ECOLE POLYTECHNIQUE
PROMOTION 2009
BOST Raphaël

RAPPORT DE STAGE DE RECHERCHE

# Nombres réels et transformation de programmes

NON CONFIDENTIEL

| | | |
|---|---|---|
| Département | : | Département d'Informatique |
| Champ de l'option | : | Informatique |
| Directeur de l'option | : | Olivier BOURNEZ |
| Directeurs de stage | : | Gilles DOWEK & Pierre NERON |
| Dates du stage | : | 1/04/2012 - 15/07/2012 |
| Adresse de l'organisme | : | INRIA - Antenne parisienne<br>23 avenue d'Italie<br>CS 81321<br>75214 Paris Cedex 13 |

**Résumé**

La présente étude propose un algorithme de transformation de programmes réalisé au sein de l'équipe DEDUCTEAM de l'INRIA Paris-Rocquencourt. L'objectif est de transformer du code, où apparaissent des opérations de racines carrées et de divisions, en un code sémantiquement équivalent ne les utilisant plus. Nous verrons que cette transformation, en éliminant les plus grandes sources d'erreurs d'arrondis dans les programmes, peut avoir un intérêt tout particulier pour les systèmes embarqués.

Différents points de vues sont abordés, aussi bien sur des considérations théoriques portant sur la complexité de nos algorithmes que sur des considérations techniques ayant trait à l'implémentation efficace de l'algorithme que nous présentons.

**Abstract**

We present a study for a program transformation algorithm developped in the DEDUCTEAM research team at INRIA Paris-Rocquencourt. The goal is to tranform a program which uses square root and division operations in a semantically equivalent code which no longer uses them. We will see that this transformation, which eliminates the biggest source of rounding errors in the programmes, can be very interesting for embedded systems.

Various points of view will be approached, with theoretical considerations on the complexity of our algorithms as well as with technical considerations on the efficient implementation of the algorithm we will have presented.

# Table des matières

# Introduction

Lors du développement de nouveaux algorithmes, nous raisonnons de manière générale en utilisant des réels sans limitation de taille ni de précision. Puis, quand il s'agit d'implémenter ces algorithmes, à moins d'utiliser des logiciels de calcul formel, nous ne pouvons utiliser les réels en tant que tels, mais uniquement des représentations (nombres à virgules flottants, à virgule fixe, représentation sous forme d'intervalles, ...). Des structures de nombres à précision arbitraire ont ainsi été développées afin de s'affranchir de certaines limitations des flottants mais certains problèmes demeurent avec ces structures. Parmi ceux-ci, le principal problème est le fait que la précision arbitraire empêche de déterminer à la compilation la taille mémoire des programmes l'utilisant - écueil majeur pour les programmes embarqués.

Dans ce cas, nous devons donc souvent travailler avec les nombres flottants et leurs erreurs d'arrondis. Cependant, celles-ci peuvent avoir des conséquences importantes sur le bon fonctionnement des algorithmes conçus au départ pour travailler avec des réels, ce qui est particulièrement critique pour des algorithmes embarqués prouvés dans les réels et censés apporter une certaine sécurité.

Pour ce type de systèmes, la source la plus importante d'erreurs d'arrondis est l'utilisation des opérations de division et de racine carrée par rapport aux trois autres opérations usuelles (addition, soustraction et multiplication). L'élimination systématique des racines carrées et des divisions dans un programme est ainsi une avancée importante dans la conception de programmes embarqués car elle dispense d'avoir à supposer l'exactitude de ces opérations pour les nombres flottants, la preuve des programmes ne reposant plus que sur des hypothèses de correction des opérations de sommes et de multiplications entre flottants.

Une méthode d'élimination a déjà été mise au point par Pierre Néron [**?**]. Cette méthode repose notamment sur un principe systématique de suppression des racines carrées dans les comparaisons. Malheureusement, cette méthode génère du code dont la taille est doublement exponentielle en le nombre de racines distinctes présentes dans les comparaisons. Notre objectif de départ était donc de développer une amélioration fondamentale de cette méthode produisant un programme plus petit que par la procédure précédente mais ayant toujours la propriété de conserver la sémantique des programmes transformés.

Nous avons ainsi finalement créé un nouvel outil capable, tout en conservant des performances acceptables, d'engendrer, dans certains cas, du code jusqu'à 140 fois plus petit que par la transformation déjà existante dans le cas de petits programmes à transformer (moins de 2 Ko).

Nous préciserons ainsi dans un premier temps le cadre de nos modifications permettant de réduire la taille du code généré puis les méthodes utilisées pour y parvenir. Ensuite, nous améliorerons ce premier algorithme en lui apportant des modifications d'importance diverse et surtout pouvant avoir une conséquence sur la complétude du programme. Nous présenterons aussi les optimisations d'implémentation essentielles que nous avons effectuées afin d'obtenir un programme parmi les plus efficaces possibles. Enfin, nous présenterons les résultats numériques sur la taille des programmes que nous avons obtenus grâce aux optimisations ainsi que les orientations futures que l'on pourrait prendre pour ce travail.

# 1 Présentation

Nous commençons par présenter un algorithme conçu par Pierre Néron pour l'élimination des racines carrées dans un programme ne contenant pas de boucle ni de récursion. Nous allons reprendre toutes ces notions dans la suite et expliquer comment améliorer ce premier algorithme.

## 1.1 Problématique

Le principal problème concernant l'emploi des racines carrées et des divisions dans un programme se situe au niveau de leur utilisation lors de comparaison. En effet, si l'on peut se permettre une certaine tolérance concernant les erreurs d'arrondis pour de simples calculs numériques (ne serait-ce qu'à cause de la précision des capteurs et des contrôleurs des systèmes embarqués), ceux-ci deviennent critiques lorsqu'ils interviennent lors de comparaisons, car ils cassent la *continuité* du résultat en fonction des approximations. Prenons par exemple le programme suivant et son exécution abstraite dans les réels d'un côté, et une exécution dans les flottants :

Listing 1: Erreurs d'arrondi

```
let x = √2 in
let y = x² in
let c =
  if y ≥ 2 then 666
  else 42
```

| Dans les réels : | Dans les flottants : |
|---|---|
| $x = \sqrt{2}$ | $x = 1.414$ |
| $y = 2$ | $y = 1.999$ |
| $c = 666$ | $c = 42$ |

Nous cherchons donc à enlever les divisions et les racines des formules comportant des comparaisons. Le principe général est basé sur une modification de telles expressions, par exemple : $\sqrt{x} > y \Leftrightarrow y < 0 \vee x > y^2$.

Nous pouvons aussi remarquer que ce problème est un cas particulier de l'élimintation des quantificateurs logiques dans les expressions où les seules opérations arithmétiques utilisées sont $+, -$ et $\times$. En effet, supposons que l'on ait une formule du type $a \ \mathcal{R} \ b$ (avec $\mathcal{R} \in \{=, \neq, >, \geq, <, \leq\}$) où $a$ s'écrit sous la forme $a = a_1 + a_2\sqrt{x}$. On a alors

$$a_1 + a_2\sqrt{x} \ \mathcal{R} \ b \iff \exists x', x'^2 = x \wedge x' \geq 0 \wedge (a_1 + a_2 x') \ \mathcal{R} \ b$$

De même,

$$(a_1 + \frac{a_2}{x}) \ \mathcal{R} \ b \iff \exists x', x \times x' = 1 \wedge (a_1 + a_2 x') \ \mathcal{R} \ b$$

Ainsi, par induction, nous pouvons remplacer toutes les racines carrées et les divisions par des quantificateurs. Or nous savons par le théorème de Tarski [**?**, **?**] que les quantificateurs peuvent être éliminés de telles formules et par conséquent qu'un procédé capable de supprimer racines et divisions existe.

## 1.2 Cadre

Nous travaillons sur des systèmes embarqués et pour lesquels les programmes s'exécutent, entre autres, avec une utilisation de la mémoire qui doit être définie à la compilation. Cela nous interdit ainsi l'utilisation de listes ou de tout autre objet dont la taille serait modifiable à l'exécution (notons que l'on peut remplacer les listes de taille fixée par des empilements de paires, c'est à dire des vecteurs de taille fixée à la compilation).

### 1.2.1 Langage

Le langage que nous avons choisi en entrée de notre programme est un langage fonctionnel typé similaire à Caml qui utilise un sous-ensemble $\mathbb{F}$ de $\mathbb{R}$ comme représentation des nombres réels. Il contient des déclarations de variables (sous la forme de `let...in...`), des tests (`if...then...else...`), les opérations arithmétiques usuelles ($+, -, \times, /, \sqrt{}$), les relations entre nombres ($=, \neq, >, \geq, <, \leq$) et les opérateurs booléens ($\wedge, \vee, \neg$). Enfin, nous ajoutons aussi les paires et les opérateurs de projection associés (`Fst` et `Snd`). Ainsi, la structure abstraite de notre langage est la suivante :

**Definition 1.2.1** : *Syntaxe du langage*

Prog = | Variable
| Constante $\in \mathbb{F} \cup \{True, False\}$
| `uop` Prog , `uop` $\in \{\neg, -, \surd\}$
| Prog `op` Prog , `op` $\in \{+, \times, /, =, \neq, >, \geq, <, \leq, \wedge, \vee\}$
| (Prog,Prog)
| `Fst` Prog
| `Snd` Prog
| `let` Variable = Prog `in` Prog
| `if` Prog `then` Prog `else` Prog

Nous introduisons pour ce langage des règles de typage classiques.

**Definition 1.2.2** : *Règles de typage du langage*
Nous définissons la structure de type

$$Type = \mathcal{F} \mid \mathcal{B} \mid Type \times Type$$

et les règles de typage :

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash e : \mathcal{F}}{\Gamma \vdash \texttt{uop}\ e : \mathcal{F}}\texttt{uop} \in \{-, \surd\}$$

$$\frac{\Gamma \vdash e_1 : \mathcal{F} \quad \Gamma \vdash e_2 : \mathcal{F}}{\Gamma \vdash e_1\ \texttt{op}\ e_2 : \mathcal{B}}\texttt{op} \in \{=, \neq, >, \geq, <, \leq\}$$

$$\frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \texttt{Fst(e)} : T_1}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2}$$

$$\frac{\Gamma \vdash f : \mathcal{B} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if}\ f\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T}$$

$$\frac{\Gamma \vdash e : \mathcal{B}}{\Gamma \vdash \neg e : \mathcal{B}}$$

$$\frac{\Gamma \vdash e_1 : \mathcal{F} \quad \Gamma \vdash e_2 : \mathcal{F}}{\Gamma \vdash e_1\ \texttt{op}\ e_2 : \mathcal{F}}\texttt{op} \in \{+, \times, /\}$$

$$\frac{\Gamma \vdash e_1 : \mathcal{B} \quad \Gamma \vdash e_2 : \mathcal{B}}{\Gamma \vdash e_1\ \texttt{op}\ e_2 : \mathcal{B}}\texttt{op} \in \{\wedge, \vee\}$$

$$\frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \texttt{Snd(e)} : T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let x =}\ e_1\ \texttt{in}\ e_2 : T_2}$$

### 1.2.2 Forme normale d'un programme

Dans nos transformations futures, nous voulons éviter d'avoir des déclarations et des tests dans les expressions arithmétiques. Par conséquent, nous avons défini une transformation (que nous ne préciserons pas ici) qui permet de les extraire et de nous assurer de l'absence de déclaration ou de tests dans les opérandes rencontrées.

Dans la suite, nous considérons donc uniquement des programmes sous forme normale, c'est-à-dire pour lesquels les tests et les expressions ne sont pas imbriqués. Les expressions E sont construites avec des opérateurs ou des paires mais sans déclaration ni test :

E = Variable | Constante | `uop` E | E `op` E | `Fst` E | `Snd` E | (E,E)

Les programmes P sous forme normale sont sous la forme

P = `let` Variable = P `in` P | `if` P `then` P `else` P | E

Voici un exemple de mise sous forme normale d'un programme :

```
let (e1, e2) =
  (if b then 1794 else 2009
   let x = √2 in x + 3
in e1 + e2
```

$\longrightarrow$

```
let (e1, e2) =
  let k = if b then 1794 else 2009 in
  let x = √2 in
    (k, x + 3)
in e1 + e2
```

### 1.2.3 Classes de programme - Notations

On peut définir différentes classes de programmes selon l'utilisation où non des opérations de racine et de division ou bien la présence d'instruction de tests et de déclaration.

Considérons les classes d'opérateurs suivantes :

$$Nuop_{\surd} = \{-, \surd\} \qquad Nuop = \{-\}$$
$$Nop_{/} = \{+, \times, /\} \qquad Nop = \{+, \times\}$$
$$Buop = \{\neg\},\ Bop = \{\wedge, \vee\} \qquad Cop = \{=, \neq, >, \geq, <, \leq\}$$

Nous pouvons alors définir différents ensembles d'expressions numériques :

$$N = Nuop\ N \mid N\ Nop\ N$$

$$N_{\surd,/} = Nuop_{\surd}\ N_{\surd,/} \mid N_{\surd,/}\ Nop_{/}\ N_{\surd,/}$$

Comme les booléens ne peuvent pas être opérandes des divisions et racines carrées, nous pouvons traiter séparément le cas des programmes booléens sans tests, racines ni division (mais avec des déclarations) :

$$B_{let} = Buop\, B_{let} \mid B_{let}\, Bop\, B_{let} \mid N\, Cop\, N \mid (B_{let}, B_{let}) \mid \texttt{Fst}\ B_{let} \mid \texttt{Snd}\ B_{let} \mid \texttt{let}\ Variable = B_{let}\ \texttt{in}\ B_{let}$$

De même, définissons les classes respectives des expressions dont les déclarations de variables sont réservées aux booléens et qui d'une part ne contiennent pas de racines ni de division et qui d'autre part peuvent en contenir :

$$E_N = N \mid B_{let} \mid (E_N, E_N) \mid \texttt{Fst}\ E_N \mid \texttt{Snd}\ E_N$$
$$E_{N_{\surd,/}} = N_{\surd,/} \mid B_{let} \mid (E_{N_{\surd,/}}, E_{N_{\surd,/}}) \mid \texttt{Fst}\ E_{N_{\surd,/}} \mid \texttt{Snd}\ E_{N_{\surd,/}}$$

Enfin, nous définissons les classes des programmes qui ne contiennent pas de racine ni de division,

$$P_N = \texttt{let}\ Variable = P_N\ \texttt{in}\ P_N \mid \texttt{if}\ P_N\ \texttt{then}\ P_N\ \texttt{else}\ P_N \mid E_N$$

et de ceux qui contiennent ces opérations mais uniquement dans l'expression numérique finale

$$P_{N_{\surd,/}} = \texttt{let}\ Variable = P_N\ \texttt{in}\ P_{N_{\surd,/}} \mid \texttt{if}\ P_N\ \texttt{then}\ P_{N_{\surd,/}}\ \texttt{else}\ P_{N_{\surd,/}} \mid E_{N_{\surd,/}}$$

Nous voulons donc transformer les programmes de la classe $P$ à la classe $P_{N_{\surd,/}}$.

# 2 Un premier algorithme de transformation

L'algorithme que nous allons présenter ici est issu des travaux de Pierre Néron. Mon principal objectif a été de l'améliorer et de le rendre plus efficace.

## 2.1 Suppression des racines et divisions dans les expressions booléennes

Nous voulons dans un premier temps supprimer les racines apparaissant dans des booléens, c'est-à-dire dans des comparaisons aussi bien dans des tests que dans des assignations de booléens. Nous devons donc remplacer des expressions du type $e_1 \mathcal{R} e_2$ où $e_1$ et $e_2$ sont des expressions algébriques de $N_{\surd,/}$ et $\mathcal{R}$ une relation de comparaison par des relations de comparaison ne faisant intervenir que des expressions de $N$.

### 2.1.1 Algorithme

Pour cela, Néron propose de mettre ces expressions sous une forme normale consistant à mettre en premier opérateur soit une division soit une racine carrée.

**Definition 2.1.1** : *Forme normale d'une expression algébrique*
Une expression sous forme normale est définie comme suit :

$$DNF = PNF \mid \frac{PNF}{PNF}$$

où

$$PNF = PNF + PNF \mid -PNF \mid PNF \times PNF \mid \sqrt{DNF}$$

Il est immédiat de construire un algorithme transformant une expression algébrique quelconque en expression en forme normale (il s'agit en fait de mettre l'expression au même dénominateur ainsi que les opérandes des racines présentes dans l'expression).

Cette forme normale permet de définir deux règles d'élimination, une pour les divisions et une pour les racines carrées et de donner un algorithme dont il est aisé de montrer la terminaison grâce à une récurrence sur le nombre de racines présentes dans l'expression à transformer.

**Definition 2.1.2** : *Règle d'élimination des divisions*

- Pour $\mathcal{R} \in \{=, \neq\}$
$$\frac{A}{B}\mathcal{R}\frac{C}{D} \longrightarrow A.D - C.B\ \mathcal{R}\ 0$$
- Pour $\mathcal{R} \in \{>, \geq, <, \leq\}$
$$\frac{A}{B}\mathcal{R}\frac{C}{D} \longrightarrow A.B.D^2 - C.D.B^2\ \mathcal{R}\ 0$$

Nous utilisons ces règles car il y a équivalence entre les deux expressions : $\frac{A}{B} \geq \frac{C}{D} \Leftrightarrow A.B.D^2 - C.D.B^2 \geq 0$.
Pour la règle d'élimination des racines dans les booléens, nous remarquons l'équivalence suivante :

$$\begin{aligned} P\sqrt{Q} + R > 0 &\Longleftrightarrow (P > 0\ \wedge\ R > 0) \vee (P > 0\ \wedge\ R \leq 0\ \wedge\ P^2Q - R^2 > 0) \vee (P \leq 0\ \wedge\ R > 0\ \wedge\ R^2 - P^2Q > 0) \\ &\Longleftrightarrow (P > 0\ \wedge\ R > 0) \vee (P > 0\ \wedge\ P^2Q - R^2 > 0) \vee (R > 0\ \wedge\ P^2Q - R^2 < 0) \end{aligned}$$

Remarquons qu'en fait, il n'y a qu'un petit nombre d'atomes différents dans cette expression et que l'on peut donc en simplifier l'écriture :

$$\begin{aligned} P\sqrt{Q} + R > 0 \quad \longrightarrow \quad & \texttt{let at\_p =}\ \ P > 0\ \ \texttt{in} \\ & \texttt{let at\_r =}\ \ R > 0\ \ \texttt{in} \\ & \texttt{let at\_pr =}\ \ P^2Q - R^2 > 0\ \ \texttt{in} \\ & \texttt{let at\_epr =}\ \ P^2Q - R^2 \neq 0\ \ \texttt{in} \\ & \quad (\texttt{at\_p} \wedge \texttt{at\_r}) \vee (\texttt{at\_p} \wedge \texttt{at\_pr}) \vee (\texttt{at\_r} \wedge \neg\texttt{at\_pr} \wedge \texttt{at\_epr}) \end{aligned}$$

On peut déterminer de telles équivalences pour toutes les autres opérations (à l'aide d'une table de vérité par exemple) et déterminer les règles d'élimination qui leur correspondent, par exemple pour $P\sqrt{Q} + R = 0$ :

$$P\sqrt{Q} + R = 0 \longrightarrow (P.R \leq 0)\ \wedge\ (P^2Q - R^2 = 0)$$

Ainsi, pour éliminer une racine $\sqrt{Q}$ d'une expression, nous factorisons $\sqrt{Q}$ dans cette expression que l'on exprime alors sous la forme $P\sqrt{Q} + R$ avec $\sqrt{Q}$ n'apparaissant ni dans $P$ ni dans $R$ ($\sqrt{\sqrt{Q}}$ peut par contre apparaître - on ne considère que les racines du premier niveau) et l'on applique une des règles donnée plus haut.

Finalement, pour supprimer toutes les racines et les divisions, nous utilisons l'algorithme schématisé par le diagramme de la figure 1.

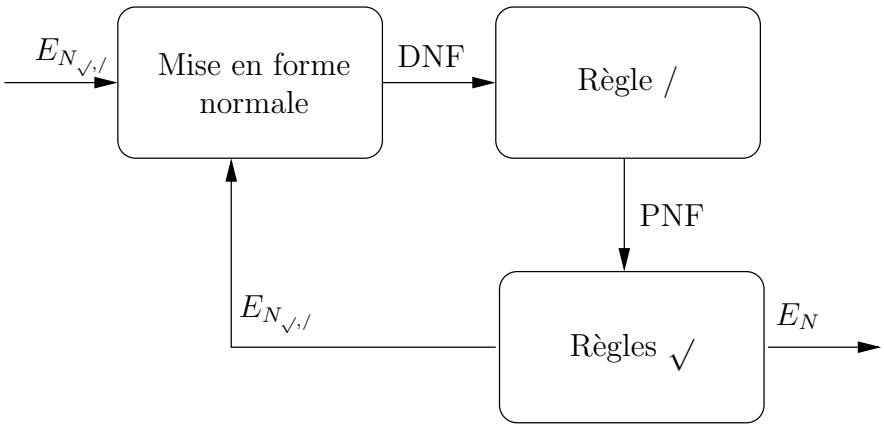



FIGURE 1: Schéma de la normalisation

#### 2.1.2 Complexité

Pour un atome $A$, notons $|A|_{\sqrt{}}$ le nombre de racines distinctes. Notons $a_n$ le nombre d'atomes après la suppression complète de $n$ racines d'une relation d'ordre $(>, \geq, <, \leq)$ et $b_n$ le nombres d'atomes après l'élimination de $n$ racines d'une égalité ou inégalité ($=$ ou $\neq$). On a alors la double relation de récurrence

$$\begin{cases} a_n = 3.a_{n-1} + b_{n-1} \\ b_n = a_{n-1} + b_{n-1} \end{cases}$$

Ainsi, le nombre d'atome final après élimination des racines d'une expression du type $A \mathcal{R} 0$ avec $n$ racines distinctes dans $A$ est $O\left(\left(2+\sqrt{2}\right)^n\right)$.

Il est par conséquent essentiel de contrôler le nombre de racines distinctes que nous pourrions produire en transformant notre programme. C'est cette remarque qui est à l'origine des réflexions nous allons présenter par la suite : nous voulons limiter le plus possible la taille du code généré en réduisant le nombre de nouvelles racines produites par notre algorithme.

### 2.2 Suppression des racines et divisions dans les déclarations

Maintenant que nous savons supprimer les racines des tests, il nous reste à les supprimer dans les déclarations de variable pour avoir un algorithme complet de suppression des racines.

#### 2.2.1 Modèles

Un moyen de supprimer les racines lors d'une affectation serait d'*expanser* (où *inliner*) les affectations mais cela conduirait à une explosion de la taille du code. A la place, nous pouvons n'inliner que les expressions posant un problème, c'est-à-dire les racines carrées et les divisions (dans la suite, $P[e/\mathtt{x}]$ signifie remplacer les occurrences de `x` dans $P$ par $e$, c'est à dire expanser $e$ dans $P$) :

| | | |
|---|---|---|
| `let x =` $a_1 + 3.a_2$ `in` $P$ | $\longrightarrow$ | `let x =` $a_1 + 3.a_2$ `in` $P$ : pas d'inlining car pas de racine dans `x` |
| `let x =` $\sqrt{b}$ `in` $P$ | $\longrightarrow$ | $P\left[\sqrt{b}/\mathtt{x}\right]$ : inlining de $\sqrt{b}$ |
| `let x =` $\frac{a_1+3.a_2+\sqrt{b+\sqrt{c}}}{d+\sqrt{e}}$ `in` $P$ | $\longrightarrow$ | `let (x1,x2) =` $(a_1 + 3.a_2, d)$ `in` $P\left[\frac{x_1+\sqrt{b+\sqrt{c}}}{x_2+\sqrt{e}}/\mathtt{x}\right]$ : les expressions qui ne sont pas sous les racines ne sont pas expansées |

Pour effectuer cet inlining partiel, introduisons la notion de *modèle* :

**Definition 2.2.1** : *Modèle*
Etant donné une expression $exp \in E_{N_{\surd,/}}$ un *modèle* (ou *template*) de $exp$ est une fonction $t : (E_N)^n \to E_{N_{\surd,/}}$ telle que

$$\exists (e_1, ...e_n) \in (E_N)^n \text{ tel que } exp = t\left((e_1, ...e_n)\right)$$

L'objectif essentiel de mon travail a été de mettre au point une méthode pour expanser de manière efficace les variables dont la définition dépend d'un test. Pour cela, nous devons aussi définir le concept de *modèle commun*. En effet, lorsque l'on cherche à supprimer les racines d'une déclaration dépendant d'un test - comme dans l'exemple suivant - nous pouvons soit dupliquer le code suivant la déclaration puis inliner les deux affectations possibles dans chacune des branches du test, doublant par là même la taille du code, soit trouver une manière commune aux deux branches d'écrire la variable affectée et n'expanser qu'une seule fois.

Listing 2: Definition multiple

```
let x =
  if test then
    √a + √b
  else
    √(a + √b)
in ...
```

Nous pouvons ici trouver une forme commune à $\sqrt{a} + \sqrt{b}$ et à $\sqrt{a + \sqrt{b}}$ en remarquant que ces deux expressions peuvent s'écrire sous la forme $\sqrt{a + x_1\sqrt{b} + x_2\sqrt{b}}$ et assigner $(x_1, x_2)$ à $(0, 1)$ ou $(1, 0)$.
De manière plus générale, nous pouvons donc définir ce qu'est un modèle commun à plusieurs expressions :

**Definition 2.2.2** : *Modèle commun*
Etant donné $m$ expressions $(exp^1, ..., exp^m) \in (E_{N_{\surd,/}})^m$, un *modèle commun* à $(exp^1, ..., exp^m)$ est une fonction $t : (E_N)^n \to E_{N_{\surd,/}}$ telle que

$$\forall j \in \{1, 2, ..., m\}\ \exists (e_1^j, ...e_n^j) \in (E_N)^n \text{ tel que } exp^j = t\left((e_1^j, ...e_n^j)\right).$$

$t$ est un modèle de toutes les expressions $exp^j$

### 2.2.2 Construction des modèles

Lorsque nous construisons les modèles, nous voulons les construire de la manière la plus économe (c'est-à-dire avec le moins de racines distinctes apparaissant à cause de la complexité exponentielle de la suppression des racines évoquée en 2.1.2 ). Si nous représentons les expressions sous forme d'un arbre, nous voulons alors créer l'arbre *union*, le plus petit arbre contenant tous les autres arbres.

Cependant, afin de pouvoir partager des expressions communes sous les racines carrées, nous utilisons une structure de *graphe dirigés acycliques* (DAGs - Directed Acyclic Graphs) au lieu d'une simple structure d'arbre. Mais nous cherchons toujours un graphe contentant les représentations de toutes les expressions dont on veut connaître le modèle commun.

L'implémentation de cette structure est effectuée de deux manières : une pour les expressions en tant que telles et une version abstraite de la première qui ne garde que la structure sémantique de l'expression. Nous ne présenterons ici que la structure concrète.

**Definition 2.2.3** : *Structure concrète de graphe acyclique dirigé*

```
type dag_elt =
  | ExprD of program
  | VecD of (program*dag_elt list)list
  | DivD of dag_elt*dag_elt
  | PairD of dag_elt*dag_elt
  | PtD of int

type dag = dag_elt list
```

ExpD désigne une expression dans laquelle aucune racine carrée n'apparait (codée dans le type `program`). DivD et PairD désignent respectivement une opération de division et une paire. PtD désigne un pointeur vers un autre élément du graphe. VecD désigne une somme de produit de racines. Ainsi `VecD [(`$a_i$`,[`$l_j$`]`$_i$`)]` a la sémantique suivante :

$$\sum_i a_i \prod_j \sqrt{l_{j,i}} \tag{2.2.1}$$

Nous stockons tous les éléments du graphe dans une liste dont la tête est la racine du graphe et nous accédons aux autres éléments *via* leur indice dans la liste. `[ VecD[ (Value a,[]); (Const 3,[PtD(1)])];Value b]` représente $a + 3\sqrt{b}$.

Pour des raison de lisibilité, nous ne ferons appel à ces constructions que pour définir formellement nos transformations et nous y préférerons une présentation en graphe ou linéaire. Ainsi, le DAG précédent correspond aux deux représentations de la figure 2.


(a) Représentation en graphe

(b) Représentation linéaire

FIGURE 2: Exemple de DAG simple

Pour illustrer l'intérêt des DAGs, regardons la représentation de $\frac{a+\sqrt{b+\sqrt{c}+\sqrt{e}}+\sqrt{c}\sqrt{e}}{d+\sqrt{c}}$ sous forme d'arbre et sous forme de DAG (figure 3).

### 2.2.3 Union des modèles

Dans le cas de déclarations dépendantes d'un test (de la forme `let if`), nous devons donc construire un modèle commun. Pour cela, nous procédons à l'union des modèles des expressions apparaissant dans la déclaration.

Pour unir les modèles, nous nous intéressons uniquement à la structure de l'expression et non à l'expression concrète. En effet, une fois une structure commune trouvée, il est aisé de trouver comment la "remplir" (c'est-à-dire trouver les differents $n$-uplets $(e_k^j)$ faisant correspondre la structure et l'expression). Il s'agit en effet de parcourir en même temps le DAG commun et celui de l'expression en même temps en mettant des constantes nulles aux branches du DAG commun n'apparaissant pas dans le DAG de l'expression (le cas contraire ne se manifeste jamais puis on a une inclusion).

Ainsi, nous pouvons définir des règles récursives d'union de deux DAGs. En voici les plus simples :

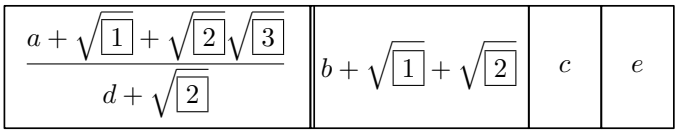
**Definition 2.2.4** : *Règles d'union de modèles*
Etant donné deux éléments de modèles abstraits (de type `temp_elt`) $t_1$ et $t_2$, nous appliquons les règles de transformation Union$(t_1, t_2)$ selon la valeur prise pas le couple $(t_1, t_2)$

$$(\text{PairT}(t_{1_1}, t_{1_2}), \text{PairT}(t_{2_1}, t_{2_2})) \longrightarrow \text{PairT}(\text{Union}(t_{1_1}, t_{2_1}), \text{Union}(t_{1_2}, t_{2_2})) \tag{2.2.2}$$

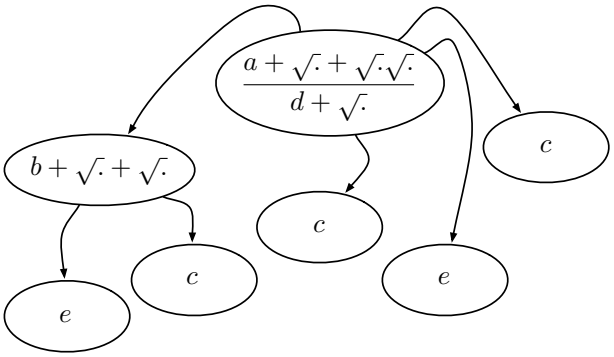
$$(\text{DivT}(t_{1_1}, t_{1_2}), \text{DivT}(t_{2_1}, t_{2_2})) \longrightarrow \text{DivT}(\text{Union}(t_{1_1}, t_{2_1}), \text{Union}(t_{1_2}, t_{2_2})) \tag{2.2.3}$$

$$(\text{DivT}(t_{1_1}, t_{1_2}), t_2), t_2 \neq \text{PairT}(...) \longrightarrow \text{DivT}(\text{Union}(t_{1_1}, t_2), \text{Union}\,(t_{1_2}, 1) \tag{2.2.4}$$

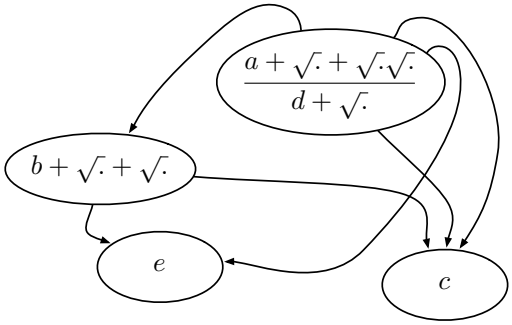
Pour deux DAGs entiers (c'est à dire une liste d'éléments), nous unissons les deux listes élément par élément.




(a) Sous-forme de DAG




(b) Sous-forme d'arbre




(c) Sous-forme de graphe

FIGURE 3: Représentations de $\frac{a+\sqrt{b+\sqrt{c}+\sqrt{e}}+\sqrt{c}\sqrt{e}}{d+\sqrt{c}}$

Expliquons un peu ces transformations :
- (2.2.2) est immédiate : on effectue l'union membre par membre
- (2.2.3) aussi : lorsque l'on doit unir deux fraction on unit ensemble respectivement les numérateurs et les dénominateurs
- (2.2.4) consiste à dire que toute expression peut se mettre sous forme fractionnaire avec un dénominateur égal à 1
- Enfin, nous unifions les listes des DAGs élément par élément afin de faire correspondre les éléments sous les racines.

Une fois cette union des modèles effectuée, nous disposons d'une méthode pour inliner les déclaration dépendant d'un test (telle celle du listing 2). Cependant, nous devons nous assurer que le modèle que nous créons ne contient pas trop de racines et de divisions et pour cela, nous devons pouvoir étudier un maximum de modèles commun aux expressions que nous voulons unir. Nous verrons cela dans la sous-section 2.3.

## 2.3 Pratique de l'union de modèles

Dans la sous-section 2.2.3, nous avons défini un certain nombre de règles d'union de modèles. Voyons d'un peu plus près comment les utiliser pour effectuer une union entre modèles qui soit satisfaisante, *i.e.* générant le moins de racines différentes.

### 2.3.1 Eléments indéterminés

La première question que nous devons nous poser est que faire lorsque nous unissons deux DAGs qui n'ont pas la même taille ? Comment faire correspondre les éléments de la liste la plus longue avec des éléments qui n'existent pas dans la liste plus petite ? Pour cela nous agrandissons la liste la plus courte pour que les deux listes aient la même taille et remplissons ces nouvelles cases par un élément vide ou *indéterminé*. Pour remplir cet indéterminé, nous avons deux choix possibles en tenant compte du fait que l'on ne doit avoir

sous des racines que des expressions positives. Ainsi, nous pouvons dans tous les cas remplacer l'indéterminé par 0. Par contre, nous ne pouvons remplacer par l'élément de la première liste que si celui si est positif. Par exemple, nous allons voir comment unir $\sqrt{a+\sqrt{b}}$ et $\sqrt{a}$ dans deux cas, selon que $b$ est positif ou non.

Listing 3: Pas d'information sur $b$

```
let x = if test then
          √(a+√b)
        else
          √a
in ...
```

Listing 4: $b$ est positif

```
if b>0 then
  let x = if test then
            √(a+√b)
          else
            √a
  in ...
else ...
```

Dans les deux cas, nous partons des DAG auxquels nous ajoutons un indéterminé (Figure 4). Puis nous remplissons cet indéterminé (Figure 5). Dans le premier cas, nous ne pouvons utiliser que le remplacement (a)

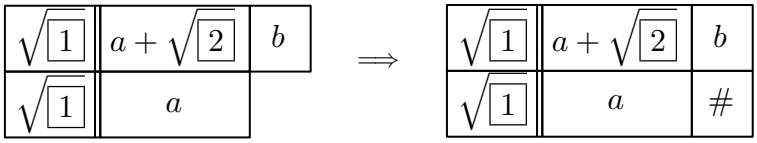
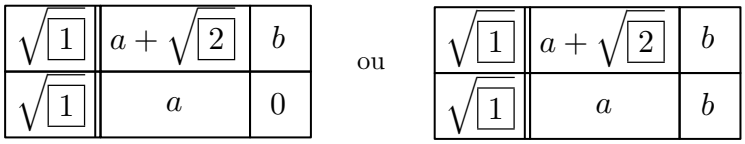



FIGURE 4: Ajout d'un indéterminé




(a) Remplacement par 0 (b) Remplacement par $b$

FIGURE 5: Possibilités de remplissage d'un indéterminé

et dans le second cas, comme nous savons que $b$ est positif, nous pouvons aussi utiliser le remplacement (b) (car nous pouvons prendre la racine carrée de $b$ sans problème alors que ce n'est pas le cas si l'on ne sait pas que $b$ est positif). Ainsi, nous pouvons même intégrer $b$ directement dans le modèle, comme une constante, réduisant le nombre d'argument du modèle. Le remplacement (b) abouti par conséquent au modèle $\sqrt{sq_1 + x_1\sqrt{b}}$ avec $(sq_1, x_1)$ valant soit $(a, 1)$ soit $(a, 0)$.

Lorsqu'il s'agit d'unir plus de deux DAGs, nous ne devons insérer un élément comme constante non seulement que s'il est positif dans tous les cas, mais aussi seulement si tous les éléments des autres DAGs de même indice sont soit égaux, soit des indéterminés, ou autrement dit, que c'est un noeud commun à tous les DAGs.

### 2.3.2 Mesure d'un modèle

Afin d'évaluer la *qualité* d'un modèle, nous devons lui associer une mesure qui permette de comparer deux modèles. Nous recherchons avant tout des modèles qui, par la suite, ne vont pas engendrer une explosion de la taille du code transformé. Or le principal point critique concernant la taille du code est l'élimination des racines et divisions dans les tests booléens où nous avons vu que le nombre d'atomes engendrés est exponentiel en la taille des racines distinctes de l'expression intervenant dans la comparaison (cf. section 2.1.2). Nous avons donc définit une mesure $\text{Mes}(e)$ de la taille d'un modèle $e$ :

$$\text{Mes}(e) = \begin{cases} \text{Mes}(e_1) + \text{Mes}(e_2) & \text{si } e = (e_1, e_2) \\ 4^{|e|_\surd} & \text{sinon} \end{cases}$$

où $|\ e\ |_\surd$ est le nombre de racines distinctes de $e$.

Nous utilisons la formule $4^{|e|_\surd}$ car on sait qu'il y a asymptotiquement $(2+\sqrt{2})^{|e|_\surd}$ atomes lors de l'élimination de $e$ dans la relation de comparaison $e\ \mathcal{R}\ 0$ et que $\lceil 2+\sqrt{2} \rceil = 4$.

Ainsi, un modèle $e = (e_1, ..., e_m)$ qui a une mesure plus petite qu'un modèle $f = (f_1, ..., f_m)$ des mêmes expressions $(exp_1, ..., exp_m)$ générera moins d'atomes lors de l'élimination de racines dans une expression du type

$$(exp_1\ \mathcal{R}_1\ 0) \vee ... \vee (exp_m\ \mathcal{R}_m\ 0)$$

### 2.3.3 Permutations

Pour obtenir le meilleur modèle commun possible, il faut aussi bien regarder les permutations possibles des DAG. Supposons que l'on construise les DAGs *via* un algorithme DFS sur les expressions et que l'on veuille unir $\sqrt{a} + \sqrt{b+\sqrt{c}}$ et $\sqrt{e+\sqrt{f}} + \sqrt{g}$. On aurait alors à unir les deux DAGs suivants :

| | | | |
|---|---|---|---|
| $\sqrt{\boxed{1}} + \sqrt{\boxed{2}}$ | $a$ | $b + \sqrt{\boxed{3}}$ | $c$ |
| $\sqrt{\boxed{1}} + \sqrt{\boxed{3}}$ | $e + \sqrt{\boxed{2}}$ | $f$ | $g$ |

Unir ces deux DAGs donnerait un gros modèle alors qu'en permutant les éléments de la seconde ligne (et en modifiant les pointeurs en conséquence) on obtient deux DAGs triviaux à unir (et dont l'union est bien l'union attendue, de la forme $\sqrt{sq_1} + \sqrt{sq_2 + \sqrt{sq_3}}$

| | | | |
|---|---|---|---|
| $\sqrt{\boxed{1}} + \sqrt{\boxed{2}}$ | $a$ | $b + \sqrt{\boxed{3}}$ | $c$ |
| $\sqrt{\boxed{1}} + \sqrt{\boxed{2}}$ | $g$ | $e + \sqrt{\boxed{3}}$ | $f$ |

Il est donc essentiel de bien étudier toutes les permutations internes possibles des DAGs pour avoir le meilleur modèle commun [1].

1. Nous ne regardons en fait que les permutations qui font que les pointeurs pointent toujours vers la droite (vers un élément du DAG dont l'indice est plus grand que l'indice de l'élément actuel) afin de s'assurer de la caractéristique acyclique de notre graphe

### 2.3.4 Méthode générale

Finalement, nous mêlons les deux points précédents pour pouvoir unir efficacement un $m$-uplet de DAGs :

1. Rajout d'indéterminés si nécessaire pour que tous les DAGs aient la même longueur,
2. Génération de toutes les permutations internes des DAGs du $m$-uplets
3. Pour toutes les permutations
   (a) On remplace les indéterminés existants comme mentionné dans 2.3.1,
   (b) On unit les DAGs permutés et complétés tel qu'on le fait dans la sous-section 2.2.3,
4. Nous choisissons le modèle qui a la plus petite mesure comme modèle commun.

De cette manière, nous obtenons donc un modèle commun dont on est sûr qu'il ne fasse pas trop augmenter la taille du code transformé (nous ne considérons jamais le pire modèle).

## 2.4 Algorithme final

Ainsi, en assemblant les différents élément précédents, nous obtenons un algorithme permettant d'éliminer systématiquement les racines carrées et les divisions d'un programme : nous le mettons tout d'abord sous forme normale telle que définie à la section 1.2.2. Le programme obtenu est alors dans la classe de programme $P$. Nous regardons ensuite ce nouveau programme instruction par instruction et selon que l'instruction rencontrée est une déclaration ou un test, nous appliquons une des deux méthodes d'élimination décrites plus haut.

# 3 Optimisations de l'algorithme initial

L'algorithme que nous avons présenté précédemment présente tout de même certains défauts importants que nous allons présenter dans un premier temps, puis nous verrons comment améliorer l'algorithme pour les surmonter.

## 3.1 Modèles non optimaux

Nous pouvons trouver certains cas où les modèles que nous génère l'algorithme de la section 2. Prenons l'exemple de l'union de $(\sqrt{a}, \sqrt{a}, \sqrt{b})$ et de $(\sqrt{c}, \sqrt{d}, \sqrt{d})$. En effet, il nous donne comme modèle commun celui de la figure (6a) alors qu'il en existe un autre qui peut être plus intéressant dans certains cas, le modèle de la figure (6b).

| | | |
|---|---|---|
| $\left(\sqrt{\boxed{1}}, x_1\sqrt{\boxed{1}} + x_2\sqrt{\boxed{2}}, \sqrt{\boxed{2}}\right)$ | $sq_1$ | $sq_2$ |

(a) Partage d'une racine, modèle généré

| | | | |
|---|---|---|---|
| $\left(\sqrt{\boxed{1}}, \sqrt{\boxed{2}}, \sqrt{\boxed{3}}\right)$ | $sq_1$ | $sq_2$ | $sq_3$ |

(b) Duplication d'une racine, modèle non généré

FIGURE 6: Deux modèles possibles pour le triplet

On peut voir cela avec les deux exemples de code suivant qui sont très similaires :

Listing 5: Test sur les trois éléments

```
let (x, y, z) = if b > 0
    then (√a, √a, √b)
    else (√c, √d, √d)
in x + y + z > 0
```

Listing 6: Test sur un seul éléments

```
let (x, y, z) = if b > 0
    then (√a, √a, √b)
    else (√c, √d, √d)
in y > 0
```

Si pour le listing 5, le meilleur modèle commun (celui faisant intervenir le moins de racines distinctes) est le modèle (6a) - on obtient à la fin l'expression $(1 + x_1)\sqrt{sq_1} + (1 + x_2)\sqrt{sq_2} > 0$ - dans le cas du listing 6, le meilleur est le modèle (6b) - on a à simplifier uniquement $\sqrt{sq_2} > 0$ au lieu de $x_1\sqrt{sq_1} + x_2\sqrt{sq_2} > 0$.

## 3.2 Cassage de partage

Pour reprendre l'exemple précédent, si nous voulons générer le modèle (6b) avec un algorithme similaire à celui de la section 2, nous devons être capable, à partir des DAGs de départ, de construire un DAG avec une duplication sciemment insérée (cf. figure 7). On appelle cette procédure *cassage de partage*. En effet, elle

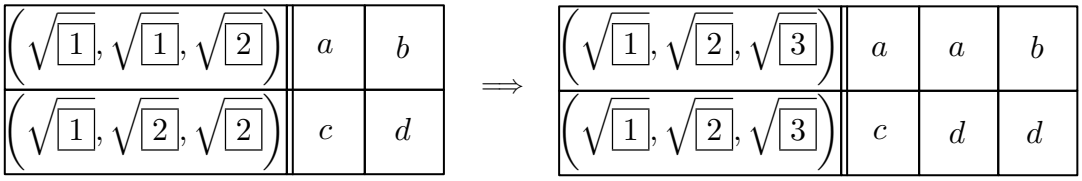



FIGURE 7: Exemple de cassage de partage

consiste à ne plus utiliser la fonction de partage des DAGs dans certains cas et donc à dupliquer des noeuds de ce DAG comme le montre la figure 8. Cette méthode permet d'augmenter le nombre de degrés de liberté lors de la création du modèle commun.

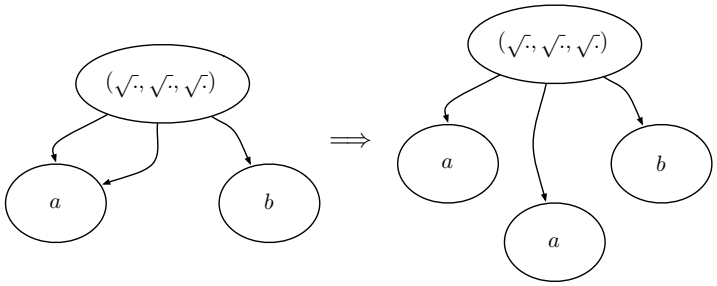



FIGURE 8: Cassage de partage sur un graphe

### 3.2.1 Intérêt du cassage

En raisonnant sur la représentation des DAGs, on peut observer qu'il est inutile d'effectuer du cassage lorsque l'on veut unir de simples expressions sans paires. Par contre, il devient utile dès que l'on a des

$n$-uplets à unir. En effet, la factorisation des expressions liée à la représentation sous forme de graphe se rapproche de la factorisation que l'on peut faire des racines carrés dans des expressions du type de (2.2.1).

Ce que l'on cherche à faire lorsque l'on casse le partage est de ne pas effectuer totalement cette factorisation. Or comme on désire limiter le nombre de racines dans le programme final, on ne peut laisser de côté la factorisation des expressions uniquement lorsque les expressions mathématiques correspondantes ne peuvent pas réellement se factoriser, c'est à dire dans le cas qui nous intéresse, en présence de paires dans les expressions.

### 3.2.2 Pratique du cassage de partage

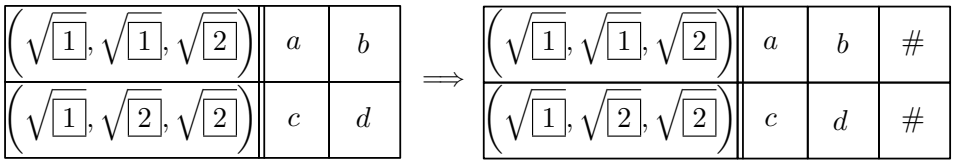
Comment faire pour arriver à casser correctement le partage des DAGs ? Pour cela, nous définissons une procédure systématique pour effectuer ce cassage.

**Definition 3.2.1** : *Procédure de cassage*
Soit $n$ le nombre de duplication de noeuds (de cassage) que l'on souhaite effectuer. On suppose que l'on a au départ $m$ DAGs qui sont tous de la même taille (qui ont le même nombre de noeuds). Si ce n'est pas le cas, nous ajoutons des indéterminés comme mentionné dans le paragraphe 2.3.1 et l'étape 1 de 2.3.4.

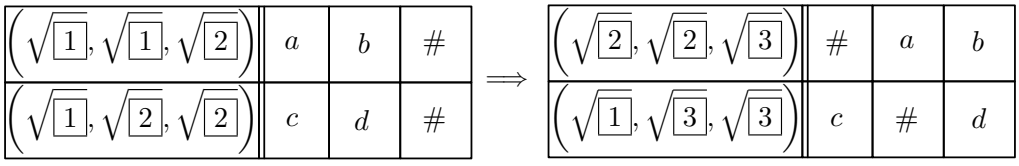
1. Ajout de $n$ indéterminés dans les DAGs (à la queue de la liste représentant chacun des DAGs)
2. Génération de toutes les permutations internes des $m$ DAGs (étape 2 de 2.3.4)
3. Pour toutes les permutations, on remplace les indéterminés existants soit par 0, soit par un élément constant (si possible, cf. 2.3.1), soit par un élément du DAG dont l'indice est supérieur à l'indice de l'indéterminé (un élément à droite dans la liste des noeuds).
4. Pour tous les nouveaux DAGs ainsi créés, nous générons tous les DAGs issus du remplacement d'un pointeur par un pointeur pointant vers un élément égal (un pointeur *équivalent*).

De cette manière nous générons de multiples $m$-uplets de DAGs qui représentent toujours les mêmes expressions et qu'il reste juste à unir *via* les règles de la définition 2.2.4.

Observons comment cette procédure nous permet d'obtenir la transformation voulue par la figure 7.



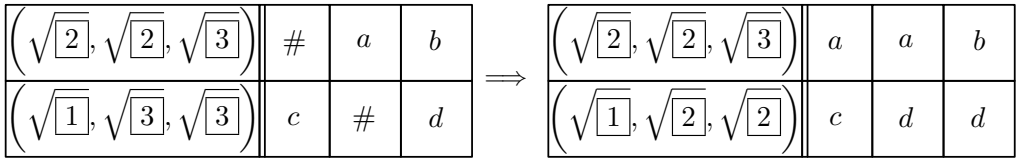

(a) Etape 1.



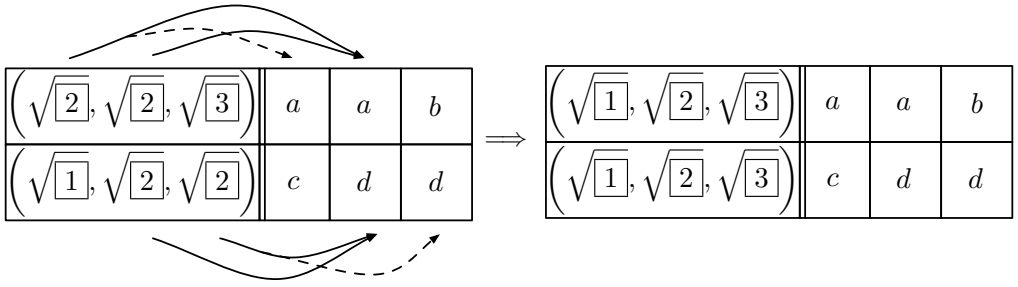

(b) Etape 2.

### 3.2.3 Nombre de duplications

Comme on l'a vu au 3.2.1, il n'est utile d'insérer des duplications que lors de l'union de paires. De même, il est inutile d'essayer de casser le partage lorsque les membres d'un $n$-uplet sont distincts deux à deux.




(c) Etape 3.




(d) Etape 4.

FIGURE 9: Etapes du cassage

Ainsi, si nous avons dans un premier temps décidé d'insérer autant de duplications que de paires dans l'expression (en gardant en tête que pour un programme bien typé, les paires sont en tête des expressions et qu'un $n$-uplet est représenté par $n-1$ paires imbriqués), nous nous sommes par la suite rendu compte qu'il suffit d'insérer autant de duplication que le maximum sur tous les DAGs du nombre d'éléments égaux moins 1.

Par exemple, pour unir $(\sqrt{a}, \sqrt{a}, \sqrt{b})$ et $(\sqrt{c}, \sqrt{d}, \sqrt{d})$, nous avons inséré une duplication (deux éléments égaux au maximum), et pour unir $(\sqrt{a}, \sqrt{a}, \sqrt{a})$ et $(\sqrt{b}, \sqrt{c}, \sqrt{d})$ nous en insérons deux (trois éléments égaux).

### 3.2.4 Algorithme avec cassage de partage

Grâce à cette procédure de cassage de partage, nous pouvons définir un algorithme dérivé du premier et qui génère bien plus de modèles communs, prenant en paramètre les $m$ DAGs à mettre sous forme commune et le nombre $n$ de duplications maximal à introduire :

1. Rajout d'indéterminés si nécessaire pour que tous les DAGs aient la même longueur
2. Pour $i$ allant de 0 à $n$, ajouter $i$ indéterminés à tous les DAGs
3. Génération de toutes les permutations internes des $m$-uplets de DAGs
4. Pour toutes les permutations, on remplace les indéterminés soit par 0, soit par un élément constant, soit par un élément situé à droite de l'indéterminé à remplir (cf. étape 3 du cassage).
5. Pour tous les $m$-uplets de DAGs créés, nous générons tous les DAGs possibles par changement de pointeurs équivalents.
6. On effectue l'union pour chacun des $m$-uplets créés à l'étape précédente

Il est essentiel de ne pas prendre ici le modèle qui a la plus petite mesure car celle-ci ne prend pas en compte le contexte dans lequel est utilisé le modèle : si nous utilisions la mesure dans l'exemple de la figure 6 nous ne sélectionnerons que le modèle (6b) et ne réglerions pas notre problème d'optimisation.

En fin de compte, nous générons donc tous les programmes possibles issus de tous les modèles communs possibles afin de ne sélectionner que le programme le plus petit.

## 3.3 Nombre de modèles communs générés

A cette étape, le nombre de modèles communs générés pose problème. En effet, si nous devons juste trouver le modèle commun de $m$ DAGs de même taille $l$ *via* la méthode de la section 2, nous devons déjà effectuer l'étude de $(l!)^m$ permutations internes. Lorsqu'en plus de juste effectuer l'union, nous devons aussi remplir les indéterminés, la combinatoire explose encore plus, chaque indéterminé pouvant être rempli par $l+1$ éléments (éléments de la queue du DAG et 0 - nous ne comptons pas l'ajout d'un élément constant). Ainsi, si nous voulons faire l'union de $m$ DAGs de taille $l$ avec $n$ duplications ajoutées, nous pouvons obtenir de l'ordre de $(l+n)!^m.l^{n.m}$ DAGs, sans compter les échanges de pointeurs ! Nous devons donc absolument réduire la quantité de modèles communs à regarder.

### 3.3.1 DAGs sans racines

Dans certains cas, nous pouvons avoir, parmi les expressions à unir, une ou plusieurs qui ne contiennent pas de racine. Leur représentation sous forme de graphe acyclique est donc triviale puisqu'elle ne comporte qu'un noeud-racine et aucune arrête, ainsi que leur représentation sous forme de liste qui ne contient qu'un seul élément.

Il est alors inutile de rajouter des indéterminés et d'effectuer des permutations sur ces indéterminés. Il est de même inutile de remplir ces indéterminés puisqu'aucun pointeur n'existe (leur valeur n'a aucune importance pour le DAG en question). Ainsi, on se limite à considérer uniquement les DAGs non triviaux, et on diminue la valeur de $m$.

En pratique, nous devons tout de même faire attention à l'insertion possible de constante lors de l'union des DAGs qui ne se fera pas si l'on remplace systématiquement les indéterminés par une valeur par défaut.

### 3.3.2 Colonnes d'indéterminés

Lorsque l'on insère des indéterminés pour ajouter des duplications dans les DAGs, après avoir effectué la permutation interne, nous pouvons nous retrouver avec une colonne d'indéterminés dans le $m$-uplet de DAGs à unir (c'est à dire qu'il existe un indice $i$ pour lequel tous les éléments d'indices $i$ des $m$ DAGs est un indéterminé) - c'est par exemple systèmatiquement le cas quand la permutation interne est l'identité. Dans ce cas, nous pouvons décider de systématiquement supprimer cette colonne dans le $m$-uplet et de poursuivre l'algorithme d'union.

De cette façon, il n'est plus nécessaire d'effectuer la boucle 2. de l'algorithme du 3.2.4 puisque les colonnes surnuméraires après permutations seront supprimées.

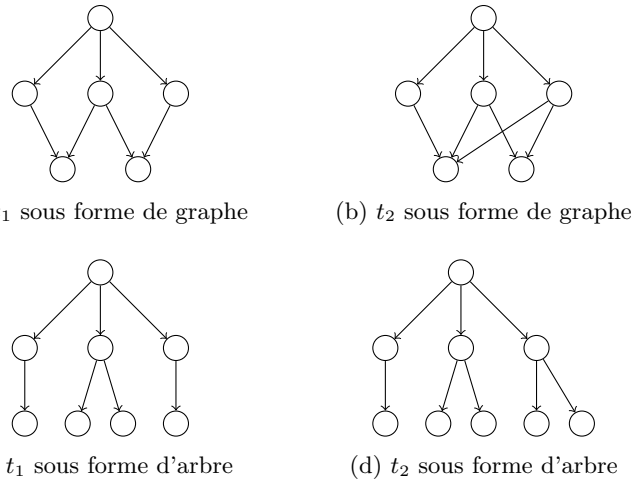
### 3.3.3 Modèles isomorphes et inclus

Comme on l'a vu à la section 2.2.1, les modèles ont une structure de graphe. Deux modèles dont la structure est isomorphe représenteront par conséquent des expressions qui auront la même sémantique selon le modèle utilisés. Il est par conséquent inutile de conserver deux modèles isomorphes.

Plus généralement, comme deux modèles communs $t_1$ et $t_2$ des mêmes expressions ne seront instanciés uniquement pour les représenter, si le graphe associé à $t_2$ contient le graphe associé à $t_1$, il suffit de garder le modèle $t_1$.

Toujours pour le même exemple des triplets $(\sqrt{a}, \sqrt{a}, \sqrt{b})$ et $(\sqrt{c}, \sqrt{d}, \sqrt{d})$, nous pouvons avoir les modèles $t_1 = \left(\sqrt{sq_1}, x_1\sqrt{sq_1} + x_2\sqrt{sq_2}, \sqrt{sq_2}\right)$ et $t_2 = \left(\sqrt{sq_1}, x_1\sqrt{sq_1} + x_2\sqrt{sq_2}, x_3\sqrt{sq_1} + x_4\sqrt{sq_2}\right)$. On voit très bien ici que $t_1$ est inclus dans $t_2$ ($t_2$ peut représenter toutes les expressions représentées par $t_1$), mais comme nous n'avons besoin que des seules expressions peut représenter $t_1$ (et non toutes celles de $t_2$), nous pouvons ne pas garder $t_2$.

Rechercher si un graphe $G_1$ est inclus dans un graphe $G_2$, c'est à dire s'il existe un graphe partiel $G_2'$ de $G_2$ isomorphe à $G_1$ est un problème NP-complet [**?**]. Or comme nous ne nous intéressons qu'à la sémantique de l'expression, nous pouvons transformer nos DAGs sous forme d'arbres pour lesquels il est aisé de déterminer l'inclusion. Il faut cependant aussi bien tenir compte du nombre de noeuds présents dans le graphe. Nous



(a) $t_1$ sous forme de graphe (b) $t_2$ sous forme de graphe

(c) $t_1$ sous forme d'arbre (d) $t_2$ sous forme d'arbre

FIGURE 10: Inclusion de modèles

ruinerions sinon tout notre travail précédent en supprimant des modèles issus de la duplications de noeud dans les DAGs.

Nous avons donc écrit une fonction permettant de détecter les inclusions d'arbres afin de ne garder que les modèles les plus petits au sens de l'inclusion.

Cette réduction est essentielle pour le fonctionnement de notre programme. En effet, pour ne pas avoir une quantité énorme de programmes issus de notre transformation à étudier, nous devons impérativement limiter le plus possible le nombre de modèles générés pour chaque déclaration et donc ne pas nous embarrasser de modèles redondants. En ne sélectionnant que les modèles les plus petits, nous réduisons drastiquement la quantité de modèles générés à chaque fois.

Pour donner un ordre d'idée, dans le cas de l'exemple du triplet de la section 3.1 avec insertion de une duplication possible, nous générons 188 modèles différents que nous réduisons à uniquement 3 modèles minimaux.

### 3.3.4 Insertion de constantes

Nous avons aussi inséré dans notre programme une option pour utiliser une heuristique qui peut se révèler avantageuse aussi bien au niveau du nombre de modèles générés à la fin que du temps passer à les générer. Nous pouvons en effet faire le choix d'insérer dès que possible des constantes dans les modèles communs.

Ce choix s'effectue au niveau du remplissage des indéterminés : lorsque nous pouvons remplir une colonne par une constante (c'est à dire que les éléments non indéterminés de la colonne sont tous égaux à un élément positif $b$), nous n'effectuons que ce remplissage, en laissant de côté les remplissages par 0 ou par des éléments de la ligne. A défaut de pouvoir mettre une constante, nous effectuons les autres remplissages.

## 3.4 Algorithme final - génération des programmes

Avec les optimisations du premier algorithme que nous venons d'exposer, nous obtenons donc un algorithme qui ne fait pas que retourner une seule racine mais qui retourne une liste de petits modèles pour lesquels il n'y a pas d'inclusion (nous n'avons pas réussi à montrer si ces modèles étaient minimaux).

1. Pour chaque DAG du $m$-uplet, calcul et ajout du nombre de duplications à insérer
2. Rajout d'indéterminés si nécessaires pour que tous les DAGs aient la même longueur
3. Pour toutes les permutations internes des $m$-uplets de DAGs, génération de tous les DAGs par cassage de partage

4. Union de tous les $m$-uplets issus de l'étape précédente
5. Sélection et renvoi des modèles minimaux

Finalement, lorsque nous faisons appel à notre outil sur un programme à transformer, nous allons générer tous les programmes possibles issus des différents choix de modèles communs. On voit encore ici qu'il est essentiel de limiter le nombre de modèles générés lors de chaque déclaration dépendant d'un test.

# 4 Optimisations de l'implémentation

Si nous avons essayé d'optimiser le plus possible la complexité de notre programme tout en gardant au maximum sa capacité à générer des modèles optimaux, nous avons dû aussi travailler sur l'implémentation de notre outil.

Il a ainsi été écrit en Caml, langage choisit pour son typage fort (qui permet d'éviter des erreurs d'implémentation mais aussi de faciliter la démonstration de la validité de l'outil) et sa capacité à travailler sur des structures récursives (ici les DAGs). Mais, afin de garder une vitesse compatible avec une utilisation réelle, il a fallu absolument optimiser la manière dont nous implémentions les algorithmes présentés plus haut.

Nous allons revenir sur deux points particuliers de cette optimisation : l'utilisation de fonctions récursives terminales et l'implémentation efficace d'une structure d'ensemble.

## 4.1 Fonctions récursives terminales

Lors de la génération des modèles, nous travaillons souvent avec des listes afin de profiter de leur flexibilité et de l'utilisation de fonctions récursives inhérentes aux langages fonctionnels. Cependant, la récursivité associée à la quantité de donnée que nous avons à traiter peut rapidement poser des problèmes d'optimisations et créer des dépassement de pile.

Il est alors souvent possible d'utiliser des fonctions récursives terminales qui évitent ce type d'erreurs à l'exécution. Il est tout de même important de veiller à la conservation de l'ordre de certaines structures (par exemple, il ne faut pas inverser l'ordre d'une liste codant un DAG) lors de l'utilisation de ces fonctions.

Le meilleur exemple concerne la génération des permutations internes : si dans un premier temps nous avions fait le choix de générer au début de l'algorithme toutes les $m$-uplets de permutations sur $\{1, 2, ..., n\}$ (c'est à dire $n!^m$ éléments) de manière récursive non terminale, nous avons par la suite, et à cause des problèmes de dépassement mémoire qui apparaissaient très vite, opté pour une génération itérative de ces permutations. Nous avons ainsi utilisé l'algorithme de Steinhaus–Johnson–Trotter [**?**] après avoir étudié le principe des bases factorielles [**?**] qui ne nous donnait pas satisfaction.

## 4.2 Hashset - une implémentation de la structure d'ensemble

Afin d'engendrer tous les modèles possibles, nous générons tout d'abord tous les $m$-uplets de DAGs dérivés de celui de départ (cf. 3.4).

Cependant, lors de cette génération, nous pouvons avoir beaucoup de redondance : les permutations internes suivies de changements de pointeurs équivalents peuvent aboutir aux mêmes DAGs. Il est par conséquent, au vu du nombre de DAGs générés, intéressant de pouvoir éliminer rapidement les doublons.

Dans un premier temps, dans un souci de simplicité, nous avions donc implémenté cette fonctionnalité *via* une liste dans laquelle nous ajoutions nos $m$-uplets si et seulement si ils ne faisaient pas de doublons. Il nous est venu très vite que cette méthode n'était pas du tout en adéquation avec notre volonté d'efficacité.

### 4.2.1 Structure de donnée

Nous avons donc implémenté la notion d'ensemble en utilisant des fonctions de hachage, un `hashset`. Pour cela, nous avons utilisé un vecteur de listes (des *buckets*) dans lequel nous insérons un élément $e$ de valeur de hachage $h(e)$ en l'ajoutant au bucket d'indice $h(e) \mod l$ (où $l$ est la longueur du vecteur) s'il n'y

est pas déjà présent. Lorsque le nombre d'élément présent dans le `hashset` est supérieur à $2l$, nous créons un nouveau vecteur de taille $2l$ et nous réinsérons les éléments un à un dans les nouveaux buckets qui leur correspondent.

Cette approche permet de limiter le nombre de comparaisons structurelle des éléments (ici des DAGs) aux seuls buckets, opération bien plus lourde que de calculer une valeur de hachage pour chacun de ces éléments.

Ainsi, si l'insertion d'un élément dans un `hashset` contenant déjà $n$ éléments peut se faire en $O(n)$ opérations dans le pire des cas, l'ajout de $n$ éléments dans un `hashset` se fait en $O(\log n)$ en moyenne.

### 4.2.2 Choix de la fonction de hachage et optimisations

Le principal problème auquel nous avons été confronté est le choix de la fonction de hachage qui se doit d'être la plus discriminante possible. Dans un premier temps, nous avons utilisé la fonction de hachage standard de Caml, qui permet de hacher n'importe quelle structure. Cependant, elle n'allait pas assez loin en profondeur de la structure lors du calcul et donc ne séparait quasiment pas les éléments. Une autre version permet de paramètrer la profondeur maximale et donc d'avoir une très bonne séparation, mais ses performances sont incompatibles avec notre désir d'efficacité. Nous avons donc créé notre propre fonction de hachage pour profiter de bonne performances aussi bien au niveau de la séparation qu'à celui de la rapidité.

Cependant, il n'y a pas de méthode systématique pour créer des fonctions de hachage et nous avons par conséquent procédé par tâtonnement.

Nous avons ainsi construit une fonction hachage qui tiens compte de la non-commutativité de certaines opérations (divisions, paires, ...). Nous avons ainsi haché les structures récursives telles que les listes ou les arbres d'expression selon le schéma suivant :

$$h(\text{liste}) = \text{tête} + k.h(\text{queue})$$

et choisi des valeurs de hachage pour les éléments terminaux de nos structures.

Nous nous sommes alors rendu compte que si la propriété de séparation semblait *a priori* remplie, lorsque nous utilisions notre fonction de hachage dans le cadre du `hashset`, les résultats étaient très décevants, avec de multiples collisions et de buckets vides. Nous avions en effet, un phénomène de résonance entre les valeurs de hachage et le nombre de buckets (la valeur de modulo). Pour le supprimer, nous avons donc pris des valeurs de $k$ et de $l$ premières.

En utilisant cette structure de donnée, nous arrivons à accélérer notre programme d'un facteur 5 dans le cas de DAGs de taille importante à unir.

### 4.2.3 Heuristique d'insertion

Lorsque l'on regarde un peu les statistiques de performance de notre outil, nous apercevons qu'une grande partie du temps CPU (un peu moins de 50%) est passé à comparer des DAGs pour voir s'il est possible de faire une insertion dans un bucket alors que dans le même temps, le nombre d'éléments effectivement insérés est très faible en comparaison du nombre de tentatives d'insertion (pour certains jeux de données, nous avons un taux de redondance supérieur à 1 pour 1000). Nous avons donc testé l'heuristique suivante en gardant en mémoire qu'en plus de la redondance stricte nous avons un phénomène de redondances lié aux inclusions de modèles (cf. 3.3.3) : nous supposons que la fonction de hachage est injective.

En pratique, cela implique que l'on insère un élément dans un bucket uniquement si celui-ci est vide. Cette heuristique augmente la vitesse de notre outil de 250% et même pour des jeux de données réduits (pour le triplet du 3.2 par exemple) nous générons à la fin toujours les mêmes modèles.

Ainsi, dans le cas de programme engendrant beaucoup de DAGs à étudier afin d'effectuer l'union optimale, nous pouvons *a priori* utiliser cette heuristique sans perdre en généralité au niveau du résultat.

# 5 Résultats et améliorations futures

## 5.1 Comparaison des deux algorithmes

Nous pouvons désormais comparer les deux algorithmes : le premier déjà existant et le second que nous avons mis au point. Le principal point de comparaison est la taille des programmes générés.

Nous allons tout d'abord comparer la taille du code généré par le premier algorithme et la taille du plus petit programme généré par le second algorithme. Dans un premier temps, nous allons comparer des programmes du type de celui du listing 6 mais avec plus de membres dans le $n$-uplet (les programmes pour les cas $n = 6$ et $n = 8$ sont décrits en annexe).

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Algo. 2.4 | 334 | 1375 | 1391 | 1960 | 12107 | 15111 | 15149 | 219885 |
| Algo. 3.4 | 334 | 369 | 406 | 444 | 1464 | 1502 | 1541 | 1597 |
| Facteur | 1 | 3.7 | 3.4 | 4.4 | 8.3 | 10.1 | 9.8 | 137.7 |

FIGURE 11: Taille (en octet) des programmes générés

Nous voyons bien que notre algorithme n'apporte aucun avantage dans le cas où nous n'avons aucun $n$-uplet. Etant données nos modification, cet état de fait était prévisible : si il n'y a pas de paires présentes dans le programme à transformer, notre algorithme amélioré n'apporte aucune différence avec l'algorithme initial. Sinon, dans le cas de $n$-uplets à unir, nous obtenons des résultats bien meilleurs qu'avec le premier algorithme.

Nous observons aussi que certaines heuristiques ne modifient pas la taille minimale des programmes trouvés comme par exemple, le choix de rajouter dès que possible des constantes (cf 3.3.4) qui par contre accélère grandement la vitesse de calcul. Par contre, le choix de supprimer des colonnes d'indéterminés peut empêcher certains modèles de se former lors du cassage de partage.

Dans le même ordre d'idée, nous remarquons, lorsque nous traitons des jeux de données nécessitant un gros traitement de la part de l'outil, que les modèles minimaux sont générés assez tôt dans l'énumération des permutations (à cause du taux de redondance mais aussi du fait que l'on fait bien attention au départ à ne regarder que des permutations engendrant des DAGs avec des dépendances à droite et donc la majorité des dernières permutations n'est pas utilisée). Nous pouvons par conséquent essayer de limiter le nombre de permutations énumérées lors de la génération des modèles *via* une option laissée à l'utilisateur.

Enfin, grâce à la grande redondance des modèles générés, nous pouvons aussi ne considérer qu'une fraction des permutations tout en gardant d'excellents résultats. Par exemple, les modèles générés par notre algorithme pour le cas $n = 6$ ont été obtenus en ne gardant qu'un pourcent des permutations, qu'une sur 1 000 000 pour le cas $n = 7$ et une sur 100 000 000 pour $n = 8$.

## 5.2 Travail futur

### 5.2.1 Nouvelles fonctionnalités

Nous nous sommes limité à la transformation des programmes tels que définis dans la section 1. La suite immédiate est d'arriver à adapter l'algorithme que nous avons construit à la transformation de programmes contenant des fonctions. Une fois cette étape franchie, nous pourrons alors transformer de manière assez simple des boucles finies (dont le nombre d'itération est définie à la compilation). Nous aurons alors toutes les fonctionnalités nécessaires à la transformation de la plupart des codes pour systèmes embarqués.

### 5.2.2 Performances

Etant donné que les algorithmes que nous avons présentés sont très parallèlisables, il pourrait être intéressant d'effectuer une parallèlisation massive de notre programme. Malheureusement, le choix du lan-

gage, Caml, nous a limité sur ce point d'implémentation. Nous pourrions tout de même essayer de porter notre programme dans un autre langage (par exemple en C).

Nous nous sommes tout de même efforcé à accélérer le plus possible notre programme, par exemple en le compilant nativement, ainsi qu'a limiter le plus possible la mémoire qu'il nécessite (une centaine de méga-octets tout au plus pour la déclaration multiple de deux 6-uplets générant plusieurs dizaines de millions modèles différents avant réduction).

### 5.2.3 Dépendance vis-à-vis du programme source

Nous nous sommes aperçu que les performances de notre outil étaient très dépendantes la manière dont le programme à transformer était écrit. Cela est lié au fait que l'on puisse éviter de faire des tests imbriqués dans certains cas. En effet, l'imbrication de tests augmente le nombre de DAGs à unir et donc augment le nombre de cas à étudier de manière exponentielle. Il faudrait ainsi mettre au point un méthode pouvant *désimbriquer* les tests lorsque cela est possible.

De même il pourrait être intéressant de déterminer les dépendances des définitions entre elles afin de *factoriser* les déclarations dépendantes d'un test et éviter de refaire les mêmes calculs de modèle commun.

# Conclusion

Au départ, si nous avions un algorithme systématique de suppression de racines et de divisions dans un programme, celui-ci n'était que peu adapté à une utilisation pratique dans le cadre de programmes embarqués. Nous avons donc effectué une étude de ce qui pourrait améliorer les résultats de cet algorithme. Nous nous sommes aperçu qu'un des principaux points d'amélioration serait l'étude des graphes acycliques dirigés et des transformations que nous pouvions leur apporter.

Une fois ces modifications formalisées, nous les avons implémentées dans un programme dérivé de l'algorithme de départ, puis optimisées, aussi bien au niveau théorique, qu'au niveau de l'implémentation et enfin nous avons développé certaines heuristiques avec un certain succès afin d'augmenter le plus possible les performances de notre outil sans en détruire l'intérêt.

Grâce aux améliorations que nous avons apportées, la méthode de transformation des algorithmes destinés aux systèmes embarqués développée dans [**?**] est désormais envisageable en pratique. Il reste cependant à concevoir la transformation des fonctions pour que l'on puisse traiter la quasi-totalité des cas de figures que nous pouvons rencontrer dans les programmes embarqués.

Mon travail sur ce projet m'a ainsi permis de travailler sur un nouveau sujet, la transformation de programmes, différent de ceux que j'avais pu étudier durant mon programme d'approfondissement. Plus généralement, j'ai eu l'occasion de travailler avec une équipe travaillant sur les preuves de programmes et ainsi découvrir de nouveaux domaines de recherche et de réflexions en informatique. J'ai pu tout de même mettre en œuvre mes connaissances acquises en algorithmique dans un certain nombre de situations qui se sont révélées essentielles dans le bon fonctionnement final de notre outil de transformation de programmes.

Je tiens enfin à remercier Gilles Dowek et Pierre Néron pour leur aide, leur intérêt et leur soutien tout au long de mon stage. Je voudrais aussi remercier toute l'équipe DEDUCTEAM pour leur accueil durant ces 3 mois et demi de stage dans leurs murs.

# Annexes

Listing 7: $n = 6$

```
let (x, y, z, t, u, v) =
if a > 0 then
  (√a, √a, √b, √b, √c, √c)
else
  (√d, √e, √e, √f, √f, √d)
in y + v > 4 ; ;
```

Listing 8: $n = 8$

```
let (x, y, z, t, u, v, w, o) =
if a > 0 then
  (√a, √a, √b, √b, √c, √c, √d, √d)
else
  (√e, √f, √f, √g, √g, √h, √h, √e)
in y + v > 4 ; ;
```

# Table des figures