

RAIM'09 – Lyon

Preuves de programmes numériques indépendantes du compilateur et du processeur

Sylvie Boldo, Thi Minh Tuyen Nguyen

INRIA – Équipe-projet ProVal

28 octobre 2009

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE



centre de recherche
SACLAY - ÎLE-DE-FRANCE

Un petit programme...

```
int main(){
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64; // y = 2-53 + 2-64
    double z = x + y;
    printf("z=%a\n", z);
}
```

Un petit programme...

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64; // y = 2-53 + 2-64
  double z = x + y;
  printf("z=%a\n", z);
}
```

⇒ $z = 1 + 2^{-52}$ en IEEE-754 strict

Un petit programme. . .

```
int main(){  
    double x = 1.0;  
    double y = 0x1p-53 + 0x1p-64; // y = 2-53 + 2-64  
    double z = x + y;  
    printf("z=%a\n", z);  
}
```

⇒ $z = 1 + 2^{-52}$ en IEEE-754 strict

⇒ $z = 1$ en x87 avec registres étendus

Un petit programme. . .

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64; // y = 2-53 + 2-64
  double z = x + y;
  printf("z=%a\n", z);
}
```

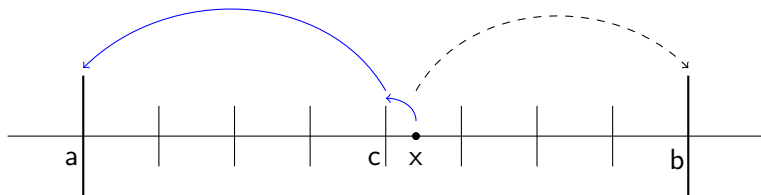
⇒ $z = 1 + 2^{-52}$ en IEEE-754 strict

⇒ $z = 1$ en x87 avec registres étendus

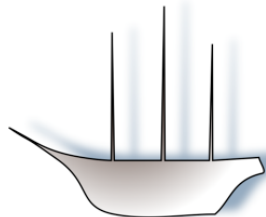
Avec les approches précédentes, on peut prouver formellement que $z > 1$.

Double arrondi

Au lieu d'avoir $\circ_{64}(x) = b$, on a $\circ_{64}(\circ_{80}(x)) = \circ_{64}(c) = a$.



Hisseo est un projet Digiteo qui a pour but de s'attaquer aux problèmes liés aux flottants/architectures multiples/compilation.



- P. Cuoq (CEA LIST)
- A. Ayad (post-doc Hisseo/DGA)
- S. Boldo (INRIA Saclay)
- X. Leroy (INRIA Paris - Rocquencourt)
- C. Marché (INRIA Saclay)
- G. Melquiond (INRIA Saclay)
- T. Nguyen (INRIA Saclay)

<http://hisseo.saclay.inria.fr/>

Plan

1 Motivations

2 Erreurs d'arrondis

3 Preuves de programmes

4 Exemple

Erreurs d'arrondis

On ne considère que les erreurs d'arrondis.

Erreurs d'arrondis

On ne considère que les **erreurs d'arrondis**.

⇒ doivent être valides quelle que soit l'architecture

- arrondi usuel 64 bits
- arrondi étendu 80 bits
- double arrondi 80 puis 64 bits

Erreurs d'arrondis

On ne considère que les **erreurs d'arrondis**.

⇒ doivent être valides quelle que soit l'architecture

- arrondi usuel 64 bits
- arrondi étendu 80 bits
- double arrondi 80 puis 64 bits

(Pour le FMA, affaire à suivre)

Erreurs suivant le type d'arrondi

0

$+\infty$



Erreurs suivant le type d'arrondi

64

$$|\circ(x) - x| \leq 2^{-1075}$$

$$\left| \frac{\circ(x) - x}{x} \right| \leq 2^{-53}$$

0

2^{-1022}

$+\infty$

Erreurs suivant le type d'arrondi

64

$$|\circ(x) - x| \leq 2^{-1075}$$

0

2^{-16382}

2^{-1022}

$+\infty$

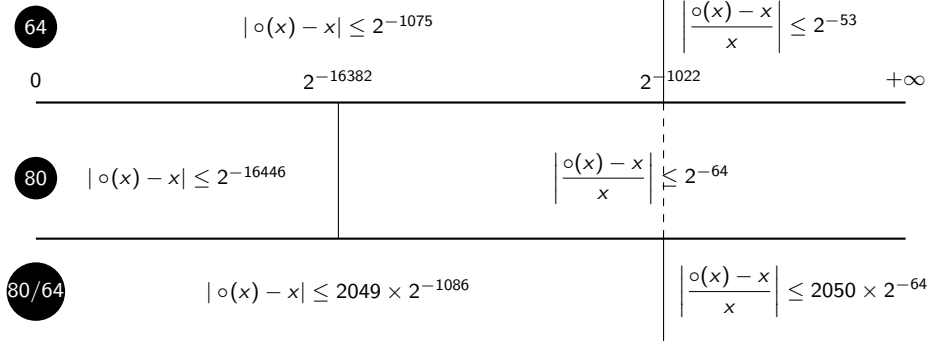
$$\left| \frac{\circ(x) - x}{x} \right| \leq 2^{-53}$$

80

$$|\circ(x) - x| \leq 2^{-16446}$$

$$\left| \frac{\circ(x) - x}{x} \right| \leq 2^{-64}$$

Erreurs suivant le type d'arrondi



Erreurs d'arrondis

Théorème

Soit un nombre réel x , soit $\square(x)$ défini par $\circ_{64}(x)$, ou $\circ_{80}(x)$, ou le double arrondi $\circ_{64}(\circ_{80}(x))$. Alors,

Erreurs d'arrondis

Théorème

Soit un nombre réel x , soit $\square(x)$ défini par $\circ_{64}(x)$, ou $\circ_{80}(x)$, ou le double arrondi $\circ_{64}(\circ_{80}(x))$. Alors,

$$\text{Si } |x| \geq 2^{-1022} \text{ alors } \left(\left| \frac{x - \square(x)}{x} \right| \leq 2050 \times 2^{-64} \text{ et } |\square(x)| \geq 2^{-1022} \right)$$

$$\text{Si } |x| \leq 2^{-1022} \text{ alors } (|x - \square(x)| \leq 2049 \times 2^{-1086} \text{ et } |\square(x)| \leq 2^{-1022})$$

Erreurs d'arrondis

Théorème

Soit un nombre réel x , soit $\square(x)$ défini par $\circ_{64}(x)$, ou $\circ_{80}(x)$, ou le double arrondi $\circ_{64}(\circ_{80}(x))$. Alors,

$$\text{Si } |x| \geq 2^{-1022} \text{ alors } \left(\left| \frac{x - \square(x)}{x} \right| \leq 2050 \times 2^{-64} \text{ et } |\square(x)| \geq 2^{-1022} \right)$$

$$\text{Si } |x| \leq 2^{-1022} \text{ alors } (|x - \square(x)| \leq 2049 \times 2^{-1086} \text{ et } |\square(x)| \leq 2^{-1022})$$

(prouvé en Coq)

Plan

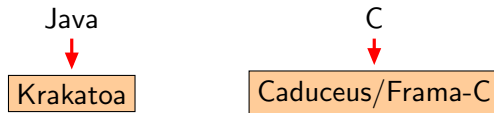
- 1 Motivations
- 2 Erreurs d'arrondis
- 3 Preuves de programmes
- 4 Exemple

Frama-C et la plate-forme Why

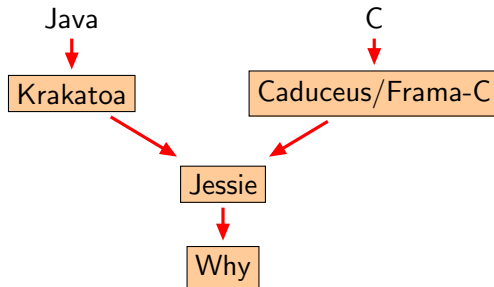
Java

C

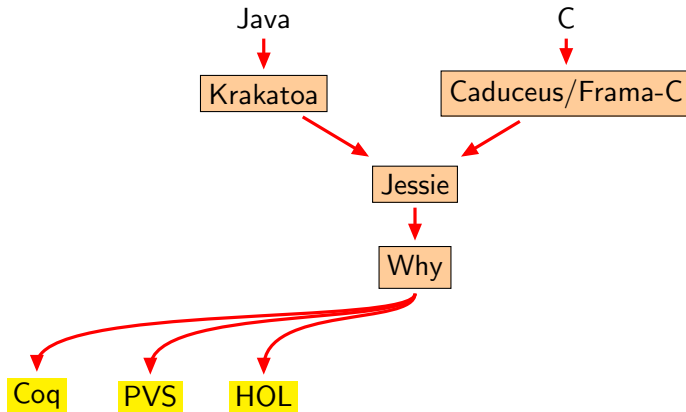
Frama-C et la plate-forme Why



Frama-C et la plate-forme Why

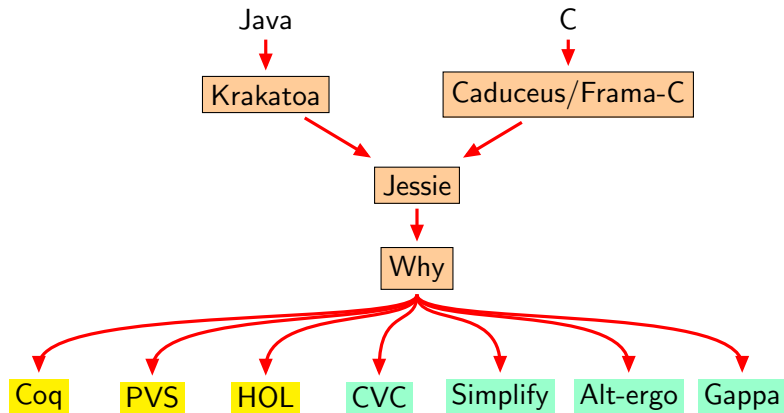


Frama-C et la plate-forme Why



Obligations de preuve

Frama-C et la plate-forme Why



Obligations de preuve

Jessie et les formules ?

Jessie est modulable (pour les développeurs) :

Jessie et les formules ?

Jessie est modulable (pour les développeurs) :

- on a modifié les post-conditions des opérations flottantes
 - ▶ addition, soustraction, multiplication, division, racine carrée

Jessie et les formules ?

Jessie est modulable (pour les développeurs) :

- on a modifié les post-conditions des opérations flottantes
 - ▶ addition, soustraction, multiplication, division, racine carrée
 - ▶ négation, valeur absolue

Jessie et les formules ?

Jessie est modulable (pour les développeurs) :

- on a modifié les post-conditions des opérations flottantes
 - ▶ addition, soustraction, multiplication, division, racine carrée
 - ▶ négation, valeur absolue
- et on a mis uniquement les formules du théorème 1 à la place.

Jessie et les formules ?

Jessie est modulable (pour les développeurs) :

- on a modifié les post-conditions des opérations flottantes
 - ▶ addition, soustraction, multiplication, division, racine carrée
 - ▶ négation, valeur absolue
- et on a mis uniquement les formules du théorème 1 à la place.

⇒ On peut **annoter et prouver des programmes** que les arrondis soient IEEE-754 strict ou avec des registres étendus.

Et le FMA ? (Fused Multiply-and-Add : $\circ(a \times b + c)$)

Et le FMA ? (Fused Multiply-and-Add : $\circ(a \times b + c)$)

Eh bien, on l'a gratuitement. . .

Et le FMA ? (Fused Multiply-and-Add : $\circ(a \times b + c)$)

Eh bien, on l'a gratuitement. . .

On considère que l'on peut avoir $\square(x) = x$.

- Un FMA est alors $\square(\square(a \times x) + b)$ avec le \square intérieur qui est l'« arrondi » identité.

Et le FMA ? (Fused Multiply-and-Add : $\circ(a \times b + c)$)

Eh bien, on l'a gratuitement. . .

On considère que l'on peut avoir $\square(x) = x$.

- Un FMA est alors $\square(\square(a \times x) + b)$ avec le \square intérieur qui est l'« arrondi » identité.
- Les formules précédentes sont évidemment encore valides.

Et le FMA ? (Fused Multiply-and-Add : $\circ(a \times b + c)$)

Eh bien, on l'a gratuitement. . .

On considère que l'on peut avoir $\square(x) = x$.

- Un FMA est alors $\square(\square(a \times x) + b)$ avec le \square intérieur qui est l'« arrondi » identité.
- Les formules précédentes sont évidemment encore valides.

⇒ On considère **tous les cas possibles d'utilisation** du FMA.

Théorème

Si nous considérons les formules du théorème 1 sur chaque opération (addition, soustraction, multiplication, division, racine carrée, négation, valeur absolue), l'erreur d'arrondi finale est correcte quelle que soit l'architecture et la compilation, à condition que le compilateur respecte le parenthésage des opérations.

Plan

- 1 Motivations
- 2 Erreurs d'arrondis
- 3 Preuves de programmes
- 4 Exemple**

- algorithme de détection et de résolution de conflits aériens

- algorithme de détection et de résolution de conflits aériens
- prouvé correct avec des calculs exacts (C. Muñoz, G. Dowek. . .)

- algorithme de détection et de résolution de conflits aériens
- prouvé correct avec des calculs exacts (C. Muñoz, G. Dowek. . .)
- On regarde un petit bout de code qui décide si on va à droite ou à gauche (selon la vitesse et la position).

- algorithme de détection et de résolution de conflits aériens
- prouvé correct avec des calculs exacts (C. Muñoz, G. Dowek. . .)
- On regarde un petit bout de code qui décide si on va à droite ou à gauche (selon la vitesse et la position).
- `return sign(sx*vx+sy*vy)*sign(sx*vy-sy*vx) ;`

- algorithme de détection et de résolution de conflits aériens
- prouvé correct avec des calculs exacts (C. Muñoz, G. Dowek. . .)
- On regarde un petit bout de code qui décide si on va à droite ou à gauche (selon la vitesse et la position).
- `return sign(sx*vx+sy*vy)*sign(sx*vy-sy*vx) ;`

⇒ calculs flottants peu sûrs.

```
#pragma JessieFloatModel(multirounding)
#pragma JessieIntegerModel(math)
```

```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;
```

```
/*@ requires e1 <= x-\exact(x) <= e2;
```

```
@ ensures \abs(\result) <= 1 &&
```

```
@ (\result != 0 ==> \result == l_sign(\exact(x)));
```

```
@*/
```

```
int sign(double x, double e1, double e2) {
```

```
  if (x > e2)
```

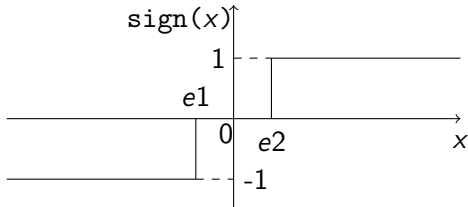
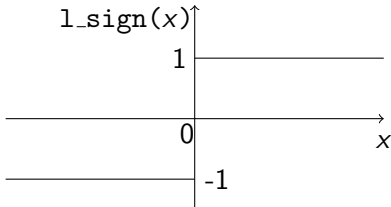
```
    return 1;
```

```
  if (x < e1)
```

```
    return -1;
```

```
  return 0;
```

```
}
```



Fonction eps_line

```
/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @   \result != 0 ==>
  @     \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy)
  @       * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx)));
  @*/
```

```
int eps_line(double sx, double sy, double vx, double vy){
  int s1, s2;

  s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);

  return s1*s2;
}
```

Fonction eps_line

- On prouve que si le résultat est non nul, alors c'est le bon.
Le programme ne ment pas.

Fonction eps_line

- On prouve que si le résultat est non nul, alors c'est le bon.
Le programme ne ment pas.
- On exige que $|s_{x/y}| \leq 100$ et $|v_{x/y}| \leq 1$.
- On prouve que l'erreur de calcul de $sx * vx + sy * vy$ est inférieure à $0 \times 1.90641p - 45$

Fonction eps_line

- On prouve que si le résultat est non nul, alors c'est le bon.
Le programme ne ment pas.
- On exige que $|s_{x/y}| \leq 100$ et $|v_{x/y}| \leq 1$.
- On prouve que l'erreur de calcul de $sx * vx + sy * vy$ est inférieure à $0x1.90641p - 45$
- quels que soient l'architecture et les choix du compilateurs !

Fonction eps_line

- On prouve que si le résultat est non nul, alors c'est le bon.
Le programme ne ment pas.
- On exige que $|s_{x/y}| \leq 100$ et $|v_{x/y}| \leq 1$.
- On prouve que l'erreur de calcul de $sx * vx + sy * vy$ est inférieure à $0x1.90641p - 45$
- quels que soient l'architecture et les choix du compilateurs !
- complètement **automatiquement** !

Proof obligations	Alt-Ergo 0.9	CVC3 20090710 (SS)	Gappa 0.12.0	Statistics
Function eps_line Default behavior	✗	✓	✗	1/1
1. postcondition	✂	✓	◊	
Function eps_line Safety	✗	✗	✓	13/13
1. check FP overflow	✂	✂	✓	
2. check FP overflow	✂	✂	✓	
3. check FP overflow	✂	✂	✓	
4. check FP overflow	✓	✓	✓	
5. check FP overflow	✂	✓	✓	
6. precondition for user call	✂	✂	✓	
7. precondition for user call	✂	✂	✓	
8. check FP overflow	✂	✂	✓	
9. check FP overflow	✂	✂	✓	
10. check FP overflow	✂	✂	✓	
11. check FP overflow	✓	✓	✓	
12. precondition for user call	✂	✂	✓	
13. precondition for user call	✂	✂	✓	
Function sign Default behavior	✗	✗	✗	6/6
1. postcondition	✓	✓	◊	
2. postcondition	✓	✂	◊	
3. postcondition	✓	✂	◊	
4. postcondition	✂	✂	✓	
5. postcondition	✓	✓	✓	
6. postcondition	✂	✂	✓	

```

sx: gen_float
sy: gen_float
vx: gen_float
vy: gen_float
H1: (float_value(sx) = exact_value(sx) and
float_value(sy) = exact_value(sy) and
float_value(vx) = exact_value(vx) and
float_value(vy) = exact_value(vy) and
abs_real(float_value(sx)) <= 100.0 and
abs_real(float_value(sy)) <= 100.0 and
abs_real(float_value(vx)) <= 1.0 and
abs_real(float_value(vy)) <= 1.0)
H6: no_overflow(Double, nearest_even, float_value(sx) * float_value(vx))
result: gen_float
H7: mul_gen_float_post(Double, nearest_even, sx, vx, result)
H8: no_overflow(Double, nearest_even, float_value(sy) * float_value(vy))
result0: gen_float
H9: mul_gen_float_post(Double, nearest_even, sy, vy, result0)
H10: no_overflow(Double, nearest_even,
float_value(result) + float_value(result0))
result1: gen_float
H11: add_gen_float_post(Double, nearest_even, result, result0, result1)
H12: no_overflow(Double, nearest_even, 0x1.90641p-45)
result2: gen_float
H13: gen_float_of_real_post(Double, nearest_even, 0x1.90641p-45, result2)
H14: no_overflow(Double, nearest_even, -float_value(result2))
result3: gen_float
H15: neg_gen_float_post(Double, nearest_even, result2, result3)
H16: no_overflow(Double, nearest_even, 0x1.90641p-45)
result4: gen_float
H17: gen_float_of_real_post(Double, nearest_even, 0x1.90641p-45, result4)

float_value(result3) <= float_value(result1) - exact_value(result1)

```

```

/*@ requires
@  sx == \exact(sx) && sy == \exact(sy) &&
@  vx == \exact(vx) && vy == \exact(vy) &&
@  \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
@  \abs(vx) <= 1.0 && \abs(vy) <= 1.0;
@ ensures
@  \result != 0
@  ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
@  * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
@*/

int eps_line(double sx, double sy, double vx, double vy){
int s1,s2;

s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);

```

Conclusion – perspectives

- ⊕ Malgré les incohérences liées aux architectures, on peut prouver des programmes quel que soit le matériel et la compilation (qui ne réorganise pas).

Conclusion – perspectives

- ⊕ Malgré les incohérences liées aux architectures, on peut prouver des programmes quel que soit le matériel et la compilation (qui ne réorganise pas).
- ⊖ On a des problèmes d'efficacité car, pour chaque calcul, on a 2 cas : normalisé ou dénormalisé.

Conclusion – perspectives

- ⊕ Malgré les incohérences liées aux architectures, on peut prouver des programmes quel que soit le matériel et la compilation (qui ne réorganise pas).
- ⊖ On a des problèmes d'efficacité car, pour chaque calcul, on a 2 cas : normalisé ou dénormalisé.
 - gérer de façon identique les résultats normalisé et dénormalisé ?
 - D'autres propriétés que les erreurs d'arrondi ? (calculs exacts)
 - Et si le compilateur réorganise les calculs ?