Elimination des racines et divisions pour du code embarqué

Pierre Neron

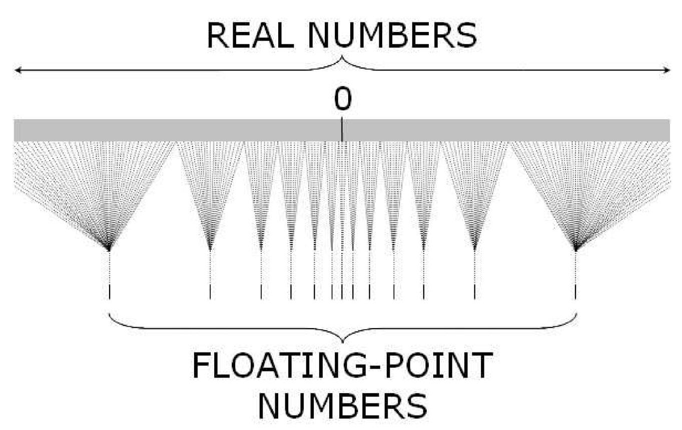
Elimination des racines et divisions pour du code embarqué

Pierre NERON

Inria / École polytechnique

1 . Contexte

- Code embarqué critique :
  - système ACCoRD pour l'aéronautique (NASA Langley)
  - opérateur conditionnel (if then else)
  - programmes sans boucles
  - pas d'allocation dynamique de mémoire
- Arithmétique réelle $+,\ -,\ \times,\ /,\ \sqrt{\ }$
- Représentation finie des réels en informatique :



$$\Longrightarrow \sqrt{2} \times \sqrt{2} > 2$$

- $\sqrt{\ }$ et $/$ créent des suites infinies :

  $\sqrt{2} = 1.41421356237...$ $\qquad 1/7 = 0.14285714285...$
- Arithmétique exacte avec $+, -, \times$:
  - entiers dynamiques
  - taille max par analyse statique

2 . Spécification

Étant donné qu'il est possible de calculer exactement avec $+, -, \times$, le but est de définir une **transformation de programmes** qui :

- élimine les racines et les divisions
- préserve la sémantique lorsqu'il n'y a pas d'échec

Si on ne peut pas toujours éliminer ces opérations (le programme $\sqrt{2}$ retournera toujours une valeur approchée), on peut en revanche les éliminer de valeurs booléennes et ainsi protéger le graphe de contrôle du programme des erreurs d'arrondis.

3 . Langage

Prog :=	Constant	| Var
	| fst Prog	| snd Prog
	| uop Prog	| Prog op Prog
	| (Prog, Prog)	| let Var = Prog in Prog
	| if Prog then Prog else Prog	

avec : $\quad$ Constant $\subset \mathbb{R} \cup \{True,\ False\}$

$\quad$ op $\in \{+, \times, /, =, \neq, >, \geq, <, \leq, \wedge, \vee\}$

$\quad$ uop $\in \{\sqrt{\ }, -, \neg\}$

4 . Expressions booléennes

Soit $E_1\ \mathcal{R}\ E_2$ une comparaison, on élimine racines et divisions en appliquant les transformation suivantes qui définissent la fonction `elim_bool` :

- Mettre les divisions en tête :

$$E_1\ \mathcal{R}\ E_2 \quad \longrightarrow \quad \frac{A}{B}\ \mathcal{R}\ \frac{C}{D}$$

- Eliminer les divisions de tête :

$$\frac{A}{B}\ \mathcal{R}\ \frac{C}{D} \quad \longrightarrow \quad A.B.D^2\ \mathcal{R}\ C.D.B^2$$

- Choisir une racine et factoriser :

$$A.B.D^2\ \mathcal{R}\ C.D.B^2 \quad \longrightarrow \quad P.\sqrt{Q} + R\ \mathcal{R}\ 0$$

- Eliminer la racine choisie : $P.\sqrt{Q} + R\ \mathcal{R}\ 0 \quad \longrightarrow$

$$(P\ \mathcal{R}\ 0 \wedge R\ \mathcal{R}\ 0) \vee (P \geq 0 \wedge P^2.Q - R^2\ \mathcal{R}\ 0) \vee (R \geq 0 \wedge 0\ \mathcal{R}\ P^2.Q - R^2)$$

- Tant qu'il y a des racines, recommencer

5 . Définitions de variables

Afin d'éviter que ces expressions booléenes ne dependent de racines ou de divisions indirectement, on élimine également ces opérations des définitions de variable en utilisant un *inlining* partiel :

◇ let x $= a.b + \sqrt{(c+d)/f}$ in P $\longrightarrow$

$\qquad$ let $(x_1, x_2, x_3) = (a.b, c+d, e)$ in P[x := $x_1 + \sqrt{x_2/x_3}$]

- Nommer les sous expressions qui ne contiennent ni racine ni division
- *Inliner* le contexte qui les contient

6 . Définitions avec conditionnelles

Cette notion d'*inlining* partiel peut s'étendre à des définitions de variables qui contiennent des tests :

◇ let x = if F then a/b else $c + \sqrt{d}$ in P

Le but est alors de trouver une représentation commune à toutes les expressions qui correspondent aux différents cas et d'*inliner* cette expression :

◇ let (x_1, x_2, x_3) = if F then $(a, b, 0)$ else $(c, 1, d)$ in P[x := $\frac{x_1 + \sqrt{x_3}}{x_2}$]

Cette représentation commune provient d'une **anti-unification** avec contraintes des expressions correspondant aux différents cas des tests.

Soient $e_1, ..., e_n$, un anti-unificateur de ces termes est un terme t tel que :

$$\forall i \in [|1, ..., n|],\ \exists \sigma_i \in \mathcal{P}erm(\mathrm{Var}),\ t.\sigma_i = e_i$$

Avec la contrainte que σ_i ne contient ni racine ni division.

Cette anti-unification nous permet donc de définir une fonction `elim_let`(x,p1,p2) qui renvoit x', p1', p2' tels que :

$$\text{let x = p1 in p2} \quad \stackrel{sem}{==} \quad \text{let x' = p1' in p2'}$$

où p1' ne contient pas de racine.

7 . Transformation complète

La transformation est donnée par la fonction récursive `Elim`(p) :

- si p est une expression booléene, retourner `elim_bool`(p)
- si p est une expression arithmétique, retourner p
- si p = let x = p1 in p2 :
  - p1r := `Elim`(p1)
  - x',p1',p2' := `elim_let`(x,p1r,p2)
  - retourner let x' = p1' in `Elim`(p2')
- si p = if F then p1 else p2

  retourner if `Elim`(F) then `Elim`(p1) else `Elim`(p2)

8 . Conclusion

Nous avons donc conçu une transformation de programme qui permet d'éliminer les racines et les divisions de tous les booléens d'un programme. Cette transformation est implantée en OCamL.

De plus nous avons :

- une spécification et la preuve de correction en PVS
- transposé cette spécification en une tactique réflexive qui permet de transformer automatiquement des buts dans PVS

9 . Référence

P. Neron. A formal proof of square root and division elimination in embedded programs. In C. Hawblitzel and D. Miller, editors, CPP, volume 7679 of *Lecture Notes in Computer Science*, pages 256–272, 2012.

http://www.lix.polytechnique.fr/~neron/