Embedded-Scilab : un compilateur Scilab pour plate-formes embarquées

Mathias Fleury Benjamin Rouxel Corentin Hardy Claude Stolze Pierre Vannier
mathias.fleury@ens-rennes.fr *benjamin.rouxel.1@etudiant.univ-rennes1.fr* *corentin.hardy@ens-rennes.fr*
claude.stolze@ens-rennes.fr *pierre.vannier@etudiant.univ-rennes1.fr*

ÉNS Rennes, Université Rennes 1

Résumé Scilab est un langage très populaire pour le prototypage d'application. Il est cependant impossible d'utiliser ces prototypes sur des architectures embarquées. En effet, ceux-ci ne peuvent embarquer un interpréteur Scilab, entre autre car l'interprétation est généralement moins efficace que l'exécution d'un code compilé en langage machine. Une solution est donc de compiler préalablement Scilab en C. Scilab étant un langage à typage dynamique, une des étapes cruciales de la compilation est l'inférence de type des variables. Ce document introduit les concepts nécessaires de la compilation, puis présente des solutions pour la compilation de Scilab en C dans le but d'obtenir un code efficace, tant en temps d'exécution qu'en occupation mémoire.

1 Introduction

Scilab (comme Matlab, dont il s'inspire) est un langage de programmation orienté vers la manipulation de matrices. L'un de ses domaines d'application usuels est le traitement d'images et de signal. Scilab est très apprécié pour la phase de prototypage d'une application embarquée car il offre une grande facilité pour la réalisation de calculs complexes.

Le langage Scilab est interprété, son typage est dynamique et implicite, ce qui lui confère une grande souplesse d'utilisation. Le type d'une variable peut évoluer au cours de l'exécution d'un programme. Cependant, l'exécution d'un programme interprété nécessite généralement plus de temps que l'exécution d'un programme compilé en langage machine, et ce indépendamment du langage source. Une fois l'étape de prototypage d'une application embarquée terminée, il est nécessaire de réécrire le code Scilab dans un langage compilable tel que le C, afin d'avoir un code plus efficace. Une alternative, plus rapide, consisterait à réaliser cette réécriture de manière automatique grâce à un compilateur Scilab vers C.

La principale différence entre ces deux langages concerne le typage des variables : il est statique en C. Une approche naïve, suivie par les premières générations de compilateurs Matlab, est de compiler vers un code C gérant dynamiquement toutes les situations possibles de type, ou de définir un type générique convenant à toutes les situations. Cela produit du code peu performant et difficile à optimiser par le compilateur C. C'est pourquoi l'étape de compilation de Scilab doit déterminer statiquement les types des variables et des expressions pour être efficace. Le code ainsi généré est plus proche de celui produit à la main. L'inférence de type est donc une étape critique pour produire un code performant.

De nombreux travaux se sont intéressés aux problèmes de compilation et de parallélisation automatique de programmes Matlab vers des langages compilés (Fortran ou C). Nous nous intéresserons ici au problème beaucoup moins étudié qu'est la génération de code pour des architectures embarquées, alors que les contraintes y sont plus fortes, ces architectures n'ont ainsi pas nécessairement d'unité arithmétique flottante, et la taille de leurs mémoires est plus limitée. Le but de ce projet est de mettre en œuvre un générateur de code C efficace à partir du langage Scilab autour de l'infrastructure Gecos et de son analyseur de code Scilab. Ce compilateur sera appliqué à un dispositif de stéréo-vision sur une architecture embarquée.

Dans ce rapport d'étude, nous commencerons par discuter des différents problèmes liés à la transformation de code Scilab en code C. Dans une deuxième partie, nous présenterons plusieurs méthodes permettant d'inférer un type à une variable. Puis, nous proposerons des méthodes permettant de réorganiser la représentation intermédiaire de Gecos en vue d'optimiser le code produit. Enfin, reprenant les solutions proposées, la réalisation dans la prochaine étape du projet de celles-ci sera présentée.

2 Transformation de Scilab vers C

2.1 Présentation de Scilab et difficultés de la traduction en C

Comme énoncé précédemment, Scilab est un langage de programmation dynamiquement typé. Les matrices et des algorithmes pour les manipuler sont implémentés d'origine, ce qui explique son utilisation pour le prototypage dans les domaines des télécommunications et du traitement d'images. C'est un langage interprété et le type des variables est déterminé lors de l'exécution. Scilab est dit structuré car il ne contient pas de branchement inconditionnel (`goto`). Il ne permet pas non plus de manipuler des pointeurs de variables et tous les arguments des fonctions sont passés par valeur. Scilab et son interpréteur, inspirés de Matlab, ont été créés par l'IRIA (Institut de Recherche en Informatique et en Automatique) dans les années 80, ensuite développé par INRIA [1], et maintenant maintenus par Scilab Entreprise [2] [1].

L'approche dynamique du typage des variables est permise par un polymorphisme des opérateurs. Par exemple, pour $\mathcal{M}$ l'ensemble des matrices (y compris vecteurs lignes et colonnes), les opérandes du produit `A=B*C` peuvent être :

— `B`, `C` $\in \mathcal{M}$, et `A` $\in \mathcal{M}$,
— `B` $\in \mathbb{R}$, `C` $\in \mathcal{M}$, et `A` $\in \mathcal{M}$,
— `B`, `C` $\in \mathbb{R}$, et `A` $\in \mathbb{R}$.

La détection du type du résultat et le choix de l'opérateur se font donc grâce aux types des opérandes. Ainsi si `B` (ou `A`) est une matrice de complexe, alors `C` l'est aussi : c'est pourquoi le typage d'une expression comme `A=B*C` est impossible sans contexte. Un programme de quelques lignes tel que celui de la Figure 1a peut se révéler difficile à traduire en C à la main, et les types de ses variables ne sont pas triviaux à déterminer, comme illustré sur l'exemple ci-dessous.

```
1  function R = compagnon(P)
2      S = size(P);
3      if S(1,2) ~= 1 then
4          P = P';
5          S = S(2:-1:1);
6      end;
7      S = S(1,1)-1;
8      R = eye(S,S);
9      R(2:S+1,1:S)=R;
10     R(1,1)=0;
11     R(1:S+1,S+1) = -P(1:S+1, 1);
12 endfunction
```

(a) Fonction de calcul de la matrice compagnon d'un polynôme

```
1  if (B.type == Matrix) {
2    if(C.type == Matrix)
3        matrixMatrixMult(B,C)
4    else //C.type = Scalaire
5        matrixScalaireMult(B,C)
6  }
7  else {
8    if(C.type == Matrix)
9        matrixScalaireMult(B,C)
10   else //C.type == Scalaire
11       B*C
12 }
```

(b) Traduction de Scilab naïve pour l'instruction $A = B * C$

FIGURE 1: Présentation d'un code Scilab, et d'une traduction non efficace d'une instruction en C

Pour le lecteur qui ne connaît pas Scilab, quelques explications s'imposent sur la Figure 1a :

— le paramètre formel de la fonction `compagnon` est une variable `P` de type inconnu ;
— la valeur retournée est celle de la variable `R` de type inconnu ;
— à la ligne 2, `size(P)` retourne un vecteur ligne qui contient les dimensions de `P` ; notons que les scalaires sont, pour Scilab, des matrices 1×1 ;
— la fonction `eye(x,y)` de la ligne 8 retourne une matrice identité de x lignes et y colonnes ;

1. `http://www.inria.fr/`
2. `http://www.scilab-enterprises.com/`

— à la ligne 4, l'opérateur ′ retourne la transconjuguée (conjugaison de la transposée ou adjoint) d'une matrice.

Comme illustré par les explications suivantes, les éléments d'une matrice sont accessibles de plusieurs façons :

— à la ligne 3, si `S` est une matrice alors `S(1,2)` correspond à l'élément de la ligne 1, colonne 2. Remarquons que si `S` est une fonction alors, dans ce cas, `S(1,2)` correspond à un appel de fonction avec 1 et 2 pour paramètres effectifs ;
— dans le cas de la ligne 9, `R(2:S+1, 1:S)` se réfère à la sous-matrice de `R` contenant les éléments de la ligne 2 à `S+1` et de la colonne 1 à `S`, où `S` est un scalaire. Dans le cas ou l'index dépasse la taille de `R`, Scilab redimensionne automatiquement `R` ;
— à la ligne 5, l'instruction `S(2:-1:1)` retourne une copie des deux premiers éléments de la matrice `S`, dont l'ordre a été inversé.

Plusieurs éléments difficiles à traduire apparaissent ici :

— la variable `S` correspond tout d'abord à un vecteur colonne (ligne 2) puis à un scalaire (ligne 7) ;
— le langage C ne contient pas d'opérateurs permettant d'accéder à plusieurs éléments d'un tableau, il sera donc nécessaire d'utiliser des boucles pour traduire les lignes 5, 9, et 11 ;
— de façon similaire, il sera nécessaire d'utiliser des variables intermédiaires lors de la traduction des lignes 5 et 9, afin de ne pas modifier les éléments de la partie droite de l'affectation ;
— à l'exécution des lignes 8, 9 et 11, la matrice `R` est de taille `(S,S)`, puis `(S+1,S)` et enfin `(S+1,S+1)`, Scilab redimensionnant dynamiquement cette matrice.

De plus, même si les dimensions de la matrice `P` sont connues au début du programme, une analyse simple, qui ne tiendrait pas compte de la sémantique de `size`, ne permet pas de savoir si la ligne 4 s'exécute, change `P` en sa transconjugué si $S_{1,2} \neq 1$. Après l'instruction conditionnelle (ligne 7), les dimensions de `P` ne peuvent plus être déterminées statiquement, et donc `P(1:S+1,1)` n'est peut-être pas une expression sémantiquement valide. L'index `(1:S+1,1)` dépasse peut-être la taille de `P`. Mais le problème le plus important est l'indéterminisme du type intrinsèque des matrices manipulées, selon leur contenu (des entiers, des flottants ou des complexes) elles correspondent à trois implémentations différentes en C.

Une des difficultés de la compilation est donc de déterminer le type de toutes les variables du programme Scilab afin de ne pas avoir à générer une gestion dynamique des différents types possibles dans le code produit, présenté Figure 1b. C'est ce que fait le compilateur actuel de Scilab vers C, comme présenté sur l'exemple 1b, est une approche naïve regarde les différents types possibles :

— si `B` est une matrice, alors :
  — si `C` est une matrice, alors c'est le produit matriciel que l'on doit choisir ;
  — si `C` est un scalaire, alors il faut multiplier chacun des éléments de `B` par `C`
— si `B` est un scalaire, alors :
  — si `C` est une matrice, il faut multiplier chacune des éléments de `B` par `C` ;
  — si `C` est un scalaire, alors c'est simplement le produit usuel.

Catégorisation des fonctions et opérateurs du langage

De Rose [2] et Joisha [3] classent les opérateurs et fonctions internes du langage en catégories :

— la catégorie *Type-I* répertorie les fonctions dont le type du résultat est complètement connu une fois le type des opérandes connus (ex : addition, multiplication, ...) ;
— la catégorie *Type-II* répertorie les fonctions qui ne sont pas dans le *Type-I* et dont le type du résultat est complètement connu une fois le type de ces arguments élémentaires connus (ex : $c \leftarrow a : b$, si a et b scalaire et $a \leqslant b$, alors c vecteur ligne)
— la catégorie *Type-III* répertorie les fonction qui ne sont pas dans les catégories précédentes et dont le type du résultat ne peut être connu même en connaissant pleinement le type de ces arguments (ex : `eval`) ;

2.2 Représentation de programme

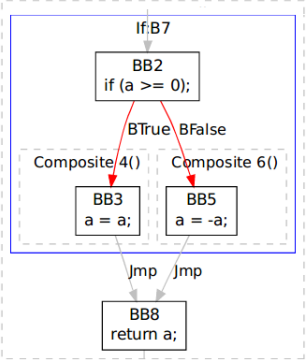
Afin de compiler du code Scilab de façon intelligente et éviter le surcoût lié à ce type de gestion, nous allons raisonner sur la structure du programme. Pour cela, nous pouvons utiliser plusieurs représentations

sous forme de graphes du même programme. C'est le niveau de détails nécessaire qui déterminera le choix de la représentation.

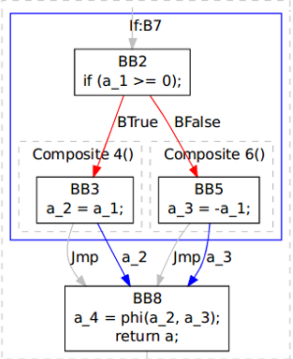
Code C :

```
1  int abs(int a){
2      if (a >= 0)
3        a = a;
4      else
5          a = -a;
6      return a;
7  }
```

(a) Fonction de calcul de la valeur absolue d'un entier (32 bits).



(b) CFG du programme présent sur la Figure 2a

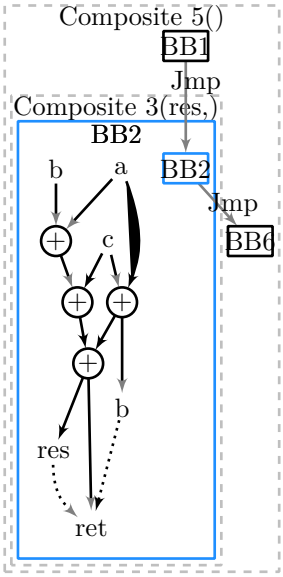


(c) Forme SSA du CFG que l'on voit à gauche sur la Figure 2b

FIGURE 2: Différentes représentations d'un programme simple.

```
1  int addertree(int a, int b
       , int c) {
2      int res = a + b +c;
3      b = a + c; res += b;
4      return res;
5  }
```

(a) Code C



(b) DAG associé

FIGURE 3: Exemple de DAG associé à un code C

L'une des premières étapes d'un compilateur est de créer une *représentation intermédiaire (RI)* sur laquelle trouver des analyses et optimisations sont plus simples. Une première possibilité est l'*arbre de syntaxe abstrait (AST)* permettant l'analyse d'un programme par un parcours d'arbre. Il est aussi possible de représenter une RI sous forme de graphe grâce au *graphe de flots de contrôle (CFG)* proposé par Allen [4]. Il s'agit d'un graphe reliant les instructions du programme, dont les arêtes sont les branchements ou séquences du programmes. Ce graphe possède un unique nœud sans prédécesseur (resp. sans successeur) : c'est le *point d'entrée* (resp. *point de sortie*) du graphe de flot de contrôle. Ces deux représentations exhibent les chemins d'exécution possibles à l'intérieur du programme (cf figure 2b) et permettent ainsi de matérialiser les dépendances entre les blocs. Ce faisant, il devient possible d'étudier les propagations de contraintes sur les types, par exemple.

Une autre représentation plus intéressante proposée par CYTRON et al [5] est la forme *static single assignment (SSA)* qui vient enrichir le CFG.

Cette représentation repose sur la particularité qu'aucune variable n'est jamais ré-assignée ; si le programme source comporte une réassignation de variable, alors la variable se voit renommée dans la seconde assignation. Dans le cas d'une variable assignée dans de deux branches du flot de contrôle, une fonction ϕ est introduite. Cette dernière permet de retourner la valeur de l'un de ces arguments choisis parmi ceux ayant été assignés par l'une des branches conditionnelles (cf figure 2c dans le dernier bloc).

La représentation sous forme de graphe de SSA, et le côté structuré de Scilab (sans `goto`) impliquent que les fonctions ϕ sont toujours l'intersection de deux arcs.

Une représentation qui montre les dépendances au sein des expressions arithmétiques se matérialise sous forme de *graphe acyclique dirigé (DAG)*.

L'exemple 3 montre la dépendance des données calculées, `b` a changé entre sa première utilisation (flèche pleine) et sa seconde utilisation (flèche pointillée).

Toutes ces différentes représentations jouent un rôle dans les différentes étapes de compilation. Par exemple, la forme SSA d'un CFG sera utilisée pour inférer le type des variables.

Propagation des informations dans un graphe
Il y a deux méthodes de propagation d'informations :

en avant celle-ci consiste à propager de l'information depuis les nœuds pères, vers les nœuds fils, l'ordre d'exécution des instructions du programme ;

en arrière elle remonte de l'information depuis les instructions suivantes vers les instructions précédentes.

3 Le système de typage

Une fois la représentation intermédiaire du programme terminée, celle-ci peut être utilisée dans l'optique de typer au mieux les expressions présentes. Le type d'une expression regroupe les informations du type intrinsèque de l'expression (réel, entier,...) mais aussi sa forme (*shape*) et sa taille. Le rôle de l'inférence de type est de déterminer les types d'expressions simples, comme les constantes, puis de propager ces informations dans le programme pour typer le plus finement possible toutes les autres expressions.

Cette section présente les méthodes d'inférence des types intrinsèques, de la forme et de la taille des expressions. Ensuite des propositions seront apportées afin de modérer les échecs de l'inférence statique du type d'une expression.

3.1 Typage intrinsèque

Différentes méthodes énoncées dans les articles de De Rose[2] et Almási[6] vont ici être présentées afin de décrire une technique pour inférer un type intrinsèque à chaque expression du programme.

Bien que Scilab n'utilise que les *réels* et les *complexes*, le système d'inférence des types intrinsèques doit gérer les types *entier* et *booléen* afin d'offrir de meilleures performances. De Rose[2] établit donc la relation d'ordre $\prec$ suivante :

$$boolean \prec integer \prec reel \prec complex$$

Cela signifie qu'un nombre entier peut être représenté sans perte de précision par un nombre complexe mais pas l'inverse.

Un ensemble finis d'éléments (ici les types) muni d'un ordre partiel complet *CPO* ($\prec$) est appelé un treillis. Comme l'ensemble est finis, le CPO permet de définir un élément maximum et un minimum, le treillis est donc de hauteur finie. Il s'agit donc d'une résolution du treillis complet (*complet lattice*) du type intrinsèque des variables qui peut être appliqué sur le CFG du programme afin d'inférer le type intrinsèque des variables.

Propagation d'informations par analyse de flot de données Cette approche s'inscrit dans le cadre plus général de l'analyse de flot de données (*data-flow analysis*). Pour cela, il est nécessaire de définir des règles permettant la propagation d'information de type. De Rose[2] fournit un opérateur $\sqcap$ (*meet*) permettant de déterminer le type résultat d'une expression en fonction du type des opérandes et de l'opérateur. Dans son article il définit donc un tableau par opérateur recoupant le type des opérandes et le type à choisir pour le résultat. Ces types intrinsèques étant propagés sur le CFG sous forme SSA du programme source, il est nécessaire de définir la règle à utiliser pour l'opérateur $\sqcap$ dans le cas du résultat de la fonction Φ, De Rose [7] définit donc le tableau de correspondance 1 page suivante.

Pour appliquer ces règles il s'agit de considérer par instruction, les informations de type disponibles en entrée In, et les informations en sortie Out. Par définition, l'utilisation d'une propagation en avant amène la création d'équations définissant l'ensemble Out en fonction de l'ensemble In, dont voici une généralité :

$$\mathrm{Out}\, s = \prod_{p \text{ prédecesseurs de } s} \mathrm{In}\, p$$

Type of first parameter	Type of the second parameter					
	null	logical	integer	real	complex	unknown
null	null	logical	integer	real	complex	unknown
logical	logical	logical	integer	real	unknown	unknown
integer	integer	integer	integer	real	unknown	unknown
real	real	real	real	real	unknown	unknown
complex	complex	unknown	unknown	unknown	complex	unknown
unknown	unknown	unknown	unknown	unknown	unknown	unknown

TABLE 1: Règle d'application de l'opérateur $\sqcap$ pour le résultat des fonctions ϕ

L'algorithme de propagation des informations le long des nœuds possède les garanties suivantes [8] :

1. si l'algorithme converge, une solution est trouvée ;
2. si les fonctions sont croissantes, alors la solution est maximale ;
3. si les fonctions sont croissantes et si le treillis est de hauteur finie, alors l'algorithme converge.

Inférence de type intrinsèque grâce aux treillis Dans l'exemple de la figure 4, la variable `A` se voit inférer le type entier lors de son initialisation, `B` le type réel. Ce qui donne pour l'équation de type de `C` : type(`C`) = type(`A`) $\sqcap$ type(`B`). Le type du résultat de `C` sera donc un nombre réel conformément aux règles arithmétiques.

Dans le cas d'un programme comportant une boucle, l'algorithme d'inférence de type est répétée de manière itérative jusqu'à l'obtention d'un point fixe. Ce dernier est toujours atteint car le treillis défini ci-avant est composé d'une fonction croissante sur l'ensemble finis des types.

$\texttt{A} = 15$
$\texttt{B} = -0.3$
$\texttt{C} = a \times b$

FIGURE 4: Exemple de propagation

1 $x_1 = 2$
2 `do`
3 $\quad x_2 = \phi(x_1, x_3)$
4 $\quad x_3 = (x_2^2 + 2)/(2 * x_2)$
5 **`while`** $x_3^2 - 2 > 0.001$

FIGURE 5: Code SSA avec boucle

L'exemple de la figure 5, x_1 est défini comme entier. Lors de la première itération de la boucle, x_2 est un entier conformément aux règles de l'opérateur $\sqcap$ énoncées par De Rose[2], et parce que le type de x_3 est inconnu à l'appel de la fonction ϕ. À l'instruction suivante x_3 est défini comme un nombre réel. Lors la seconde itération, type(x_2) = type(x_1) $\sqcap$ type(x_3) = *réel*. Une troisième itération de la boucle laisse les types inchangés, le point fixe est atteint.

Une fois le type intrinsèque déterminé, il est maintenant nécessaire de passer à l'étape suivante de recherche de forme et de taille pour compléter l'inférence de type des variables.

3.2 Inférence de formes et tailles

Au cours des années précédentes, plusieurs travaux de recherche ont été menés sur le sujet notamment par Wand [9], De Rose *et al* [7][2], Joisha [10][11][3], Almási [6]. Cependant les premiers se sont révélés incomplets, et des explications concernant ces limites seront proposées avant que ne soit décrite la solution la plus précise.

Le terme *shape-tuple* sera utilisé pour se référer à la représentation mathématique de forme. Celui-ci est un tableau `t` par un « tuple » $\texttt{t}^k = \langle \texttt{t}_1, \texttt{t}_2, ..., \texttt{t}_\texttt{k} \rangle$, où `k` est la dimension de `t`, et $\texttt{t}_\texttt{i}$ est la taille de la dimension `i`.

```
1  function S = quadrature(a,b)
2      h = (b-a)/2;
3      mid = (a+b)/2
4      fa = 13.*(a-a.^2).*exp(-2.*a./2);
5      Fmid = 13.*(mid-mid.^2).*exp(-3.*mid./2);
6      Fb = 13.*(b-b.^2).*exp(-3.*b./2);
7      S = h*(fa+4*Fmid+Fb)/3
8  endfunction
```

(a) Calcul de $\int_a^b 13(x-x^2)e^{\frac{-3x}{2}}\mathrm{d}x$ avec la règle de Simpson [3]

```
1  function unsafe()
2      x = rand(3,2)
3      a = rand()
4
5      while p
6          a = [a, rand()]
7      end
8      y = a*x
9  endfunction
```

(b) Unsafe shape function [3]

FIGURE 6: Exemple de code exposant les limites de l'utilisation des treillis

Limite des approches basées sur les treillis De Rose et Padua ont choisi de définir un treillis sur les tuples dans leur compilateur Matlab/Fortran Falcon[2]. Une utilisation similaire des treillis a été mise en place dans le compilateur MaJIC présenté par Almási *et al* [6]. Cependant Joisha *et al* [3] démontrent que même si cette solution permet de toujours générer du code fonctionnel, elle est parfois peu efficace. Dans la fonction présentée par la figure 6a, Falcon attribue la forme la plus générale ($\langle$UNKNOWN, UNKNWON$\rangle$) aux variables locales, alors que les variables `h`, `mid`, `fa`, `Fmid`, `Fb`, et `S` sont de même taille. Il est aussi possible de voir que les arguments a et b sont des matrices carrées.

De plus, le treillis choisi pour l'inférence de forme et taille, n'est pas de hauteur finie, donc dans une boucle une variable peut grossir incrémentalement un nombre inconnu de fois sans que l'on atteigne un point fixe. Dans la figure 6b, MaJIC réalise un nombre fixe et arbitraire d'itérations de la boucle (ligne 5), mais la forme de a ne converge pas vers un point fixe. La forme la plus générale définie dans le treillis ($\langle$UNKNOWN, UNKNWON$\rangle$), est alors choisie pour a. Cependant, ce choix amène un non-respect des règles de multiplication matricielle de la ligne 8, résultant en un programme mal-formé.

Utilisation d'un système algébrique La création d'un système algébrique revient à Joisha *et al* [10]. Il permet de définir une structure de données, ainsi qu'un ensemble de règles permettant de déterminer, de façon sûre, la forme et un majorant de la taille d'une variable. La suite définira ce système ainsi que la façon de l'utiliser dans un cas réel.

Les axiomes du système

Les axiomes du système algébrique sont représentés par la notion de *shape-tuple*. Pour des considérations d'ordre pratique, toute variable est vue comme un tableau à deux dimensions minimum. Un tuple ne pourra donc pas avoir moins de deux éléments. Des dimensions de taille 1 (extension unité) sont ajoutées par la droite d'un tuple pour satisfaire cette contrainte. Ces extensions sont sans signification effective, et peuvent donc être ajoutées sans changer la forme d'une variable. Voici quelques exemples :

— scalaire : $\langle 1, 1\rangle$;
— vecteur colonne de 2 éléments : $\langle 2, 1\rangle$;
— vecteur ligne de 4 éléments : $\langle 1, 4\rangle$, équivalent à $\langle 1, 4, 1\rangle$ ou $\langle 1, 4, 1, 1, 1, 1\rangle$;
— matrice 2×3 : $\langle 2, 3\rangle$;
— matrice $2 \times 3 \times 4$: $\langle 2, 3, 4\rangle$ équivalent à $\langle 2, 3, 4, 1, 1, 1\rangle$.

À cette notation, Joisha *et al* [10] ajoutent la notion de forme canonique qui est la représentation minimale d'un tuple. Pour l'obtenir, il suffit de retirer les extensions unités. Attention cependant à respecter la contrainte de deux éléments minimum. Le nombre d'éléments dans une forme canonique exprime le rang canonique d'un tuple. Avec cette nouvelle notion, une relation d'équivalence peut être émise : toute forme est équivalente à une autre, si elles sont représentées par la même forme canonique.

La figure 7 révèle la représentation numérique d'un tuple par une matrice diagonale carrée de dimensions $n \times n$. Joisha *et al.* ont choisi cette représentation pour la puissance et les propriétés qu'offre l'arithmétique

$$\langle p_1, p_2, \ldots, p_n \rangle = \begin{pmatrix} p_1 & 0 & \cdots & 0 \\ 0 & p_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & p_n \end{pmatrix}$$

FIGURE 7: Représentation d'un tuple

sur les matrices de ce type. En effet toutes opérations entre deux matrices diagonales carrées, ou avec un scalaire retourne un résultat de la même forme.

Notion de formes illégales

Afin de compléter les axiomes du système, il est nécessaire d'ajouter une notion de formes illégales. Celle-ci sera utile, par exemple dans le cas d'une multiplication matricielle de deux variables dont les dimensions ne satisfont pas les contraintes liées à cette opération. Joisha *et al.* nous propose la notation $\langle \pi_1, \pi_2 \rangle$, avec $\pi_i \in \mathbb{Z}_-$ à laquelle il est possible d'ajouter des extensions unités pour former la classe d'équivalence des formes illégales.

Définition de prédicats

Afin d'établir des règles sur les opérations arithmétiques de variables, Joisha *et al.* ajoutent des prédicats. Ceux-ci permettent de définir trois fonctions, qui pour un tuple donné retourne un booléen déterminant si le tuple satisfait ou non une condition :

— θ : retourne vrai si la forme est valide ;
— β : retourne vrai si le tuple est une matrice ;
— α : retourne vrai si le tuple est un scalaire.

Ils ont choisis aussi d'ajouter une fonction utilitaire σ qui retourne 0 si son paramètre est différent de 0, et 1 si il est égal à 0. Les prédicats α et β peuvent être définies mathématiquement ainsi pour un shape-tuple $u = \langle r_1, r_2, ..., r_n \rangle$:

— $\beta(u) = \theta(u)\sigma(r_3 - 1)\sigma(r_4 - 1)\cdots\sigma(r_n - 1)$
— $\alpha(u) = \theta(u)\sigma(r_1 - 1)\sigma(r_2 - 1)\cdots\sigma(r_n - 1)$

Au contraire des deux autres, le prédicat θ est dépendant du contexte dans lequel la variable est assignée, c'est la sémantique des opérateurs qui permet d'établir une équation de validité. Comme Joisha *et al.*, prenons l'exemple du produit $c \leftarrow a * b$. Dans ce cas, on ne sait pas si a et b sont des scalaires et/ou matrices. Nommons s, t et u respectivement les tuples représentant a, b et c, ce qui donne l'équation suivante :

$$\theta(u) = \underbrace{\theta(s)\theta(t)}_{\substack{\text{on vérifie que } b \\ \text{et } c \text{ sont bien} \\ \text{valides}}} \underbrace{\Big[1 - \overbrace{\big(1-\alpha(s)\big)\big(1-\alpha(t)\big)}^{\substack{\text{si } b \text{ et } c \text{ sont des scalaires, alors il n'y a rien de plus à vérifier}}} \overbrace{\Big(1 - \underbrace{\beta(s)\beta(t)}_{\substack{\text{ce sont des} \\ \text{matrices}}} \times \underbrace{\sigma(s_2 - t_1)}_{\substack{\text{dimensions compatibles} \\ \text{pour la multiplication}}}\Big)}^{\substack{\text{sinon on vérifie que ce sont des matrices compatibles}}}}_{= 0 \text{ si opération valide}} \Big].$$

Grâce aux prédicats et cette fonction, il est possible de définir un système d'équations pour chaque opérateur arithmétique. Dans une étude plus approfondie, Joisha *et al* [3] présentent un tableau contenant une partie des opérateurs mathématiques de Matlab de type I et la formule de calcul de validité du résultat (θ).

Application du système algébrique sur les opérateurs arithmétiques

Avant d'énoncer la création de formules mathématiques pour le calcul de la forme et de la taille d'une

variable, Joisha *et al.* [10] montre qu'il est possible de déterminer le rang canonique maximum du tuple résultat de l'opération. Pour ce faire nous avons la propriété suivante : $R(u) = \max(v, w)$, où u est le tuple représentant le résultat, v et w ceux des opérandes. Il est donc possible d'étendre les *shape-tuples* des opérandes afin qu'ils soient de même dimension, et ainsi satisfaire toutes les contraintes des opérateurs utilisés dans le calcul du *shape-tuple*.

L'équation suivante, donné par Joisha *et al.* [10], correspond au calcul du *shape-tuple* résultat de l'opération $c \leftarrow a * b$ où u, s et t correspondent respectivement au résultat c et aux opérandes a et b.

$$u = \underbrace{\left(1-\theta\left(u\right)\right)\pi^{\star}}_{(1)} + \theta\left(u\right)\Bigg(\underbrace{s^{\star}\alpha\left(t\right) + t^{\star}\alpha\left(s\right)\left(1-\alpha\left(t\right)\right)}_{(2)} + \Big(\underbrace{s^{\star}\Gamma_1 + t^{\star}\Gamma_2}_{(3')} + \underbrace{\iota^{\star} - \Gamma_1 - \Gamma_2}_{(3'')}\Big)\left(1-\alpha\left(s\right)\right)\left(1-\alpha\left(t\right)\right)\Bigg)$$

où π, ι sont respectivement la forme illégale et l'identité, et Γ est une matrice dont le 1er et le second élément de la diagonale sont à 1 et le reste à 0. Le symbole $\star$ dénote n'importe quel élément du *shape-tuple*. Lors de l'utilisation de l'équation il sera donc remplacé par l'indice de l'élément courant.

Cette équation peut être découpée en trois parties. La première (1) concerne la détection d'une forme illégale, grâce au prédicat θ seul cette partie renverra un résultat dans le cas où la variable a est mal formée. La seconde partie (2) de l'équation permet de déduire la taille si a ou b sont des matrices. La dernière comporte une première partie ($3'$) permettant de s'assurer de la contrainte de taille dans un tuple, la seconde ($3''$) de calculer la taille des dimensions suivantes.

En suivant la même logique, Joisha *et al* [3] définissent les équations permettant le calcul du résultat des fonctions et opérateurs de Type-I. Pour conclure la définition du système algébrique, ils associent un opérateur à ces formules (i.e. $\circledast$ pour le calcul du *shape-tuple* de $\times$) , pour ensuite démontrer les propriétés qui les définissent tel que l'associativité, la commutativité, etc. Ce qui permet enfin de définir un système algébrique complet.

Application du système algébrique sur le CFG Il s'agit maintenant de mettre en action le système définit ci-avant. Joisha *et al.* [10] présentent la mise en application de cette technique sur l'exemple $c \leftarrow a * b$ avec $s = \langle p_1, p_2, 1\rangle$, $t = \langle q_1, q_2, q_3\rangle$ et u respectivement les tuples représentant a, b et c. Après remplacement dans les équations exprimées ci-avant, voici la matrice calculant le *shape-tuple* u pour la variable c :

$$u = \begin{pmatrix} X & 0 & 0 \\ 0 & Y & 0 \\ 0 & 0 & Z \end{pmatrix}$$

$$\text{où } X = \left(1-\theta\left(u\right)\right)\pi_1 + \theta\left(u\right)\Big(p_1\alpha\left(t\right) + q_1\alpha\left(s\right)\left(1-\alpha\left(t\right)\right) + \ p_1\left(1-\alpha\left(s\right)\right)\left(1-\alpha\left(t\right)\right)\Big)$$

$$Y = \left(1-\theta\left(u\right)\right)\pi_2 + \theta\left(u\right)\Big(p_2\alpha\left(t\right) + q_2\alpha\left(s\right)\left(1-\alpha\left(t\right)\right) + \ q_2\left(1-\alpha\left(s\right)\right)\left(1-\alpha\left(t\right)\right)\Big)$$

$$Z = \left(1-\theta\left(u\right)\right) + \theta\left(u\right)\Big(\alpha\left(t\right) + q_3\alpha\left(s\right)\left(1-\alpha\left(t\right)\right) + \ \left(1-\alpha\left(s\right)\right)\left(1-\alpha\left(t\right)\right)\Big)$$

Il est donc maintenant possible de déterminer pour chaque variable, assignée par une opération arithmétique, le *shape-tuple* correspondant. Cependant cette technique se limite aux opérations arithmétiques et aux fonctions de Type-I. Les fonctions des deux autres catégories nécessitent un autre type d'analyse qui n'entre pas dans le cadre de cette étude. Il est cependant nécessaire d'inférer un type aux variables assignées par ces fonctions.

3.3 Type indéterminé

Parfois l'analyse statique ne peut pas déterminer le type d'une variable. La cause peut être une ambiguïté sur le type dû au polymorphisme des opérateurs Scilab, ou un manque d'informations provenant par exemple de fonctions internes du langage dont les opérandes et résultat ne peuvent être connus. Dans ce cas, il est nécessaire de recourir à d'autres techniques qui consiste à demander au développeur des informations de type supplémentaires, ou par une génération de code différente par compilateur.

Utilisation de pragmas Un pragma est une directive ajoutée au programme qui ne modifie pas sa sémantique. Il s'agit d'une instruction pour le compilateur n'ayant pas d'incidence sur l'exécution du programme initial. Dans le cas de l'inférence de type, les pragmas permettent au développeur de préciser le type d'une variable au compilateur, levant ainsi toute ambiguïté, car le compilateur considère la parole du développeur comme toujours exacte (sauf si cela conduit à une contradiction).

Utilisation de variables cachées Dans le cas où la phase statique échoue à inférer un type intrinsèque, DeRose[2] propose l'ajout de variables cachées. Il s'agit tout d'abord d'éclater une expression arithmétique en une séquence d'opérations binaires, ce qui générera pour chaque résultat une variable temporaire du type correspondant à l'opération (assimilable à un code trois adresses). L'analyse d'une expression devient alors plus simple, et les mécanismes présentés précédemment réussissent à inférer le bon type pour chaque variable créée.

DeRose [2] utilise des variables cachées pour la détermination de taille lors de l'exécution du programme. Celles-ci sont initialisées à 0 en début de programme, puis mises à jour lors de l'exécution. Le principe est une gestion dynamique des informations de taille par génération de code C. Cette solution reprend ce qui existe dans le compilateur Scilab/C existant concernant la génération de code naïf comme indiqué dans la présentation de Scilab (cf exemple 1b).

Type générique Comme indiqué par DeRose [2], l'utilisation de matrices de complexes lorsque les informations de type sont inconnues est une approche satisfaisante. En effet leur article le montre avec la réalisation d'une série de tests, les cas où l'inférence statique de type échouent sont rares.

4 Analyse et optimisations

Une fois l'inférence de type, forme et taille achevée, on peut passer à une autre étape importante : *l'optimisation* du code produit. En effet traduire le code sous forme SSA sans modification, signifie un code avec de nombreuses variables. En outre, si Scilab a besoin d'une seule instruction pour assigner des tableaux, ce n'est pas le cas en C, les boucles qui en résultent doivent être regroupées pour un meilleur résultat.

4.1 Regroupement de l'utilisation des variables

La *kernelization* est évoquée dans l'article de Prasad [12]. Il s'agit de regrouper les instructions qui peuvent être exécutées par un GPU. Dans les sytèmes embarqués l'utilisation de système hétérogène est peu fréquente. Nous ne reprendrons donc de ce document que le concept de groupement d'instructions appelées *kernel*.

```
1  E = F + D
2  A = B + C
3  C = D + B
4  A = A + D
```

```
1  tempVar0 = B + C
2  A = tempVar0 + D
3  E = F + D
4  C = F + B
```

FIGURE 8: Exemple de regroupement

Dans notre cas, un *kernel* est le regroupement en terme de localité dans le programme des instructions utilisant les mêmes variables. L'exemple de la figure 8 montre le rassemblement des instructions définissant `A` d'une part, et utilisant `F` d'autre part. Le but est de pouvoir ensuite libérer l'espace mémoire des variables le plus tôt possible dans l'exécution du programme, de rassembler des boucles (cf. les sections suivantes 4.2, 4.3) et de limiter les *cache miss* (i.e. besoin de recharger des informations depuis la mémoire).

Joisha [11] propose l'utilisation d'un graphe d'interférence de variable. Puis il applique un algorithme de coloration de graphe afin d'identifier les instructions qui peuvent être jointes.

4.2 Minimisation des variables temporaires

La *kernelization* évoquée ci-avant, a donc permis l'identification des instructions qui peuvent être groupées. Cependant, et de par la définition de la forme SSA, un grand nombre de variables temporaires ont été injectées dans la représentation intermédiaire. Celles-ci correspondent à des copies qui vont ensuite être assignées à la variable initiale. Ces copies occupent de la mémoire, et lorsque celles-ci portent sur des tableaux/matrices de grandes tailles, la mémoire peut être saturée sur des dispositifs embarqués.

Pour réduire leur nombre, il est nécessaire dans un premier temps de déterminer les variables d'entrée et de sortie du kernel comme indiqué dans l'article de Prasad *et al* [12], puis de réaliser une analyse de propagation de copie comme évoquée par Joisha *et al*[11].

Comme indiqué par Prasad *et al*[12], la minimisation de l'empreinte mémoire des variables temporaires passe par 2 étapes. La première est la scalarization des temporaires, puis par l'élimination des index des tableaux. Ces deux étapes se traduisent par l'insertion de boucle pour tout ce qui est des opérations sur les tableaux. Dans la première phase, cela permet d'avoir des variables temporaires principalement de type scalaire. Dans la seconde, il s'agit du remplacement des opérateurs d'accès aux index d'un tableau qui sont spécifiques à Scilab tel que l'opérateur " : ", ou "`end`".

4.3 Regroupement de boucles

Une fois toutes les copies de tableaux supprimées, il ne reste que les variables importantes et des boucles sur les tableaux qu'il va être possible d'optimiser.

Il existe de nombreuses opérations que l'on peut faire sur les boucles pour optimiser l'exécution d'un programme. Une première approche est de regrouper au mieux certaines boucles afin de réduire le temps d'exécution, i.e deux boucles successives avec les mêmes bornes. Nous allons voir ce cas sur un exemple avec `A`, `B` et `C`, trois matrices carré de taille 100 (voir figure 9).

Le code généré de manière naïve crée deux matrices de même type que les matrices `A` et `B` afin de stocker les résultats des calculs intermédiaires tandis que lorsque l'on regroupe les boucles, il est possible de stocker ces valeurs dans de simples scalaires.

Dans un second temps, l'ordre dans les boucles peut être changé s'il y a plusieurs boucles imbriquées [12]. Dans ce cas le principe est de chercher à maximiser la localité des variables pour le CPU, ainsi si nous accédons à un tableau, nous allons chercher à accéder aux valeurs consécutives en mémoires. Par exemple, pour initialiser tous les $(M(i,j))_{i,j}$ à zéro, il est préférable d'itérer sur i, puis à i fixé d'itérer sur j car les cases dans les lignes sont consécutives en mémoire, contrairement aux colonnes.

Une autre amélioration portant sur les boucles est l'optimisation polyédrique. Elle consiste à garder le même espace d'itération mais elle modifie l'ordre de parcours (tout en conservant les dépendances), notamment pour améliorer la localité des instructions. Le but est de permettre d'appliquer le plus simplement possible cette optimisation (grâce à des librairies par exemple), ce qui implique d'en comprendre les contraintes. Les limites de cette technique sont les bornes des tableaux et la manière dont on accède aux éléments : il faut des accès affines, et s'il y a des indirections (comme `A[B[i]]`), alors seulement en lecture, pas en écriture. Il faut essayer de ne pas cacher les modifications par des fonctions intermédiaires, qui pourraient rendre l'application des outils automatiques inopérantes. L'optimisation polyédrique est en outre capable de détecter des boucles pouvant présenter des problèmes de caches (ce qui limite les performances) et le cas échéant de les séparer en deux boucles.

4.4 Préallocation des variables

Il est possible que la taille d'une matrice change au cours de l'exécution d'un programme Scilab. Dans le langage C, une instruction de réallocation doit être utilisée. Ces changements de tailles sont coûteux en temps d'exécution, et doivent donc être limités. Cela implique deux choses :

- déterminer le changement de tailles pour allouer dès l'initialisation de la variable la taille maximum nécessaire ;
- ne pas placer de réallocation dans le corps d'une boucle.

```
1  A = (B + C) - (A + C)
```

(a) Code Scilab

```
1  Tmp1 = B + C
2  Tmp2 = A + C
3  A1 = Tmp1 - Tmp2
```

(b) Représentation intermédiaire

```
1  int i,j;
2  //creation de deux matrices Tmp1 et Tmp2
3  for(i = 0; i<100; i++)
4     for(j = 0; j<100; j++)
5         Tmp1 = B[i][j] + C[i][j];
6
7  for(i = 0; i<100; i++)
8      for(j = 0; j<100; j++)
9          Tmp2 = A[i][j] + C[i][j];
10
11 for(i = 0; i<100; i++){
12     for(j = 0; j<100; j++){
13         Tmp1 = B[i][j] + C[i][j];
14         Tmp2 = A[i][j] + C[i][j];
15         A1[i][j] = Tmp1[i][j] - Tmp2[i][j];
16     }
17 }
```

(c) Code C naïf

```
1  int i,j;
2  //creation de 2 scalaires Tmp1 et
       Tmp2
3  for(i = 0; i<100; i++){
4      for(j = 0; j<100; j++){
5          Tmp1 = B[i][j] + C[i][j];
6          Tmp2 = A[i][j] + C[i][j];
7          A1[i][j] = Tmp1 - Tmp2;
8      }
9  }
```

(d) Code C après un regoupement de boucle

FIGURE 9: Exemple de regroupement de boucles

Comme il n'y a que des boucles `for` et qu'une modification de l'indice de boucle ne change pas le nombre d'exécution, il suffit de déterminer à l'entrée de la boucle, quel est la valeur de fin d'itération. Puis de calculer la taille maximale de la variable après la boucle, pour finalement initialiser la matrice avec la bonne taille en début de fonction. Si le compilateur échoue à déterminer la taille en sortie de boucle, il est intéressant de déterminer un maximum du nombre d'itérations, et de calculer un maximum pour la taille à ré-allouer.

Après la présentation de différentes optimisations, il s'agit maintenant de réaliser leur implémentation afin de déterminer leurs réelles efficacités.

5 Mise en oeuvre

La mise en oeuvre des solutions proposées dans les chapitres précédents consistera à poursuivre les travaux sur le projet Gecos. Afin d'introduire l'implémentation de notre étude, nous vous présenterons ce projet.

5.1 Introduction à Gecos

Le projet GeCoS *Generic Compiler Suite*[3] est développé par l'équipe Cairn (INRIA/ÉNS Cachan) depuis 2004. Il peut être utilisé comme un compilateur C (avec un support partiel du C++) ou comme compilateur sur une représentation intermédiaire autonome grâce à un *back-end* entièrement reconfigurable. Il est écrit en Java, et s'appuie sur *Eclipse Modelling Framework* (EMF) en tirant parti du mécanisme de plug-in, et est donc développé sous la forme d'un plug-in Eclipse[4].

3. http://gecos.gforge.inria.fr/doku.php
4. http://www.eclipse.org/

Le projet s'interface avec des outils externes, notamment Tom/Gom [5] qui facilite le détection de motifs (*pattern-matching*) sur des arbres. Ce dernier se couple avec Java et est composé d'un compilateur de motif pour la détection de sous-arbre, ainsi que d'un moteur de ré-écriture de termes. Ce projet est développé à INRIA par l'équipe Pareo, qui souhaite fournir un moyen élégant d'exprimer des *pattern-matching* sur des arbres de syntaxes abstraits et autres représentations intermédiaires sous forme d'arbres.

5.2 Présentation de l'existant

Gecos est déjà fonctionnel dans le sens où les *front-end* et *back-end* du compilateur ont déjà été réalisés et connectés pour travailler sur une même représentation intermédiaire. Il est possible de convertir le CFG sous la forme SSA et vice versa, il est aussi possible de générer les DAG des instructions.

La face avant permet de parser du code Scilab et la partie arrière de générer les structures de contrôle en C du programme initiale, l'inférence de type étant manquante. Les évolutions, que nous apporterons, porteront donc sur l'ajout d'information de type dans la représentation intermédiaire, et permettre ainsi au *back-end* de générer un code C complet.

Afin de réaliser cette opération, nous allons devoir utiliser des algorithmes de parcours d'arbres. Pour réaliser cela, le projet Gecos comporte déjà des mécanismes que nous pourrons utiliser, tel que l'utilisation du patron de conception *Visiteur*. Actuellement, l'analyseur syntaxique réalise déjà un parcours d'arbre afin de propager les informations de type découvert lors de l'initialisation des variables par des constantes.

Dans la section 3.3, nous avons évoqué l'utilisation de pragmas. Il est à noter que le projet Gecos implémente une série de pragmas, notamment pour dérouler les boucles ou les fonctions *for-inlining, function inlining.*

5.3 Inférence de type

Nous avons prévu dans la partie d'inférence de type de procéder par étape, d'abord l'inférence du type intrinsèque, puis la forme et la taille des variables.

Il nous faudra donc commencer par utiliser les mécanismes de parcours d'arbre afin de réaliser l'analyse *data-flow* sur le CFG sous forme SSA du programme source, décrite dans la section 3.1. Grâce au treillis de cette même section nous pourrons déterminer le type intrinsèque de toutes les variables. Cette passe du compilateur sera exécutée un nombre arbitraire de fois dépendant de la complexité du programme source, jusqu'à l'obtention d'un point fixe.

Ensuite nous créerons une nouvelle passe qui sera chargée de calculer la forme et la taille des variables en appliquant les équations du système algébrique défini par la section 3.2. Après concertation avec notre tuteur, nous procéderons à une étape préalable. Dans la section évoquée ci-avant, nous avons présenté les limites de l'approche basée sur les treillis concernant l'inférence de forme : nous allons dans un premier temps utiliser cette technique ; en effet la mise en place du système algébrique étant plus complexe, nous ne l'implémenterons que si le temps nous le permet.

En plus des mécanismes d'inférence statique que nous mettrons en place, nous poursuivrons l'implémentation des pragmas dans Gecos afin de permettre au développeur de code Scilab de préciser des types au compilateur. Ces précisions supplanteront toutes inférences statiques sur la variable concernée.

Dans le cas où notre système serait dans l'incapacité d'inférer statiquement le type d'une variable, nous afficherons un message (*warning*) à l'utilisateur lui indiquant de préciser un pragma, ou de compléter son code. Cependant, afin de pouvoir générer un code C complet, nous utiliserons le type générique comme décrit dans la section 3.3.

5.4 Génération de boucles/de code

Comme nous en avons déjà discuté, l'une des principales difficultés de la traduction du code Scilab vers du C est la manipulation des matrices en Scilab, en particulier les « *magical ends* » et l'assignation de tableau.

5. `http://tom.loria.fr/`

Le premier problème est l'utilisation des « magic end » : ils vont être remplacés des `size(A)`. Il faut donc remplacer les `M(a:)` par des `M(a:length(M))`.

```
1 int step1 = a>b ? 1 : -1;
2 int step2 = d<e ? 1 : -1;
3 {
4   int j=d;
5   for(int i=a; i != b; i += step1,
6     j+= step2)
7      A[i] = C[j];
8 }
```

L'autre difficulté est l'assignation de tableaux qui n'existe pas en C. Prenons un exemple courant en Scialb : `A[a:b] = C[d:e]`, cela correspond à assigner les éléments de `A` par des éléments de `C`. L'idée pour faire la traduction est l'ajout d'une boucle. Il faut faire attention à plusieurs choses. Tout d'abord, nous souhaitons supporter une particularité Matlab, qui n'est pas supportée par Scilab : si $\texttt{a} > \texttt{b}$, il faut alors parcourir le tableau dans l'autre sens. Comme nous supposons le programme correct, nous supposerons que les intervalles ont bien la même longueur. Comme le montre la traduction à gauche, la première étape consiste à déterminer le sens dans lequel on parcourt le tableau (on incrémente ou on décrémente), puis la copie de `C` vers `A` est faite.

Cependant dans ce cas simple, pour que cet exemple fonctionne, des hypothèses implicites ont été faites. Tout d'abord, il faut que la taille soit suffisamment grande : pour cela, il faut, si nécessaire, agrandir la matrice (avec le problème de la position des tests et des réallocations déjà décrit). L'autre difficulté est la dépendance des données si $C = A$; par exemple, la boucle précédente, ne va pas donner le résultat attendu pour `A(2:4))=A(1:3)` : lors de la lecture `A(2)`, c'est la nouvelle valeur qui va être lue (la même que dans `A(1)`, car cette copie a déjà été faite). Il faudra donc, soit modifier l'ordre d'écriture, soit faire une copie de tableau.

5.5 Réalisation des optimisations

La réalisation des optimisations présentées dans le document se fera sur la représentation intermédiaire donnée par Gecos, sur laquelle nous avons préalablement fait une analyse du type des expressions. Selon l'optimisation, soit une simple recherche de motif d'arbre va suffire (notamment pour les permutations sur l'ordre des boucles), alors que pour les réordonnancements d'instructions et les fusions de boucles, une étude plus précise sur les dépendances entre les variables est nécessaire. Il est difficile de prévoir à quel point l'ordre dans lequel les transformations seront effectuées aura une importance, mais il faudra probablement faire plusieurs cycles d'optimisation : certaines optimisations pourraient permettre d'appliquer d'autres optimisation qui n'avaient pas été reconnues comme applicables auparavant.

5.6 Application sur un programme de stéréovision

Dans le cadre du projet, les tests seront réalisés sur une plate-forme embarquée connectée à un dispositif de *stéréo-vision*. L'architecture matérielle est composée de :

- un BeagleBoard-xM [6] qui avec 512MB de RAM et un processeur mono-cœur cadencé à 1GHz (Cortex A8) ;
- deux caméras.

Il s'agira donc d'utiliser un algorithme écrit en Scilab dont le principe est de recréer une image à partir des deux images provenant des caméras. Un exemple d'algorithme réalisant cette fonctionnalité est disponible pour Matlab [7].

Cette application a été choisie car elle illustre bien les problèmes évoqués précédemment. En effet, l'implémentation du programme de *stéréo-vision* nécessite de réaliser des calculs sur des matrices correspondant aux images, avec une contrainte de temps réel. Cette contrainte d'exécution nécessite d'être le plus efficace possible afin que l'enchaînement des images résultantes ne soit pas saccadé. Nous allons donc pouvoir tester et mesurer la qualité des solutions proposées antérieurement après compilation du programme Scilab en C

6. `http://beagleboard.org/Products/BeagleBoard-xM`
7. `http://www.mathworks.fr/products/computer-vision/examples.html?file=/products/demos/shipping/vision/videostereo.html`

puis en code machine via un compilateur externe (comme GCC ou Clang) et enfin par le lancement de l'exécutable sur la plate-forme de *stéréo-vision*.

6 Conclusion

La compilation du langage Scilab révèle de nombreux problèmes dont le principal est l'inférence de type. La problématique générale de la compilation est la génération de code efficace. Cela est d'autant plus vrai dans le domaine des systèmes embarqués car rappelons-le, ce sont des architectures possédant des ressources limitées aussi bien en termes de rapidité de calcul que de capacité mémoire. Cependant la génération de code efficace depuis un langage source utilisant un typage dynamique tel que Scilab est pleinement dépendante de la capacité du compilateur à correctement inférer le type des variables. Comme démontré dans la section 3, ce document vise à répondre au mieux à ces problématiques d'inférence de type et de génération de code efficace.

Mais l'inférence de type étant NP-dur[9], les solutions présentées sont donc correctes mais pas optimales. En effet un code C produit par un développeur sera toujours plus efficace que ce que peut générer un compilateur Scilab vers C. Une amélioration possible serait d'étendre l'inférence de forme afin de distinguer, par exemple, les matrices triangles, diagonales, etc des matrices quelconques. Cette évolution aurait pour effet d'augmenter les performances de l'application, car le compilateur saurait alors générer des algorithmes adaptés à la forme des matrices. En effet une multiplication de deux matrices diagonales est linéaire $O(n)$, alors que l'algorithme général est cubique $O(n^3)$.

Phase indispensable d'une conception de projet de recherche, la théorisation des limites et contournements de ces limites permettent de gagner un temps précieux lors de la réalisation. Ainsi, les solutions proposées au niveau conceptuel pourront et seront testées lors de leur implémentation.

Références

1. Scilab Enterprises, *Scilab : Le logiciel open source gratuit de calcul numérique*. Scilab Enterprises, Orsay, France, 2012.
2. L. De Rose and D. Padua, "Techniques for the translation of matlab programs into fortran 90," *ACM Trans. Program. Lang. Syst.*, vol. 21, pp. 286–323, Mar. 1999.
3. P. G. Joisha and P. Banerjee, "An algebraic array shape inference system for matlab®," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 5, pp. 848–907, 2006.
4. F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, pp. 1–19, July 1970.
5. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, vol. 13, pp. 451–490, 1991.
6. G. Almási and D. Padua, "Majic : Compiling matlab for speed and responsiveness," *SIGPLAN Not.*, vol. 37, pp. 294–303, May 2002.
7. L. A. D. Rose, O. D. Rose, M. Stat, L. Antonio, D. Rose, and P. D, "Compiler techniques for matlab programs," 1996.
8. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers : Principles, Techniques and Tools*. Addison-Wesley, 1988.
9. M. Wand and P. O'Keefe, "On the complexity of type inference with coercion," in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, (New York, NY, USA), pp. 293–298, ACM, 1989.
10. P. Joisha, U. Nagaraj Shenoy, and P. Banerjee, "Computing array shapes in matlab," in *Languages and Compilers for Parallel Computing* (H. Dietz, ed.), vol. 2624 of *Lecture Notes in Computer Science*, pp. 395–410, Springer Berlin Heidelberg, 2003.
11. P. G. Joisha and P. Banerjee, "Static array storage optimization in matlab.," in *PLDI* (R. Cytron and R. Gupta, eds.), pp. 258–268, ACM, 2003.
12. A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of matlab programs for synergistic execution on heterogeneous processors," *SIGPLAN Not.*, vol. 46, pp. 152–163, June 2011.