

# Vérification formelle d'un modèle mémoire pour le langage C

Projet ANR ARA SSIA CompCert (<http://compcert.inria.fr>)

Sandrine Blazy, Xavier Leroy

CEDRIC-ENSIIE et INRIA Rocquencourt

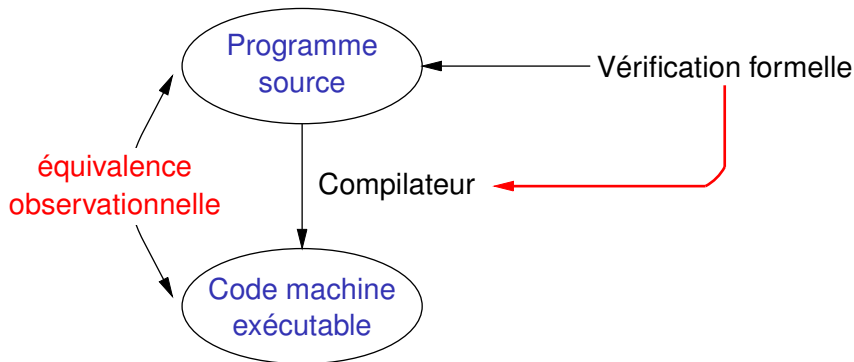
CEA-LIST, 18 mars 2008



# Plan

- 1 Vérification formelle de compilateurs
- 2 Modèle mémoire
  - Modèle abstrait
  - Modèle concret
- 3 Transformations de programmes opérant sur la mémoire
- 4 Conclusion

# Faites-vous confiance à votre compilateur ?



## Compilateur formellement vérifié :

Garantit que le code produit se comporte comme prescrit par la sémantique du programme source.

# La vérification formelle de compilateurs

Appliquer les méthodes formelles au compilateur lui-même pour établir un théorème de **préservation sémantique** :

## Théorème

*Pour tous les codes source  $S$ ,  
si le compilateur transforme  $S$  en le code machine  $C$ ,  
sans signaler d'erreur de compilation,  
et si  $S$  a une sémantique bien définie,  
alors  $C$  a la même sémantique que  $S$   
à équivalence observationnelle près.*

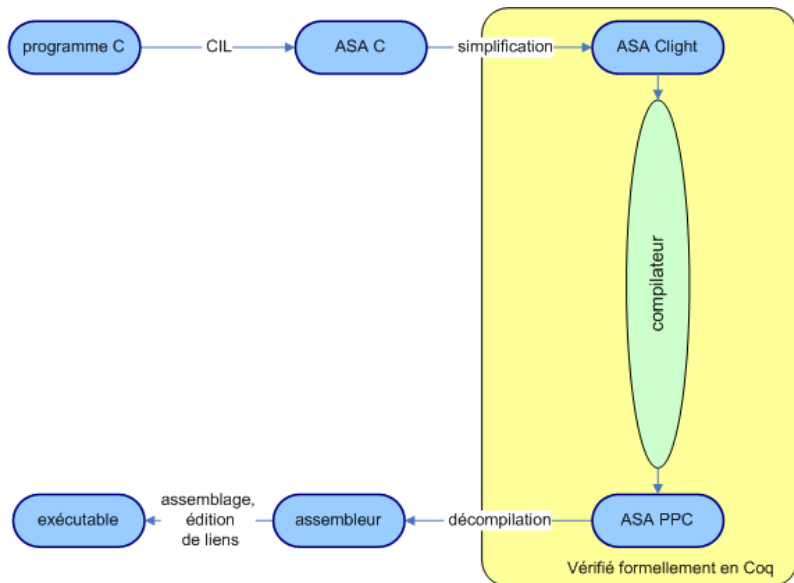
# Le projet CompCert

ANR SSIA (2005-2008)

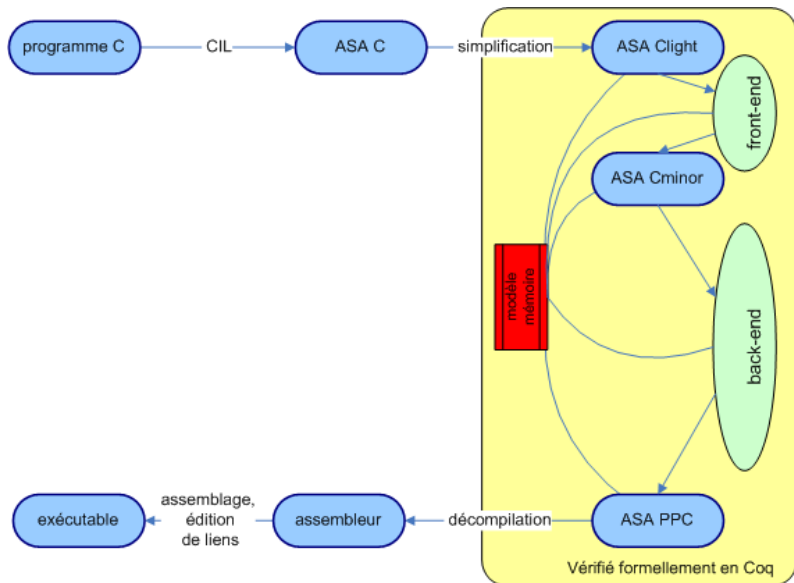
Développement d'un compilateur réaliste, formellement vérifié et utilisable dans l'embarqué.

- Langage source : un vaste sous-ensemble de **C** (pas d'instruction de saut).
- Langage cible : assembleur **Power PC**.
- Le compilateur effectue quelques **optimisations**.
- Vérification formelle en Coq.
- Les parties vérifiées du compilateur sont programmées directement dans le langage de spécification de Coq, dans un style fonctionnel pur.

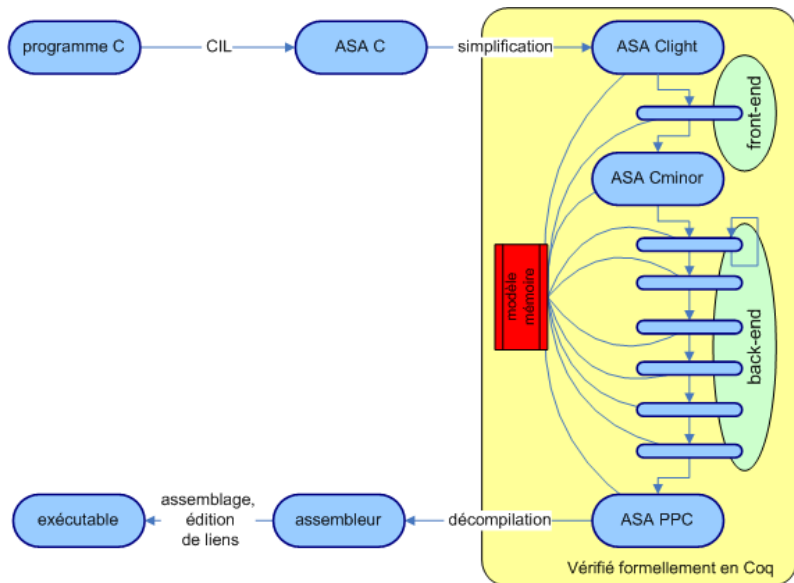
# Architecture du compilateur CompCert



# Architecture du compilateur CompCert



# Architecture du compilateur CompCert



# Méthodologie

- Découpage du compilateur en passes de transformations successives, chacune étant certifiée.
- Nécessite des sémantiques formelles pour les 10 langages du compilateur.
- Le modèle mémoire est commun à tous les langages du compilateur.
- Chaque passe peut être soit certifiée directement, soit effectuée par du code non certifié + vérification des résultats par un vérificateur certifié.

# Plan

1 Vérification formelle de compilateurs

2 **Modèle mémoire**

- Modèle abstrait
- Modèle concret

3 Transformations de programmes opérant sur la mémoire

4 Conclusion

# Modèle mémoire

(généralités)

But : définir la **géographie** de la mémoire, les **opérations** de gestion (lire, écrire, allouer, libérer), et leurs **propriétés**.

Caractéristiques :

- généricité
- séparation en blocs indépendants de taille variable
- existence d'une valeur indéfinie
- tests aux bornes des blocs
- adapté à la preuve sur machine

# Un modèle mémoire pour C

(niveau d'abstraction)

Quel **niveau d'abstraction** ?

- Trop concret (tableau d'octets) :
  - ▶ Les propriétés attendues ne sont pas exprimables (par ex., séparation de zones).
  - ▶ Ne permet pas de raisonner sur la mémoire.
- Trop abstrait (collection de blocs disjoints) :
  - ▶ Ne permet pas de définir l'arithmétique de pointeurs.
  - ▶ Les sémantiques deviennent incorrectes.

# Un modèle mémoire pour C

(niveau d'abstraction)

Quel **niveau d'abstraction** ?

Besoins antagonistes :

- pointeurs et arithmétique de pointeurs  
→ chevauchement partiel de zones référencées par deux pointeurs
- garanties d'isolation et de fraîcheur  
→ séparation des zones de la mémoire correspondant à 2 appels successifs à `malloc`

Certaines passes du compilateur effectuent des transformations des blocs de mémoire non triviales, nécessitant de raisonner sur la mémoire. → Plusieurs niveaux d'abstraction

# Un modèle mémoire pour C

(niveau d'abstraction)

Quel **niveau d'abstraction** ?

Besoins antagonistes :

- pointeurs et arithmétique de pointeurs  
→ chevauchement partiel de zones référencées par deux pointeurs
- garanties d'isolation et de fraîcheur  
→ séparation des zones de la mémoire correspondant à 2 appels successifs à `malloc`

Certaines passes du compilateur effectuent des transformations des blocs de mémoire non triviales, nécessitant de raisonner sur la mémoire. → Plusieurs niveaux d'abstraction

# Paramètres du modèle mémoire

Valeurs et types des données stockées en mémoire

`val ::=`

`int(n) | float(f) | ptr(b, ofs) | undef`  
`undef`  $\in$  `val`

`mementype ::=`

`int8signed | int8unsigned |`  
`int16signed | int16unsigned |`  
`int32 | float32 | float64`

Utile pour la lecture ou l'écriture d'une donnée en mémoire ( $\tau$ ) :

- taille  $|\tau|$  et alignement  $\langle\tau\rangle$      `$|\tau| = \text{sizeof}(\tau)$`      `$\langle\tau\rangle = 1$`
- garantie de compatibilité entre une écriture et une lecture ultérieure

# Paramètres du modèle mémoire

Valeurs et types des données stockées en mémoire

`val ::=`

`int(n) | float(f) | ptr(b, ofs) | undef`  
`undef`  $\in$  `val`

`mementype ::=`

`int8signed | int8unsigned |`  
`int16signed | int16unsigned |`  
`int32 | float32 | float64`

Utile pour la lecture ou l'écriture d'une donnée en mémoire ( $\tau$ ) :

- taille  $|\tau|$  et alignement  $\langle\tau\rangle$      `$|\tau| = \text{sizeof}(\tau)$`      `$\langle\tau\rangle = 1$`
- garantie de compatibilité entre une écriture et une lecture ultérieure

# Paramètres du modèle mémoire

Valeurs et types des données stockées en mémoire

`val ::=`

`int(n) | float(f) | ptr(b, ofs) | undef`  
`undef`  $\in$  `val`

`mementype ::=`

`int8signed | int8unsigned |`  
`int16signed | int16unsigned |`  
`int32 | float32 | float64`

Utile pour la lecture ou l'écriture d'une donnée en mémoire ( $\tau$ ) :

- taille  $|\tau|$  et alignement  $\langle\tau\rangle$      `$|\tau| = \text{sizeof}(\tau)$`      `$\langle\tau\rangle = 1$`
- garantie de compatibilité entre une écriture et une lecture ultérieure

# Paramètres du modèle mémoire

Valeurs et types des données stockées en mémoire

`val ::=`

`int(n) | float(f) | ptr(b, ofs) | undef`  
`undef`  $\in$  `val`

`memtype ::=`

`int8signed | int8unsigned |`  
`int16signed | int16unsigned |`  
`int32 | float32 | float64`

Utile pour la lecture ou l'écriture d'une donnée en mémoire ( $\tau$ ) :

- taille  $|\tau|$  et alignement  $\langle\tau\rangle$  |  $|\tau| = \text{sizeof}(\tau)$      $\langle\tau\rangle = 1$
- garantie de compatibilité entre une écriture et une lecture ultérieure

# Séquences de lectures et écritures compatibles

## Exemple de séquence écriture-lecture

```
union { int i; float f; } u;  
float x;  
...  
u.i = 8;  
x = u.f;  
...
```

Standard C : comportement indéfini

Comportement : l'écriture d'une donnée de type  $\tau$  ne peut être suivie que d'une lecture d'une donnée d'un type  $\tau'$  **compatible** avec  $\tau$  ( $\tau \sim \tau'$ ).

## Axiomes

$\tau \sim \tau$

si  $\tau_1 \sim \tau_2$ , alors  $|\tau_1| = |\tau_2|$  et  $\langle \tau_1 \rangle = \langle \tau_2 \rangle$

# Niveaux d'abstraction

## Modèle abstrait (Coq)

- géographie de la mémoire non définie
- axiomatisation des opérations de gestion de la mémoire
- autres axiomes

## Raffinement 1 (Coq)

- davantage d'axiomes
- lemmes (dérivés)

## Raffinement 2 (Coq)

- Toutes les abstractions sont définies.
- Tous les axiomes sont prouvés.

## Implémentation (Caml)

# Modèle abstrait

Non définis :

- `mem`, `block`, `memtype`, `val`,
- `empty : mem`

## Définition intuitive de la mémoire

- La mémoire est une collection de blocs séparés.
- Chaque bloc se comporte comme un tableau d'octets.
- Une adresse est un couple  $(b, i)$ .

# Opérations de gestion de la mémoire

## Définition abstraite

`alloc` :  $\text{mem} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \text{option}(\text{block} \times \text{mem})$

`free` :  $\text{mem} \times \text{block} \rightarrow \text{option mem}$

`load` :  $\text{memtype} \times \text{mem} \times \text{block} \times \mathbb{Z} \rightarrow \text{option val}$

`store` :  $\text{memtype} \times \text{mem} \times \text{block} \times \mathbb{Z} \times \text{val} \rightarrow \text{option mem}$

Une valeur `option (t)` est soit  $\epsilon$  (échec),  
soit  $\lfloor x \rfloor$  (succès produisant le résultat  $x : t$ ).

### Propriétés de bonne formation des variables

- Si  $\text{alloc}(m, l, h) = \lfloor b, m' \rfloor$  et  $b' \neq b$ , alors  
 $\text{load}(\tau, m', b', i) = \text{load}(\tau, m, b', i)$
- Si  $\text{free}(m, b) = \lfloor m' \rfloor$  et  $b' \neq b$ , alors  
 $\text{load}(\tau, m', b', i) = \text{load}(\tau, m, b', i)$
- Si  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$  et  $\tau \sim \tau'$ , alors  
 $\text{load}(\tau', m', b, i) = \text{convert}(v, \tau')$
- Si  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$  et  
 $b' \neq b \vee i' + |\tau'| \leq i \vee i + |\tau| \leq i'$ , et  
 $\text{load}(\tau', m', b', i') = \text{load}(\tau', m, b', i')$

load n'est pas complètement spécifiée (cf. standard C).

# Validité des blocs en mémoire

## Propriétés abstraites

### Définition intuitive

$$m \models b$$

$b$  est valide dans  $m$  si  $b$  a été alloué dans  $m$  mais pas encore libéré.

### Axiomatisation de la relation de validité

- Si  $\text{alloc}(m, l, h) = [b, m']$ , alors  $\neg(m \models b)$ .
- Si  $\text{alloc}(m, l, h) = [b, m']$ , alors  $m' \models b' \Leftrightarrow b' = b \vee m \models b'$ .
- Si  $\text{store}(\tau, m, b, i, v) = [m']$ , alors  $m' \models b' \Leftrightarrow m \models b'$ .
- Si  $\text{free}(m, b) = [m']$  et  $b' \neq b$ , alors  $m' \models b' \Leftrightarrow m \models b'$ .
- Si  $m \models b$ , alors il existe  $m'$  tel que  $\text{free}(m, b) = [m']$ .

# Bornes des blocs de mémoire

## Propriétés abstraites

### Bornes d'un bloc

$$\mathcal{B}(m, b) = [l, h[$$

### Axiomatisation de la fonction $\mathcal{B}$

- Si  $\text{alloc}(m, l, h) = [b, m']$ , alors  $\mathcal{B}(m', b) = [l, h[$ .
- Si  $\text{alloc}(m, l, h) = [b, m']$  et  $b' \neq b$ , alors  $\mathcal{B}(m', b') = \mathcal{B}(m, b')$ .
- Si  $\text{store}(\tau, m, b, i, v) = [m']$ , alors  $\mathcal{B}(m', b') = \mathcal{B}(m, b')$ .
- Si  $\text{free}(m, b) = [m']$  et  $b' \neq b$ , alors  $\mathcal{B}(m', b') = \mathcal{B}(m, b')$ .

# Accès valides aux blocs de mémoire

## Propriétés abstraites

$$m \models \tau @ b, i$$

$$m \models b \wedge \langle \tau \rangle \text{ divise } i \wedge \mathcal{L}(m, b) \leq i \wedge i + |\tau| \leq \mathcal{H}(m, b)$$

## Axiomatisation de la relation de validité

Si  $m \models \tau @ b, i$  alors il existe  $m'$  tel que  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ .

## Propriétés dérivées

- Si  $\text{alloc}(m, l, h) = \lfloor b, m' \rfloor$  et  $\langle \tau \rangle$  divise  $i$  et  $l \leq i$  et  $i + |\tau| \leq h$ , alors  $m' \models \tau @ b, i$ .
- Si  $\text{alloc}(m, l, h) = \lfloor b, m' \rfloor$  et  $m \models \tau @ b', i$ , alors  $m' \models \tau @ b', i$ .
- Si  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ , alors  $m' \models \tau @ b', i \Leftrightarrow m \models \tau @ b', i$ .
- Si  $\text{free}(m, b) = \lfloor m' \rfloor$  et  $b' \neq b$ , alors  $m' \models \tau @ b', i \Leftrightarrow m \models \tau @ b', i$ .

# Exemple d'utilisation des propriétés

## Preuve de programmes

```
int * x = malloc(2 * sizeof(int));  
int * y = malloc(sizeof(int));  
x[0] = 0;  
x[1] = 1;  
*y = x[0];  
x[0] = x[1];  
x[1] = *y;
```

# Exemple d'utilisation des propriétés

## Preuve de programmes

```
int *x= malloc(2*sizeof(int));
int *y= malloc(sizeof(int));
x[0] = 0;
x[1] = 1;
*y = x[0];
x[0] = x[1];
x[1] = *y;
```

```
(x,m) = alloc(m, 0, 8);
(y,m) = alloc(m, 0, 4);
m = store(int, x, 0, 0);
m = store(int, x, 4, 1);
t = load(int, x, 0);
m = store(int, y, 0, t);
t = load(int, x, 4);
m = store(int, x, 0, t);
t = load(int, y, 0);
m = store(int, x, 4, t);
```

# Exemple d'utilisation des propriétés

## Preuve de programmes

Notation  $P : x \neq y, m \models x, m \models y$ .

```
(x, m) = alloc(m, 0, 8); m  $\models$  x
(y, m) = alloc(m, 0, 4); P
m = store(int, x, 0, 0); P, load(m, x, 0) = [0]
m = store(int, x, 4, 1); P, ... = [0], load(m, x, 4) = [1]
t = load(int, x, 0); P, ... = [0], ... = [1], t = 0
m = store(int, y, 0, t); P, ... = [0], ... = [1], load(m, y, 0) = [0]
t = load(int, x, 4); P, ... = [0], ... = [1], load(m, y, 0) = [0], t = 1
m = store(int, x, 0, t); P, ... = [1], ... = [1], load(m, y, 0) = [0]
t = load(int, y, 0); P, ... = [1], ... = [1], load(m, y, 0) = [0], t = 0
m = store(int, x, 4, t); P, ... = [1], ... = [0], load(m, y, 0) = [0]
```

# Modèle concret

## Géographie de la mémoire

$\text{block} = \mathbb{N}$

### État mémoire

$\text{mem} = (N, B, F, C)$

- $N : \text{block}$  1<sup>er</sup> bloc non encore alloué
- $B : \text{block} \rightarrow \mathbb{Z} \times \mathbb{Z}$  bornes
- $F : \text{block} \rightarrow \text{boolean}$  bloc libéré (true) ou non (false)
- $C : \text{block} \rightarrow \mathbb{Z} \rightarrow \text{option}(\text{metype} \times \text{val})$   
contenu de chaque adresse :  $(b, i)$ 
  - ▶  $\epsilon$  signifie “invalidé”,
  - ▶  $[\tau, v]$  :  $v$  a été écrite à cette adresse, avec un type  $\tau$ .

# Modèle concret

(suite)

Soit  $m = (N, B, F, C)$ .

Validité d'un bloc

$m \models b$

$b < N \wedge F(b) = \text{false}$

Bornes d'un bloc

$B(m, b) = B(b)$

État mémoire initial

$\text{empty}$

$\text{empty} = (0, \lambda b. [0, 0[, \lambda b. \text{false}, \lambda b. \lambda i. \epsilon)$

Allocation

$\text{alloc}(m, l, h)$

si  $\text{can\_allocate}(m, h - l)$  alors  $[b, m']$  sinon  $\epsilon$

avec  $b = N$

et  $m' = (N + 1, B\{b \leftarrow [l, h[ \}, F\{b \leftarrow \text{false}\}, C\{b \leftarrow \lambda i. \epsilon\})$

## Modèle concret

Soit  $m = (N, B, F, C)$ .

### Libération

$\text{free}(m, b)$

si non  $m \models b$  alors  $\epsilon$

sinon  $[N, B\{b \leftarrow [0, 0[]\}, F\{b \leftarrow \text{true}\}, C]$

### Écriture

$\text{store}(\tau, m, b, i, v)$

si non  $m \models \tau @ b, i$  alors  $\epsilon$

sinon  $[N, B, F, C\{b \leftarrow c'\}]$

avec  $c' = C(b)\{i \leftarrow [\tau, v], i + 1 \leftarrow \epsilon, \dots, i + |\tau| - 1 \leftarrow \epsilon\}$

### Lecture

$\text{load}(\tau, m, b, i)$

si non  $m \models \tau @ b, i$  alors  $\epsilon$

sinon si  $C(b)(i) = [\tau', v]$  et  $\tau' \sim \tau$

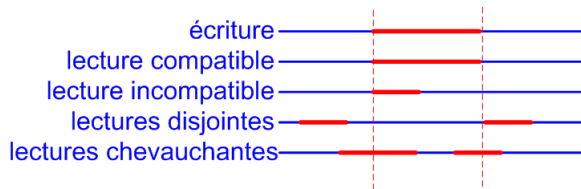
et  $C(b)(i + j) = \epsilon$  pour tout  $j = 1, \dots, |\tau| - 1$

alors  $[\text{convert}(v, \tau)]$  sinon  $[\text{undef}]$

# Propriétés supplémentaires du modèle concret

- $m \models \tau @ b, i \Leftrightarrow \exists m', \text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$
- $m \models \tau @ b, i \Leftrightarrow \exists v, \text{load}(\tau, m, b, i) = \lfloor v \rfloor$
- Si  $\text{alloc}(m, l, h) = \lfloor b, m' \rfloor$  et  $\text{load}(\tau, m', b, i) = \lfloor v \rfloor$ , alors  $v = \text{undef}$ .
- Si  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$  et  $\tau \not\sim \tau'$  et  $\text{load}(\tau', m', b, i) = \lfloor v' \rfloor$ , alors  $v' = \text{undef}$ .
- Si  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$  et  $i' \neq i$  et  $i' + |\tau'| > i$  et  $i + |\tau| > i'$  et  $\text{load}(\tau', m', b, i') = \lfloor v' \rfloor$ , alors  $v' = \text{undef}$ .

# Propriétés supplémentaires du modèle concret



Si  $\text{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$  et  $m \models \tau' @ b', i'$ , alors un seul des 4 cas suivants se produit :

- **Compatible** :  $b' = b$  et  $i' = i$  et  $\tau \sim \tau'$ , auquel cas  $\text{load}(\tau', m', b', i') = \lfloor \text{convert}(v, \tau') \rfloor$ .
- **Incompatible** :  $b' = b$  et  $i' = i$  et  $\tau \not\sim \tau'$ , auquel cas  $\text{load}(\tau', m', b', i') = \lfloor \text{undef} \rfloor$ .
- **Disjoint** :  $b' \neq b$  or  $i' + |\tau'| \leq i$  ou  $i + |\tau| \leq i'$ , auquel cas  $\text{load}(\tau', m', b', i') = \text{load}(\tau', m, b', i')$ .
- **Chevauchement** :  $b' = b$  et  $i' \neq i$  et  $i' + |\tau'| > i$  et  $i + |\tau| > i'$ , auquel cas  $\text{load}(\tau', m', b', i') = \lfloor \text{undef} \rfloor$ .

# Relation de fraîcheur

## Fraîcheur d'un bloc

$m \# b$

$b \geq N$  pour  $m = (N, B, F, C)$

- $m \# b$  et  $m \models b$  sont mutuellement exclusives.
- Si  $\text{alloc}(m, l, h) = [b, m']$ , alors  $m \# b$ .
- Si  $\text{alloc}(m, l, h) = [b, m']$ , alors  $m' \# b' \Leftrightarrow b' \neq b \wedge m \# b'$ .
- Si  $\text{store}(\tau, m, b, i, v) = [m']$ , alors  $m' \# b' \Leftrightarrow m \# b'$ .
- Si  $\text{free}(m, b) = [m']$ , alors  $m' \# b' \Leftrightarrow m \# b'$ .

# Comparaison de mémoires

Domaine

$$\text{Dom}(m_1) = \text{Dom}(m_2)$$

$$\forall b, (m_1 \# b \Leftrightarrow m_2 \# b)$$

- Si  $\text{alloc}(m_1, l, h) = [b_1, m'_1]$  et  $\text{alloc}(m_2, l, h) = [b_2, m'_2]$  et  $\text{Dom}(m_1) = \text{Dom}(m_2)$ , alors  $b_1 = b_2$  et  $\text{Dom}(m'_1) = \text{Dom}(m'_2)$ .
- Si  $\text{free}(m, b) = [m']$ , alors  $\neg(m' \models b)$ .
- Si  $\text{free}(m, b) = [m']$ , alors  $\mathcal{L}(m', b) = \mathcal{H}(m', b)$ .

# Plan

- 1 Vérification formelle de compilateurs
- 2 Modèle mémoire
  - Modèle abstrait
  - Modèle concret
- 3 Transformations de programmes opérant sur la mémoire
- 4 Conclusion

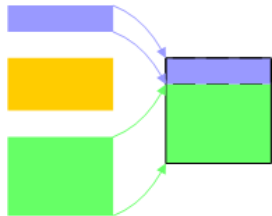
# Transformations de programmes opérant sur la mémoire

La plupart des passes de compilation préservent le comportement de la mémoire.

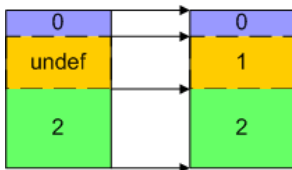
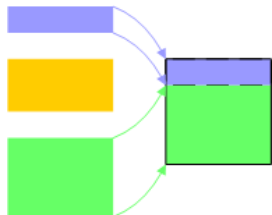
3 passes de compilation transforment la mémoire :

- la 1<sup>re</sup> traduction de Clight à Cminor,
- l'allocation de registres,
- le vidage de registres, effectué après l'allocation de registres.

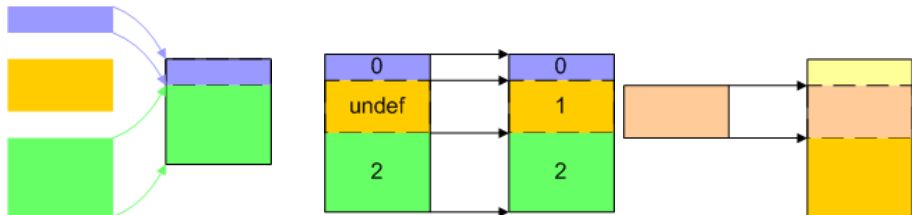
# Transformations de la mémoire



# Transformations de la mémoire



# Transformations de la mémoire



# Première traduction (de Clight à Cminor)

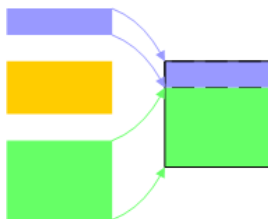
## Injection mémoire

- En Clight, chaque variable réside en mémoire.
- Opérateur de prise d'adresse.
- Sémantique : environnement associant des variables aux blocs de mémoire.

Complice l'allocation de registres et les optimisations du compilateur.

Première traduction :

- Les variables scalaires dont l'adresse n'est jamais prise deviennent des variables locales Cminor, dont la valeur est stockée dans un environnement.
- Les autres variables locales Cminor restent en mémoire mais sont regroupées dans un seul bloc.



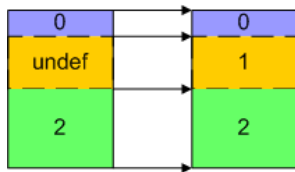
# Allocation de registres

## Raffinement des valeurs stockées en mémoire

- Avant : initialisation à `undef` des variables locales et des temporaires (à chaque appel).
- Après : certaines variables locales et temporaires correspondent à des registres globaux, qui ne sont pas initialisés pareillement.

La sémantique des programmes RTL bien formés ne change pas.

Par contre, le contenu de la mémoire change.

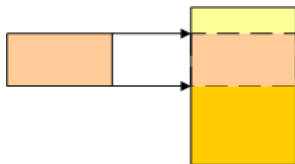


# Vidage de registres en mémoire

## Extension de la mémoire

- Vidage en mémoire : stockage dans la trame de pile courante.
- Dans cette trame, besoin de place supplémentaire afin de stocker les valeurs des registres sauvegardés par l'appelé.

Nécessité d'agrandir les trames de pile, et d'ajouter des instructions de lecture et d'écriture.



# Plongement mémoire

- **Invariants** reliant les états mémoire en tout point de l'exécution du programme initial et du programme transformé.
- Propriétés de simulation entre les opérations mémoire.
- Définition de 3 relations entre états mémoires.

## Plongement de mémoire (E)

$E : \text{block} \rightarrow \text{option}(\text{block} \times \mathbb{Z})$

- $E(b_1) = \epsilon$                        $E(b_1) = [b_2, \delta]$
- Soit  $E \vdash v_1 \hookrightarrow v_2$ , entre les valeurs des 2 programmes.
- $E \vdash m_1 \hookrightarrow m_2$

Toute lecture réussie dans un bloc de  $m_1$  est simulée par une lecture réussie dans le sous-bloc correspondant de  $m_2$  :

$$E(b_1) = [b_2, \delta] \wedge \text{load}(\tau, m_1, b_1, i) = [v_1] \\ \Rightarrow \exists v_2, \text{load}(\tau, m_2, b_2, i + \delta) = [v_2] \wedge E \vdash v_1 \hookrightarrow v_2$$

# Plongement mémoire

- **Invariants** reliant les états mémoire en tout point de l'exécution du programme initial et du programme transformé.
- Propriétés de simulation entre les opérations mémoire.
- Définition de 3 relations entre états mémoires.

## Plongement de mémoire (E)

$E : \text{block} \rightarrow \text{option}(\text{block} \times \mathbb{Z})$

- $E(b_1) = \epsilon$                        $E(b_1) = [b_2, \delta]$
- Soit  $E \vdash v_1 \hookrightarrow v_2$ , entre les valeurs des 2 programmes.
- $E \vdash m_1 \hookrightarrow m_2$

Toute lecture réussie dans un bloc de  $m_1$  est simulée par une lecture réussie dans le sous-bloc correspondant de  $m_2$  :

$$E(b_1) = [b_2, \delta] \wedge \text{load}(\tau, m_1, b_1, i) = [v_1] \\ \Rightarrow \exists v_2, \text{load}(\tau, m_2, b_2, i + \delta) = [v_2] \wedge E \vdash v_1 \hookrightarrow v_2$$

# Propriétés de commutation et de simulation

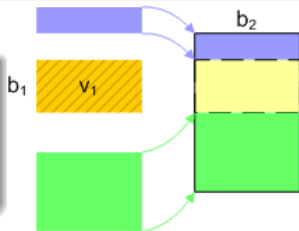
(entre plongements et opérations concrètes de gestion de la mémoire)

## Accès valides

Si  $E(b_1) = [b_2, \delta]$  et  $E \vdash m_1 \hookrightarrow m_2$ , alors  $m_1 \models \tau @ b_1, i$  implique  $m_2 \models \tau @ b_2, i + \delta$ .

## Lemmes de simulation (écritures, cas 1/3)

Si  $E(b_1) = \epsilon$  et  $E \vdash m_1 \hookrightarrow m_2$  et  $\text{store}(\tau, m_1, b_1, i, v) = [m'_1]$ , alors  $E \vdash m'_1 \hookrightarrow m_2$ .



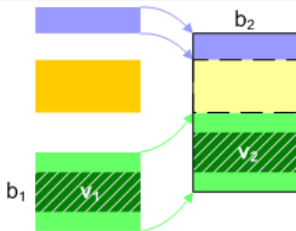
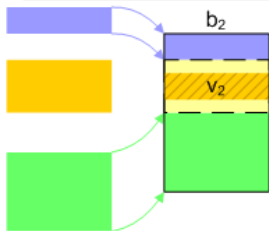
# Propriétés de commutation et de simulation

(entre plongements et écritures)

## Lemmes de simulation (écritures, cas 2 et 3)

- Soient  $b_2, i, \tau$  une référence à la mémoire  $m_2$  telle que  $\forall b_1, \delta$ ,  
 $E(b_1) = [b_2, \delta] \Rightarrow \mathcal{H}(m_1, b_1) + \delta \leq i \vee i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$ .  
Si  $E \vdash m_1 \hookrightarrow m_2$  et  $\text{store}(\tau, m_2, b_2, i, v) = [m'_2]$ , alors  
 $E \vdash m_1 \hookrightarrow m'_2$ .

• ...



# Propriétés de commutation et de simulation

## Plongement sans alias

Des blocs distincts correspondent à des sous-blocs disjoints.

$$b_1 \neq b_2 \wedge E(b_1) = [b'_1, \delta_1] \wedge E(b_2) = [b'_2, \delta_2]$$

$$\Rightarrow b'_1 \neq b'_2$$

$$\vee [\mathcal{L}(m, b_1) + \delta_1, \mathcal{H}(m, b_1) + \delta_1] \cap [\mathcal{L}(m, b_2) + \delta_2, \mathcal{H}(m, b_2) + \delta_2] = \emptyset$$

## Lemme de simulation (cas 3)

Supposons  $E \vdash \text{undef} \leftrightarrow \text{undef}$ ,  $v_1, v_2$  et  $\tau$  tels que  $E \vdash v_1 \leftrightarrow v_2$  et  $\forall \tau', \tau \sim \tau' \Rightarrow E \vdash \text{convert}(v_1, \tau') \leftrightarrow \text{convert}(v_2, \tau')$ .

Si  $E \vdash m_1 \leftrightarrow m_2$  et  $E$  est sans alias dans  $m_1$  et  $E(b_1) = [b_2, \delta]$  et  $\text{store}(\tau, m_1, b_1, i, v_1) = [m'_1]$ , alors il existe  $m'_2$  tel que  $\text{store}(\tau, m_2, b_2, i + \delta, v_2) = [m'_2]$  et de plus  $E \vdash m'_1 \leftrightarrow m'_2$ .

# Propriétés de commutation et de simulation

## Plongement sans alias

Des blocs distincts correspondent à des sous-blocs disjoints.

$$b_1 \neq b_2 \wedge E(b_1) = [b'_1, \delta_1] \wedge E(b_2) = [b'_2, \delta_2]$$

$$\Rightarrow b'_1 \neq b'_2$$

$$\vee [\mathcal{L}(m, b_1) + \delta_1, \mathcal{H}(m, b_1) + \delta_1] \cap [\mathcal{L}(m, b_2) + \delta_2, \mathcal{H}(m, b_2) + \delta_2] = \emptyset$$

## Lemme de simulation (cas 3)

Supposons  $E \vdash \text{undef} \hookrightarrow \text{undef}$ ,  $v_1, v_2$  et  $\tau$  tels que  $E \vdash v_1 \hookrightarrow v_2$  et  $\forall \tau', \tau \sim \tau' \Rightarrow E \vdash \text{convert}(v_1, \tau') \hookrightarrow \text{convert}(v_2, \tau')$ .

Si  $E \vdash m_1 \hookrightarrow m_2$  et  $E$  est sans alias dans  $m_1$  et  $E(b_1) = [b_2, \delta]$  et  $\text{store}(\tau, m_1, b_1, i, v_1) = [m'_1]$ , alors il existe  $m'_2$  tel que  $\text{store}(\tau, m_2, b_2, i + \delta, v_2) = [m'_2]$  et de plus  $E \vdash m'_1 \hookrightarrow m'_2$ .

# Propriétés de commutation et de simulation

(suite)

## Lemmes de simulation (allocations)

- Si  $E \vdash m_1 \hookrightarrow m_2$  et  $\text{alloc}(m_2, l, h) = [b_2, m'_2]$ , alors  $E \vdash m_1 \hookrightarrow m'_2$ .
- Si  $E \vdash m_1 \hookrightarrow m_2$  et  $\text{alloc}(m_1, l, h) = [b_1, m'_1]$  et  $E(b_1) = \epsilon$ , alors  $E \vdash m'_1 \hookrightarrow m_2$ .
- Supposons  $E \vdash \text{undef} \hookrightarrow \text{undef}$ . Si  $E \vdash m_1 \hookrightarrow m_2$  et  $\text{alloc}(m_1, l_1, h_1) = [b_1, m'_1]$  et  $\text{alloc}(m_2, l_2, h_2) = [b_2, m'_2]$  et  $E(b_1) = [b_2, \delta]$  et  $l_2 \leq l_1 + \delta$  et  $h_1 + \delta \leq h_2$  et  $\text{max\_alignment}$  divise  $\delta$ , alors  $E \vdash m'_1 \hookrightarrow m'_2$ .
- ...

# Extension mémoire

(vidage de registres)

Plongement  $E$  instancié à la fonction identité

$$\forall b, E_{id}(b) = [b, 0]$$

Plongement entre valeurs

$$E_{id} \vdash v_1 \hookrightarrow v_2 \text{ ssi } v_1 = v_2$$

$m_1 \subseteq m_2$  :  $m_2$  étend  $m_1$

$$m_1 \subseteq m_2 \stackrel{\text{def}}{=} E_{id} \vdash m_1 \hookrightarrow m_2 \wedge \text{Dom}(m_1) = \text{Dom}(m_2)$$

### Lemmes de simulation

- $\subseteq$  est réflexive et transitive.
- Si  $m_1 \subseteq m_2$  et  $\text{load}(\tau, m_1, b, i) = \lfloor v \rfloor$ , alors  $\text{load}(\tau, m_2, b, i) = \lfloor v \rfloor$ .
- Toute opération d'allocation, écriture ou libération dans  $m_1$  est simulée par une opération similaire dans  $m_2$ , qui préserve la relation d'extension mémoire.
- Le programme transformé peut effectuer des écritures supplémentaires, dans des zones non atteintes par le programme initial.

# Plan

- 1 Vérification formelle de compilateurs
- 2 Modèle mémoire
  - Modèle abstrait
  - Modèle concret
- 3 Transformations de programmes opérant sur la mémoire
- 4 Conclusion

# Conclusion

## Bilan

- Modèle mémoire adapté à la preuve de propriétés de préservation sémantique, concernant des transformations de programmes effectuées par un compilateur.
- Description de quelques violations courantes des programmes C.
- Niveau d'abstraction intermédiaire, garantissant des propriétés de séparation, des tests de bornes.
- Développement en Coq, à l'aide de modules.
- 1070 lignes de spécifications, 970 lignes de preuves.
- Formalisation en logique du premier ordre.
- Automatisation des preuves de la partie abstraite à l'aide de prouveurs automatiques du premier ordre (Ergo, Simplify, Z3) : 42 lemmes sur 50 ont été prouvés.

# Conclusion

## Travaux en cours et futurs

- Logique de séparation (lien avec la preuve de programmes).
- Réutilisation dans un contexte concurrent.
- Raffiner le modèle en un modèle permettant de modéliser d'autres violations courantes du C.
- Définir un modèle à la Burstall-Bornat, étant une abstraction du modèle présenté.
- Analyses statiques estimant la quantité de mémoire nécessaire à l'exécution d'un programme.