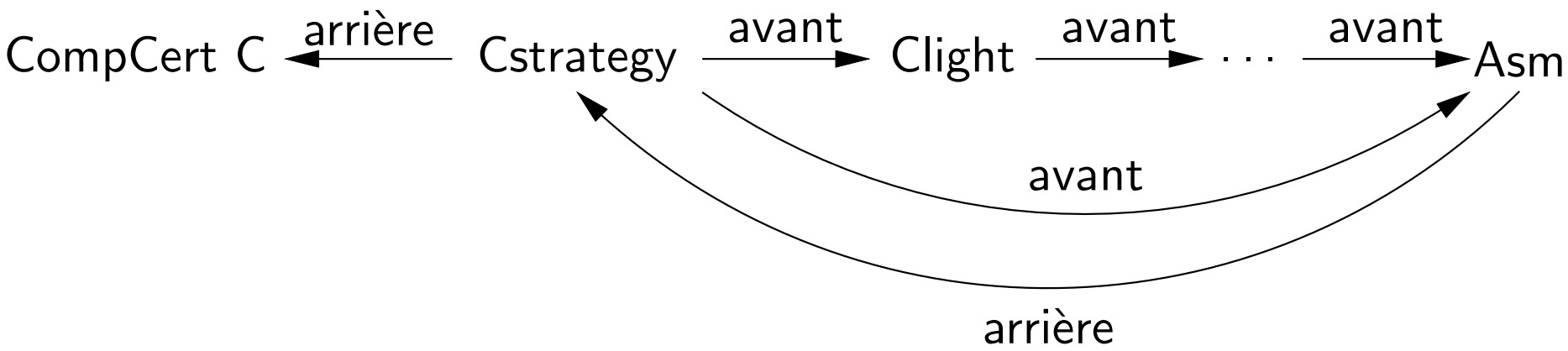
Si la compilation réussit, il existe une simulation arrière entre le source CompCert C et l'assembleur PowerPC/ARM/x86 produit par le compilateur.



Le théorème principal de CompCert

(Module `Compiler`)

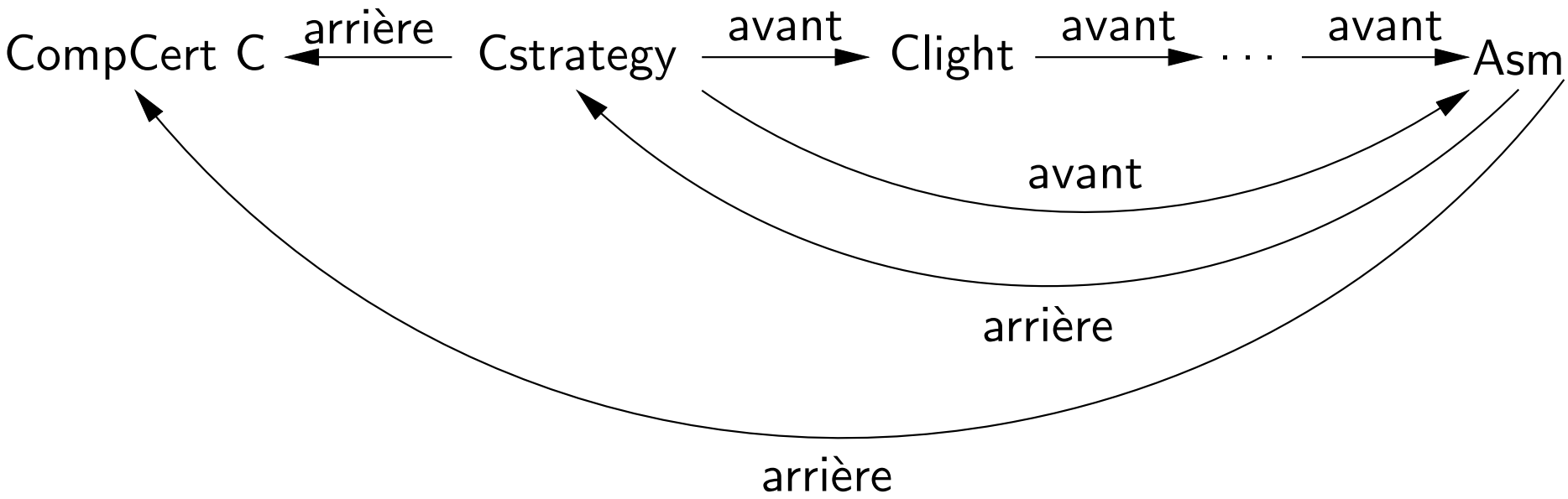
Si la compilation réussit, il existe une simulation arrière entre le source CompCert C et l'assembleur PowerPC/ARM/x86 produit par le compilateur.



Le théorème principal de CompCert

(Module `Compiler`)

Si la compilation réussit, il existe une simulation arrière entre le source CompCert C et l'assembleur PowerPC/ARM/x86 produit par le compilateur.



Conséquence de ce théorème

(Module `Complements`)

Tout comportement observable de l'assembleur généré est un comportement légal du programme source, possiblement amélioré.

```
Theorem transf_c_program_preservation:
  forall p tp beh,
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) beh ->
  exists beh', program_behaves (Csem.semantics p) beh'
            /\ behavior_improves beh' beh.
```

Exemple de comportement amélioré par le compilateur

```
#include <stdio.h>
int main()
{
  int x;
  printf("Crash!\n");
  x = 1 / 0;
  return 0;
}
```

D'après la sémantique de CompCert C, le comportement est `Goes_wrong`, comme le montre l'interprète de référence. (Essayez !)

Cependant, le compilateur élimine le calcul `x = 1 / 0;` car c'est du code mort. Le comportement du code généré est donc `Terminates` avec la même trace d'observables.

Exemple de comportement amélioré par le compilateur

```
#include <stdio.h>
int main()
{
  int x[2] = { 12, 34 };
  printf("x[2] = %d\n", x[2]);
  return 0;
}
```

Là aussi, la sémantique de CompCert C prédit le comportement `Goes_wrong`. (Accès hors-bornes.)

Cependant, le compilateur ne produit pas de code vérifiant les bornes des tableaux. Le code généré peut ≪planter≫ mais en général il va imprimer un entier arbitraire et terminer normalement.

Autre conséquence

(Module `Complements`)

Si le source C ne peut pas planter (parce que vérifié avec Astrée p.ex.), le comportement observable de l'assembleur produit est exactement l'un des comportements observables autorisés pour le source :

```
Theorem transf_c_program_is_refinement:
  forall p tp,
  transf_c_program p = OK tp ->
  (forall beh, program_behaves (Csem.semantics p) beh -> not_wrong beh) ->
  (forall beh, program_behaves (Asm.semantics tp) beh ->
                          program_behaves (Csem.semantics p) beh).
```

En particulier, l'assembleur produit ne peut pas planter.

Q.E.D.