Formalisation du modèle polyhédrique: le problème des dépassements arithmétiques

Bruno Cuervo Parrino Julien Narboux Eric Violard
Nicolas Magaud

Université de Strasbourg - INRIA Camus

Décembre 2011, St Hippolyte

Plan

1. Contexte
   - Le modèle polyédrique
   - La compilation certifiée
2. Le problème des dépassements arithmétiques
   - Description du Problème
   - Solution dans Polly
   - Solution proposée
3. Vers une formalisation en Coq

Avertissement

Travail en cours de construction les preuves ne sont pas terminées.

Motivations

- Inutile de prouver un programme source si le compilateur est bugué.
- Les compilateurs sont de plus en plus complexes.
- Les compilateurs parallélisants, spéculatifs, . . . , sont particulièrement complexes.

Projet

Objectif

Intégration du modèle polyédrique dans CompCert (Blazy, Leroy, Tristan)

Afin de:

A moyen terme améliorer la localité des données (spatiale et temporelle)

A plus long terme exhiber du parallélisme

Projet en collaboration avec Alexandre Pilkiewicz - INRIA Gallium.

Modèle polyédrique

Idées principales:

- La plupart des calculs sont réalisés dans des boucles imbriquées.
- Restriction à des problèmes linéaires: on manipule des polyèdres dans $\mathbb{Z}^n$.

Le cadre

Exemple

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
  {
   x[2*i+j]=y[j+1]+f(y[j]);
  }
```

On s'interdit :

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    { x[i*i]=y[j+1]+y[j]; }
```

Et aussi :

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    { x[y[i]]=y[j+1]; }
```

Ou encore :

```
for(i=0;i<n*n;i++)
  for(j=0;j<m;j++)
    { x[i]=y[j+1]; i:=i+3; }
```

Le cadre

Hypothèses

- les accés aux tableaux sont linéaires en les paramètres et les indices de boucles
- pas d'effets de bords
- les tableaux sont disjoints (pas d'alias)

Quelques transformations de boucles

- Pavage de boucles
- Distribution et fusion de boucles
- Déroulage de boucles
- Echange de boucles
- Inversions de boucles

- Les transformation unimodulaires

L'ordre d'execution

```
for(V1, L1, U1)
  for(V2, L2, U2)
    ...
      for(Vn, Ln, Un)
        S1;
        S2;
        ...
        Sn;
```

$$S_{i_I} <_{exe} S_{j_J}$$
ssi
$$I <_{lex} J \vee (I = J \wedge i < j)$$
c'est à dire
$$(I, i) <_{lex} (J, j)$$

L'ordre parallèle

Dépendances

On note $u \, \delta \, v$ le fait que u dépend de v, c'est à dire que u doit être executé après v.

Ordre parallèle

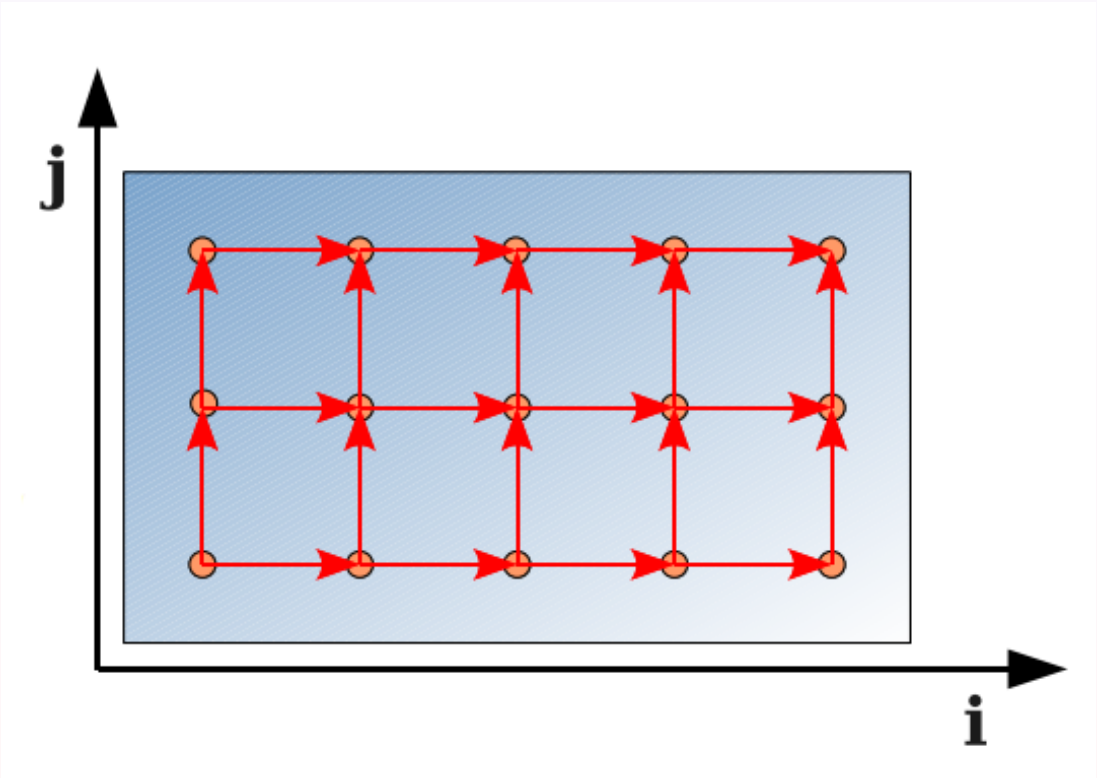
$$<_{//} = (<_{exe} \cap \delta)^+$$

Modèle polyédrique

- Les domaines d'itérations peuvent être représentés par des polyèdres.
- Les dépendances peuvent être représentées par des polyèdres (sous ensembles des produits cartésiens des domaines d'itérations).

Exemple

```
for (i = 0 ; i < 5 ; i++)
  for (j = 0 ; j < 3 ; j++)
    A[i][j] = A[i][j-1] + A[i-1][j] ;
```



Critères de correction

Une transformation de boucle est valide si:

1. l'ensemble des instructions exécutées dans la boucle transformée est égal à l'ensemble d'instructions executées dans la boucle de départ.
2. l'ordre d'exécution des instructions dans la boucle transformée est compatible avec les dépendances.

Correction

1. les matrices unimodulaires représentent une bijection, on obtient alors la première condition
2. pour l'ordre d'execution il faut vérifier que:

$$<_{//} \cap >_{exe_transformee} = \emptyset$$

$\longrightarrow$ utilisation de l'omega test (à l'origine de la tactique omega de Coq).

Implantations

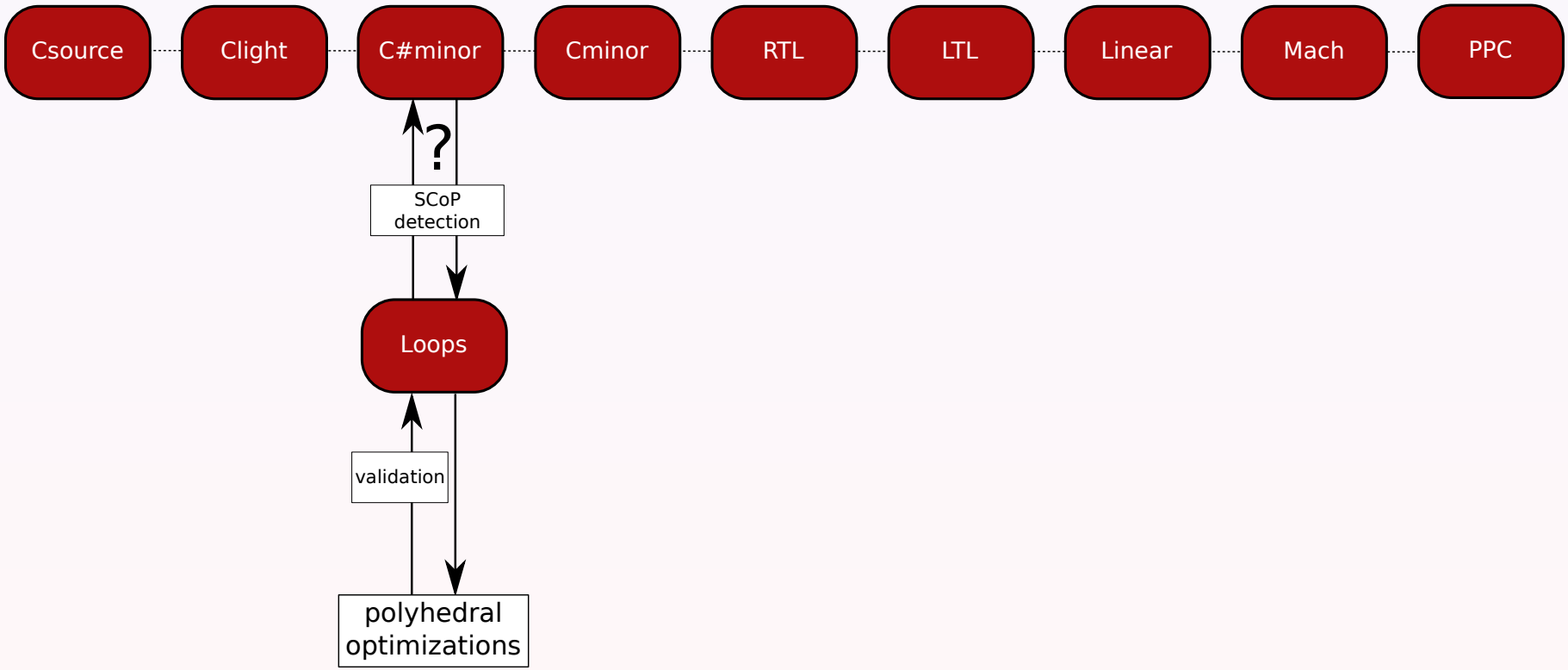
- Dans LLVM: Polly par Tobias Grosser (2010-2011)
- Dans Gcc: Graphite par Konrad Trifunovic et al. (2009-2011)

Projet: CompCert + Loops

Approche à base de validateur *a posteriori* (Zuck, Pnueli at al - 2002) prouvé en Coq (Leroy, Tristan - 2008).

- Cela permet de réutiliser des outils existants.
- Cela suffit pour obtenir la même garantie de correction que dans le cas d'une passe de compilation prouvée formellement.
- Mais ne garantit pas que le compilateur fournira un résultat.

Architecture



Problème

Le modèle polyédrique a l'avantage de fournir une *représentation mathématique simple* pour des morceaux de programmes. L'inconvenient c'est que c'est justement une représentation mathématique éloignée de la représentation machine.

Dépassements arithmétiques

1. Le code original peut produire des dépassements arithmétiques[a].
2. Le code "optimisé" peut produire des dépassements arithmétiques.

[a] Ici seuls importent les dépassements dans les accés aux tableaux et dans le calcul des bornes de boucles

Il faut donc faire le pont entre le monde avec overflows et le monde sans overflows.

The next interesting problem is the selection of the data types for the newly generated induction variables and array subscripts. At the moment Polly uses always 64 bit induction variables and signed calculations. This is correct as long as the input loops use signed calculations for their induction variables (ensured in the frontend), these variables have at most 64 bit size and the schedules of the statements do not produce any larger values. A schedule that may produce larger values is for instance Stmt[i,j] $\longrightarrow$ [i+j]. It may produce an overflow when calculating the bounds of i+j. However, in practical programs we have not yet seen such an overflow as most loops do not get close to the maximal value possible in a 64 bit counter. To ensure correctness even in uncommon pathological cases and to get native types for non 64 bit architectures, we plan to automatically derive the minimal and maximal values the newly generated induction variables may have. For this we plan to add additional information to the polyhedral representation, that defines the maximal values induction variables may reach due to their integer types in the original program. This information is then available after code generation through a polyhedra interface, such that we can conveniently derive the minimal type, needed to ensure that no unforeseen integer overflows occur.

Tobias Grosser

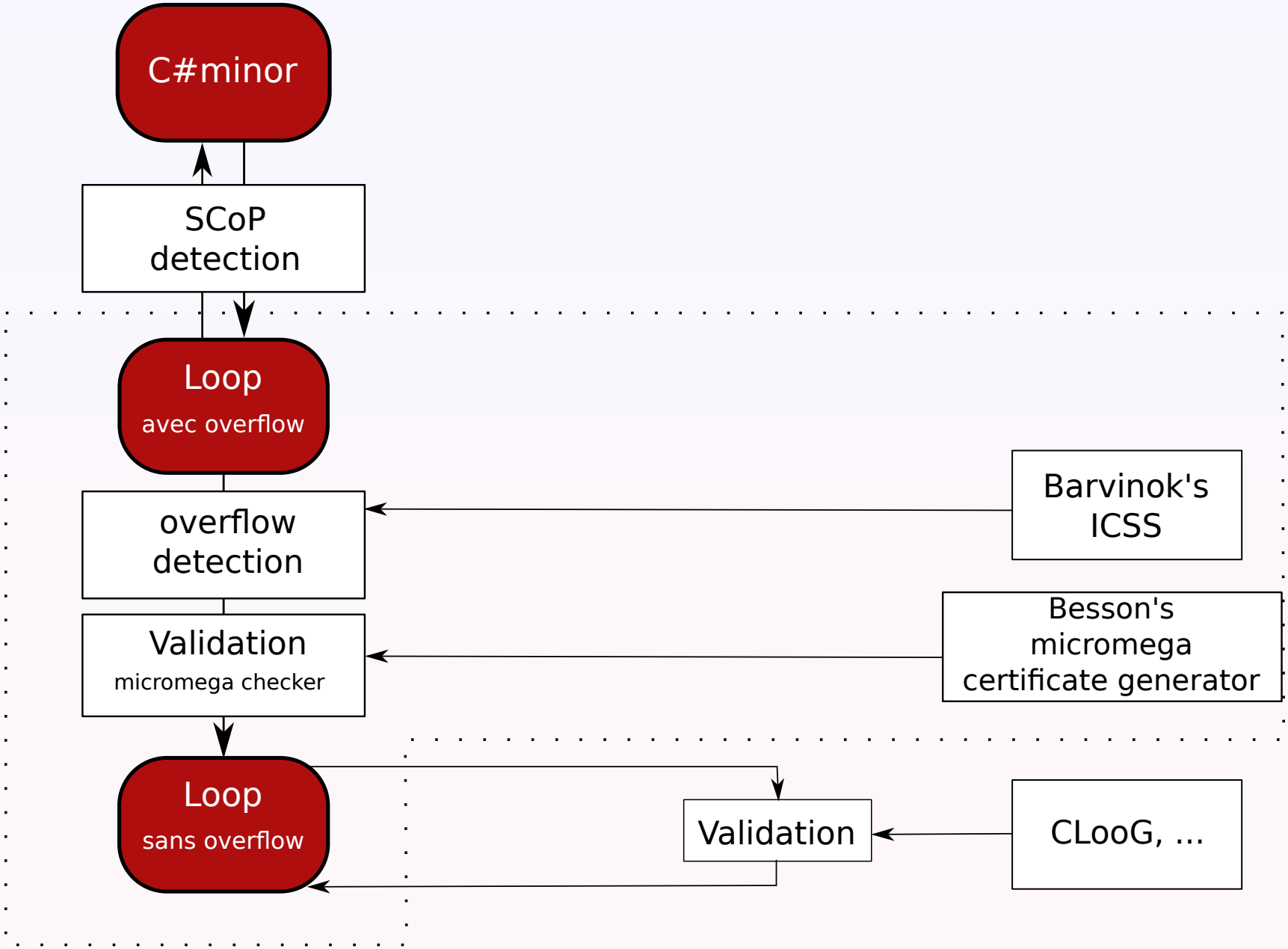
Solution dans Polly

1. Tester si les variables de boucle utilisent uniquement des types signés: en C la sémantique des overflow n'est pas définie, donc pas de problème. Ignorer les nids de boucles qui comportent des variables de boucle non signées.
2. Trouver un type *assez large* pour représenter les variables de boucles dans le programme optimisé.

Solution proposée

Il est impossible de faire une analyse statique, car nos nids de boucles dépendent de paramètres.

1. Trouver une condition suffisante sur les paramètres pour qu'il n'y ait pas d'overflow ni dans le programme de départ ni dans le programme optimisé.
2. Tester dynamiquement cette condition.


C#minor
SCoP detection
Loop avec overflow
overflow detection
Validation micromega checker
Loop sans overflow
Barvinok's ICSS
Besson's micromega certificate generator
Validation
CLooG, ...

Langage

Syntaxe :

$$
\begin{array}{lcl}
l & ::= & n \mid x \mid n * l \mid l + l \mid l - l \\
e & ::= & n \mid x \mid T[l] \mid e + e \mid e - e \mid e * e \mid e/e \\
i & ::= & \text{skip} \mid T[l] := e \mid i; i \mid \text{Loop } x \text{ from } 0 \text{ to } l \text{ do } i \text{ done} \\
b & ::= & \text{true} \mid \text{false} \mid l \leq l \mid b \text{ and } b \mid b \text{ or } b \\
j & ::= & \text{If } b \text{ then } i \text{ else } i
\end{array}
$$

Expressions arithmétiques (e) en Coq :

```
Inductive aexpr: Type :=
| Anum : int32 -> aexpr
| Avar : positive -> aexpr
| Aarray : array -> lexpr -> aexpr
| Aplus : aexpr -> aexpr -> aexpr
| Aminus : aexpr -> aexpr -> aexpr
| Amult : aexpr -> aexpr -> aexpr
| Adiv : aexpr -> aexpr -> aexpr.
```

Langage

```
Inductive instr: Type :=
| Skip : instr
| Assign : array -> lexpr -> aexpr -> instr
| Seq : instr -> instr -> instr
| Loop : positive -> lexpr -> instr -> instr
| Loop' : positive -> int32 -> instr -> instr.

Inductive instrgen : Type :=
| Instr : instr -> instrgen
| If : bexpr -> instr -> instr -> instrgen.
```

Condition d'absence d'overflow

On définit une condition suffisante `cond_overflow`, par induction sur la structure des programmes:

Skip

$$True$$

t(l):=e

$$\bigwedge_{expr \in subexpr(l) \cup lsubexpr(e)} min \leq expr \leq max$$

$i_1; i_2$

$$cond_overflow(i_1) \wedge cond_overflow(i_2)$$

$Loop(x, l)\ i$

$$0 \leq x \leq l \Rightarrow cond_overflow(i)$$

Exemple

```
for (i=0;i<=N;i++)
 {
  T[i+1]=55;
  for (j=0;j<=M;j++)
    T[i+j]=12;
  for (j=0;j<=i;j++)
    T[i+2*j] = 42;
 }
```

Exemple

http://www.cs.kuleuven.be/cgi-bin/dtai/barvinok.cgi

Requête

```
{[n,m]} - {[n,m] | exists i: exists j:
(0 <= i <= n /\ (-32768 > i+1 \/ i+1 > 32767))
\/ (0 <= i <= n /\ 0 <= j <= m /\ (-32768 > i+j \/
                                   i+j > 32767))
\/ (0 <= i <= n /\ 0 <= j <= i /\ (-32768 > 2*j \/
            2*j > 32767 \/ -32768 > i+2*j \/ i+2*j > 32767))
};
```

Réponse

```
{ [n, m] : n <= -1 or (n <= 10922 and
                       n >= 0 and
                       m <= 32767 - n) }
```

Nos hypothèses

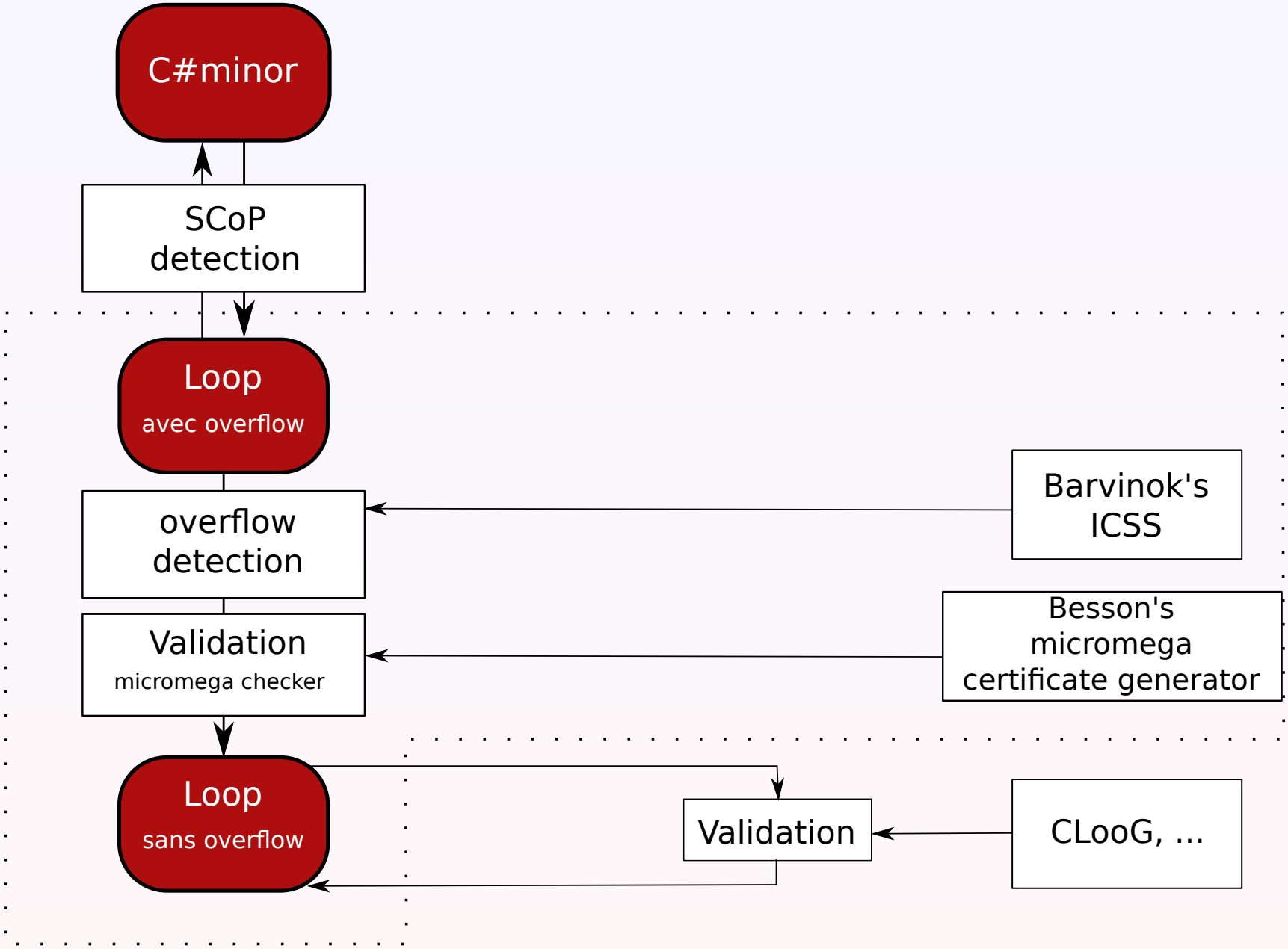
Nous supposons que l'on a:

- une passe d'optimisation utilisant le modèle polyédrique et
- la preuve que cette passe est correcte dans un monde sans overflows (sur les indices de boucles).

En Coq

```
Definition polyhedral_optim (p: instr) := ...

Axiom polyhedral_optim_correct :
 forall p p', polyhedral_optim p = Some p'->
 loop.equiv NO p p'.
```


C#minor
SCoP detection
Loop avec overflow
overflow detection
Validation micromega checker
Loop sans overflow
Barvinok's ICSS
Besson's micromega certificate generator
Validation
CLooG, ...

Compilateur

```
(* Implemented in ml using iscc *)
Parameter oracle : instr -> instr -> bexpr.

Definition compile p :=
  match (polyhedral_optim p) with
  None => None
| Some opt =>
   let new := If (oracle p opt) opt p in
    if validator p opt new then Some new else None
end.
```

Validateur

Entrée:
- org le programme original
- opt le programme envoyé par l'optimiseur polyédrique
- new le programme qu'on a produit qui execute soit org soit opt suivant les overflows.

Sortie: si on renvoie `true` c'est que le nouveau programme est équivalent à l'original, sinon on ne sait pas.

Algo: On regarde le programme, s'il n'a pas la forme: `If b org opt` alors on renvoie `false`, sinon on teste si b implique bien (cond_overflow org) et (cond_overflow opt).

Validateur

Entrée:
- org le programme original
- opt le programme envoyé par l'optimiseur polyédrique
- new le programme qu'on a produit qui execute soit org soit opt suivant les overflows.

Sortie: si on renvoie `true` c'est que le nouveau programme est équivalent à l'original, sinon on ne sait pas.

Algo: On regarde le programme, s'il n'a pas la forme: `If b org opt` alors on renvoie `false`, sinon on teste si b implique bien (cond_overflow org) et (cond_overflow opt).
De plus on vérifie que b ne fait apparaître que des paramètres.

Validateur en Coq

```
Definition validator (org:instr) (opt:instr) (new:instrgen):=
 match new with
  If b p q =>
   instr_dec p opt && (* p=opt *)
   instr_dec q org && (* q=org *)
   disjoint_lists (loopvarlist p) (bvars b) &&
   disjoint_lists (loopvarlist q) (bvars b) &&
      (* b contient que des paramètres *)
   overflow_checker p b && (* b => pas d'overflow dans p *)
   overflow_checker q b    (* b => pas d'overflow dans q *)
| Instr st => false
 end.
```

Une propriété à prouver

```
Lemma validator_correct :
  forall p p' new,
   loop.equiv NO p p' ->
   validator p p' new = true ->
   equiv_gen 0 new (Instr p).
```

Problème

L'évaluation de la condition d'absence d'overflow elle-même peut produire un overflow...

Pistes:

1. Trouver une condition suffisante pour que l'évaluation de la condition d'absence d'overflow ne produise pas d'overflow.
   - Problème: L'évaluation de la condition d'absence d'overflow elle-même peut produire un overflow...
     1. Trouver une condition suffisante pour que l'évaluation de la condition d'absence d'overflow ne produise pas d'overflow.

     ...
2. Trouver une condition suffisante dont on peut prouver que l'évaluation ne peut pas produire d'overflow.
3. Evaluer la condition d'absence d'overflow en précision arbitraire/suffisante.
4. Tester si l'évaluation de b produit un overflow.

Pistes:

1. Trouver une condition suffisante pour que l'évaluation de la condition d'absence d'overflow ne produise pas d'overflow.
   - Problème: L'évaluation de la condition d'absence d'overflow elle-même peut produire un overflow...
     1. Trouver une condition suffisante pour que l'évaluation de la condition d'absence d'overflow ne produise pas d'overflow.

     ...
2. Trouver une condition suffisante dont on peut prouver que l'évaluation ne peut pas produire d'overflow.
3. Evaluer la condition d'absence d'overflow en précision arbitraire/suffisante.
4. Tester si l'évaluation de b produit un overflow.

Tester si l'évaluation de b produit un overflow.

Une approche pragmatique

- On modifie la sémantique d'évaluation des expressions linéaires

$$\mathcal{L} \vdash_{oc} l \longrightarrow (n, e)$$

n : valeur calculée pour l
e : booléen = *true* si l'évaluation de *l* se fait sans overflow,
false sinon.

- Les tests ($\leq$) sur les expr. linéaires produisent *false* en cas d'overflow.
- Si l'évaluation de b produit un overflow,
  on exécute le programme initial.

Conclusion/Travaux futurs

Conclusion

- 3000 lignes de Coq, 300 lignes de Ocaml.
- Réutilisation de la tactique Coq `micromega` assez facile.

Overflows

- Finir les preuves.
- Etude du passage à l'échelle.
- Porter les preuves vers un vrai langage (CompCert).

Modèle polyédrique

- Preuve de la détection de SCoP.

Merci.