Le compilateur : un outil mystérieux du calcul scientifique

Les calculateurs utilisés dans les grands problèmes du calcul scientifique ont atteint des degrés élevés de parallélisme. D'autre part les applications deviennent très sophistiquées, que ce soit en raison de leur algorithmique interne ou en raison de la nécessité d'exposer un parallélisme de plus en plus massif.

Les langages de programmation offrent une aide importante à la mise en oeuvre de programmes. Nous essaierons dans cet exposé de donner une idée des concepts et des technologies sous-jacentes, en particulier concernant la phase de "compilation" destinée à obtenir du code machine.

Notre propos est surtout d'aider à la construction de grandes applications en facilitant la compréhension d'ensemble du processus de construction applicative.

Points couverts

- Notions sur les langages de programmation
- Notions de sémantique
- Le processus de traduction
- Les représentations intermédiaires
- Traduction vers une représentation de haut niveau
- Transformations à haut niveau
- Traduction vers une représentation de bas niveau
- Génération de code
- Optimisation des ressources
- Ordonnancement

Première partie

Langages de programmation et sémantique

- Principes des langages impératifs
- Syntaxe des langages
- Types et données
- La sémantique: donner un sens au programme

Langages impératifs

Un langage de programmation impératif possède les notions:

variable: élément mémorisant. Retient la valeur assignée et permet de la récupérer jusqu'à la prochaine assignation.

instruction: peut utiliser l'ensemble des valeurs des variables accessibles et modifier certaines par assignation de nouvelles valeurs.

En pratique on distingue:

1. les déclarations: définissent des variables
2. les instructions: opérent sur les variables

Syntaxe

Le langage de programmation définit une syntaxe précise pour les instructions et les déclarations. Cette syntaxe est décrite à partir de règles grammaticales et lexicales.

Le compilateur doit d'abord analyser la syntaxe. Le processus est en général à deux niveaux:

- analyse lexicale. Partitionne le texte en jetons lexicaux. Peut souvent se formaliser par une grammaire d'expressions régulières, décodée par un automate fini. *La plupart des compilateurs ont un parseur lexical réalisé avec du code ad-hoc.*
- analyse syntaxique. Reconnait la structure syntaxique du code, a l'aide d'un automate à pile défini à partir de la description grammaticale du langage.

Exemples d'extrait de grammaire

```
%start debut
%% /* Grammar rules and actions follow */
debut   : liste_expr;

liste_expr : expression
           | liste_expr  '=' expression ;

expression :   expr_arith

expr_arith :   expr_atome
             | expr_arith '+' expr_arith
             | expr_arith '-' expr_arith
             | expr_arith '*' expr_arith
             | expr_arith '/' expr_arith
             | '-' expr_arith  %prec NEG
             | expr_arith '^' expr_arith
             | '(' expr_arith ')'

expr_atome : ENTIER
           | FLOTTANT
;
```

Avec une sémantique d'interprétation directe (calculette)

```
debut   : liste_expr ;

liste_expr :  expression                 { return_value("%le\n",$1); }
            | liste_expr  '=' expression  { return_value("%le\n",$3); }
;

expression :   expr_arith                { $$ = $1; }

expr_arith :   expr_atome                { $$ = $1;         }
             | expr_arith '+' expr_arith  { $$ = $1 + $3;    }
             | expr_arith '-' expr_arith  { $$ = $1 - $3;    }
             | expr_arith '*' expr_arith  { $$ = $1 * $3;    }
             | expr_arith '/' expr_arith  { $$ = $1 / $3;    }
             | '-' expr_arith  %prec NEG  { $$ = -$2;        }
             | expr_arith '^' expr_arith  { $$ = pow ($1, $3); }
             | '(' expr_arith ')'         { $$ = $2;         }

expr_atome : ENTIER                      { $$ = $1; }
           | FLOTTANT                    { $$ = $1; }
;
```

Ou en est t'on après l'analyse syntaxique ?

Avec des types de données simples, il serait évidemment possible d'essayer d'émettre du code immédiatement après l'analyse syntaxique. Ceci consiste à enregistrer le programme de la calculette de l'exemple ci-dessus au lieu de l'interpréter immédiatement.

L'inconvénient est que l'on a très peu d'information pour optimiser et même pour comprendre le sens[a] du code. Le compilateur pcc qui était distribué avec Unix dans les années 1980-90 faisait cela pour le code des tests, des boucles et utilisait une technique d'optimisation pour les expressions.

Pour pouvoir optimiser, on enregistre toute l'information sous la forme d'un arbre de syntaxe abstraite. En gros, il s'agit d'un arbre avec un noeud par règle dans la grammaire. *On ne fait alors qu'enregistrer la décomposition syntaxique du programme...*

[a] types!!

Types

Un type définit:

1. l'ensemble des valeurs que peut prendre une variable ou une expression,
2. les opérations qui peuvent être appliquées à ces valeurs ou variables.

En pratique, les types interviennent à deux étapes:

1. dans les déclarations.
   - associer des types aux variables, des signatures aux procédures,
   - construire des types plus riches que les types de base du langage: les types définis par l'utilisateur à l'aide des expressions de type.
2. dans l'analyse des expressions.
   - vérification du typage,
   - règles d'évaluation,
   - règles de conversion.

Typage statique ou dynamique ?

typage dynamique: la vérification des types est effectuée à l'exécution. Exemple: Matlab.

typage statique: la vérification des types est effectuée à la compilation. Un langage est fortement typé, si par construction du langage, le compilateur peut se garantir contre toutes les erreurs de typage.

En pratique, le typage statique permet un grand nombre d'optimisations:

- sélection des opérateurs : génération de code machine,
- sélection des fonctions à appliquer : inlining des appels,
- simplifications diverses d'expressions intermédiaires.

Problème: héritage dans les langages orientés objet.

Sémantique: donner un sens au programme

Plusieurs méthodes existent:

- sémantique opérationnelle: on donne un sens au programme en spécifiant un procédé d'interprétation de référence. Par exemple, on donne des règles d'interprétation des expressions et de la structure de contrôle. Important en compilation, la cible est un **langage machine** qui définit une *sémantique opérationnelle*.
- sémantique par assertions: Il s'agit de formaliser la relation entre les conditions à l'entrée (precondition) et les résultats produits (postcondition). Notion utilisée en génie logiciel.
- sémantique dénotationnelle: il s'agit de construire une fonction représentant le programme. On s'affranchit ainsi de la description des états intermédiaires. En pratique, il s'agit de procédés procédant récursivement à partir de la syntaxe.

Du programme à sa sémantique opérationnelle.

Dans le cadre de la compilation, on définit le plus souvent la sémantique opérationnelle par une machine virtuelle.

La compilation se conçoit alors comme une cascade de traductions, la dernière machine virtuelle correspondant à la cible concrète.

L'obtention d'une description de la sémantique opérationnelle à partir de la syntaxe consiste à

- préciser les types. Ceci requiert des tables de symboles, et conduit à des algorithmes pour typer les résultats d'expressions.
- décrire une mise en oeuvre à bas niveau des instructions de contrôle,
- décrire la mise en oeuvre de mécanismes permettant de gérer les variables.

Seconde partie

Le processus de traduction et les représentations intermédiaires

- Le processus de traduction
- Les représentations intermédiaires
- Traduction vers une représentation de haut niveau
- Traduction vers une représentation de bas niveau

Le processus de traduction

Le processus de traduction consiste à transformer une représentation du programme en une autre.

Ce processus n'est pas continu il est du type:

traduction ➔ (transformation ou optimisation)* ➔ traduction

Durant tout ce processus, on respecte la sémantique. C'est à dire que le sens ne change pas.

Les transformations opèrent souvent sur une représentation interne du programme en forme de code pour la n-ième machine virtuelle.

Pourquoi des cibles intermédiaires ?

Tout ne peut être bien représenté en langage machine

Les raisons de l'introduction de machines virtuelles intermédiaires sont multiples:

- faire en sorte que la majeure partie du compilateur soit portable,
- utiliser une description idéalisée, qui n'ait pas à s'encombrer de contraintes liées à la nature des ressources sur la machine cible. Il faut reculer le moment de prendre en compte ces contraintes pour ne le faire qu'au vu de l'information qui permettra d'optimiser. *Exemple*: **allocation de registres**,
- ne pas s'encombrer de contraintes sur la forme des instructions, par exemple adressages permis pour les opérandes,

Pour permettre l'optimisation, on cherche à représenter dans le code intermédiaire tout le potentiel de parallélisation.

Cet objectif en lui seul est contradictoire avec l'utilisation séquencée de ressources en petit nombre.

Les représentations intermédiaires

On se bornera à décrire deux représentations intermédiaires:

1. la représentation RTL de GCC. Elle sera dite de bas niveau.
2. une représentation dite SSA due a J. Ferrante et R. Cytron. Elle sera dite de haut niveau.

Une représentation de bas niveau

1. Programme représenté par une liste doublement chaînée d'insn (instructions)
2. Formats d'instructions:
   - insn
   - jump_insn
   - call_insn
   - code_label

Les instructions indiquent l'effet produit sous la forme d'une expression. Y sont détaillés:

- les ressources mémoire ou registre, une expression d'adressage si besoin,
- les types de base,
- les codes conditions utilisés ou générés
- les informations rendues invalides

Une représentation de bas niveau

```
int main(int argc, char * argv[]) {
   return argc;
}
```

```
;; Function int main(int, char**)
(note 3 8 4 NOTE_INSN_FUNCTION_BEG)
(insn 14 13 15 (nil) (set (reg:SI 62)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 argc+0 S4 A32])) -1 (nil)
    (nil))
(insn 15 14 16 (nil) (set (reg:SI 58)
        (reg:SI 62)) -1 (nil)
    (nil))
(jump_insn 16 15 17 (nil) (set (pc)
        (label_ref 27)) -1 (nil)
    (nil))
(barrier 17 16 18)
(insn 22 21 23 (nil) (set (reg:SI 58)
        (const_int 0 [0x0])) -1 (nil)
    (nil))
(jump_insn 23 22 24 (nil) (set (pc)
        (label_ref 27)) -1 (nil)
    (nil))
(note 25 24 29 NOTE_INSN_FUNCTION_END)
```

Une représentation de bas niveau

```
int main(int argc, char * argv[]) {
   return argc ? 0 : 2;
}
```

```
;; Function int main(int, char**)
(note 3 8 4 NOTE_INSN_FUNCTION_BEG)
(insn 14 13 15 (nil) (set (reg:CCZ 17 flags)
        (compare:CCZ (mem/f:SI (reg/f:SI 53 virtual-in-args)
            (const_int 0 [0x0]))) -1 (nil)
    (nil))
(jump_insn 15 14 16 (nil) (set (pc)
        (if_then_else (ne (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 19)
            (pc))) -1 (nil)
    (nil))
(insn 16 15 17 (nil) (set (reg:SI 61)
        (const_int 2 [0x2])) -1 (nil)
    (nil))
(jump_insn 17 16 18 (nil) (set (pc)
        (label_ref 21)) -1 (nil)
    (nil))
(barrier 18 17 19)
(code_label 19 18 20 2 "" [0 uses])
(insn 20 19 21 (nil) (set (reg:SI 61)
        (const_int 0 [0x0])) -1 (nil)
    (nil))
(code_label 21 20 22 3 "" [0 uses])
(insn 22 21 23 (nil) (set (reg:SI 58)
        (reg:SI 61)) -1 (nil)
    (nil))
(jump_insn 23 22 24 (nil) (set (pc)
        (label_ref 34)) -1 (nil)
    (nil))
(insn 29 28 30 (nil) (set (reg:SI 58)
        (const_int 0 [0x0])) -1 (nil)
    (nil))
(jump_insn 30 29 31 (nil) (set (pc)
        (label_ref 34)) -1 (nil)
    (nil))
(note 32 31 36 NOTE_INSN_FUNCTION_END)
```

Avantages et inconvénients de la représentation de bas niveau

Avantages:

- Structure linéaire proche du code machine,
- Sémantique opérationnelle simple à interpréter (ressemble à du code machine)
- Beaucoup de choix déjà fait sur la représentation mémoire.

Inconvénients:

- Sémantique des structures de contrôle (boucles, nids de boucles, ...) peu apparente,
- Sémantique “dénotationnelle” cachée: Que fait le programme ? Quelle expression calcule t’on ?
- Représentation en mémoire pas nécessairement optimisée ? Avec quelles informations ? Pour quel critère ?

Troisième partie

Spaghetti ?

Une partie importante de la sémantique est cachée dans la relation:

Quelle définition de valeur correspond à chaque utilisation ?

On exprime ces relations par un graphe USE - DEF entre les utilisations et les définitions. La précision ou la pertinence de la représentation est réduite par les alias.

Situations d'alias:

1. utilisation de la même variable pour représenter des valeurs distinctes,
2. on ne dispose pas des éléments pour décider du fait que deux variables représentent bien des valeurs distinctes, ou du fait que deux variables sont distinctes .

Analyse du flot de données

On désigne par flot de données la relation entre:

- les définitions (DEF)
- les utilisations (USE)

Ceci conduit directement à une première série d'optimisations:

- Elimination des expressions redondantes,
- Elimination des expressions constantes,
- Elimination du code mort.

Attention: Optimisation et mauvaise programmation sont des questions orthogonales. L'optimisation ne peut rien faire pour un code mal conçu. Dans un code bien conçu, l'optimisation est nécessaire pour traduire des abstractions de haut niveau de manière effi cace. L'intérêt pour le programmeur est de pouvoir se concentrer sur l'essentiel à haut niveau.

Analyse globale du flot de données

Il s'agit de recueillir et de traiter ce type d'information USE - DEF de la manière la plus globale possible.

> Etant donné le flot de contrôle du programme, quel est le flot de données ?
>
> Quelle définition de donnée affecte quelle utilisation ?

Dans de nombreux cas, il est possible de définir les informations à calculer par des jeux d'équations sur des ensembles. Il faut résoudre ces équations, lorsque c'est possible.

Une typologie des méthodes, qui sont souvent combinées:

1. méthodes itératives,
2. méthodes basées sur la théorie des graphes,
3. méthodes approchées.

Pour les détails, voir l'article "A survey of Data Flow Analysis Techniques", par K. Kennedy, dans [PFA].

Une représentation de haut niveau

Nous allons décrire la construction d'une représentation SSA [SSA] qui veut dire Static Single Assignment (Assignation Unique Statique).

Les objectifs[a] assignés à cette représentation:

- améliorer l'analyse globale de flots de données,
- améliorer l'efficacité algorithmique de nombreuses optimisations,
- offrir des opportunités supplémentaires d'optimisation,
- être extensible aux questions de parallélisme associée aux tableaux [ASSA].

[a] à moins qu'il ne s'agisse des vertus reconnues à SSA depuis son invention ?

Les bases pour la représentation SSA

Cette représentation se construit à partir des notions:

- Le Graphe de Flot de Contrôle CFG. Regroupe les séquences d'instructions sans branchement (Blocs de Base).
  1. Les Blocs de Base BB sont les sommets du CFG ([a]),
  2. Les arcs décrivent les branchements dans le code([b]),
- Chaque instruction évalue une ou plusieurs expressions, et alternativement:
  1. assigne des valeurs à une ou plusieurs variables distinctes,
  2. utilise le résultat pour sélectionner la prochaine instruction (Branchement).

[a] On convient d'ajouter un noeud entrée et un noeud sortie

[b] On ajoute un arc allant de entrée à sortie, pour des raisons techniques

Le caractère distinctif de la représentation SSA

Assignation unique: On renomme les variables de telle sorte qu'une variable soit la cible d'une seule assignation. Evite les alias.

Fonctions Φ: Il s'agit de fonctions de combinaison. On utilise ces fonctions pour faire en sorte qu'une seule définition aboutisse à chaque utilisation.

Les intérêts de la représentation SSA sont:

- une très grande simplification des graphes USE-DEF,
- la possibilité de regrouper et de résumer l'information sémantique([a]).
- algorithmes optimisés pour de nombreuses questions.

[a] par exemple résultant des appels de fonction

Représentation SSA

Nous allons donner une suite d'exemples, en commençant par les cas les plus simples de blocs de base.

Intuitivement, *l'intérêt de la représentation SSA est qu'il s'agit d'une représentation compacte des relations de dépendances décrivant l'usage des variables.*

En outre, il est possible de résumer et d'intégrer l'effet des appels de fonctions: on a évite de placer une frontière trop imperméable entre les optimisations locale et globale.

Conversion vers SSA: bloc de base

```
A = 5;
... = A + 1;
A = 6;
... = A + 2;
```

A_1 = 5;
... = A_1 + 1;
A_2 = 6;
... = A_2 + 2;

Dans ce cas, on a identifié chaque définition par une variable distincte. Clairement, dans les algorithmes qui seront utilisés par la suite, la relation DEF-USE est représentée directement dans la désignation des variables.

Conversion vers SSA: structure de contrôle

```
if (A > B) {
C= ...;
} else {
C= ...;
}
... = C;
... = C;
```

```
if (A > B) {
C1= ...
} else {
C2= ...
}
C3= Φ(C1, C2);
... = C3;
... = C3;
```

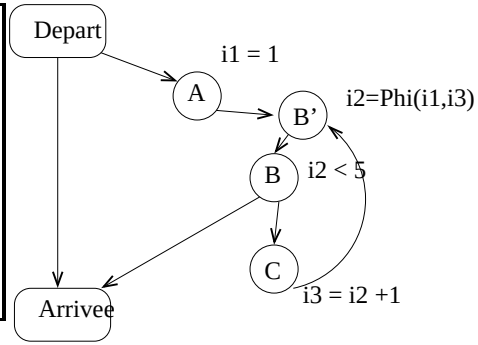
On commence par renommer en fonction des défi nitions. A la sortie de la conditionnelle, la fonction Φ formalise[a] le choix de la bonne défi nition. L'introduction de C_3 simplifi e la suite de l'analyse.

[a] i.e. ne se retrouve pas nécessairement dans le code produit.

Conversion vers SSA: boucle

```
A    int i = 1;
B    while (i < 5) {
C      i = i + 1;
     }
```

```
A    int i1 = 1;
B    while (i2 = Φ(i1,i3), i2 < 5) {
C      i3 = i2 + 1;
     }
```



Le principe est le même, il faut tenir compte des données provenant des itérations antérieures. Ici, B peut être atteint par deux chemins, l'un provenant du début du code, l'autre de l'itération de la boucle. On place donc une définition et une fonction Φ pour les regrouper.

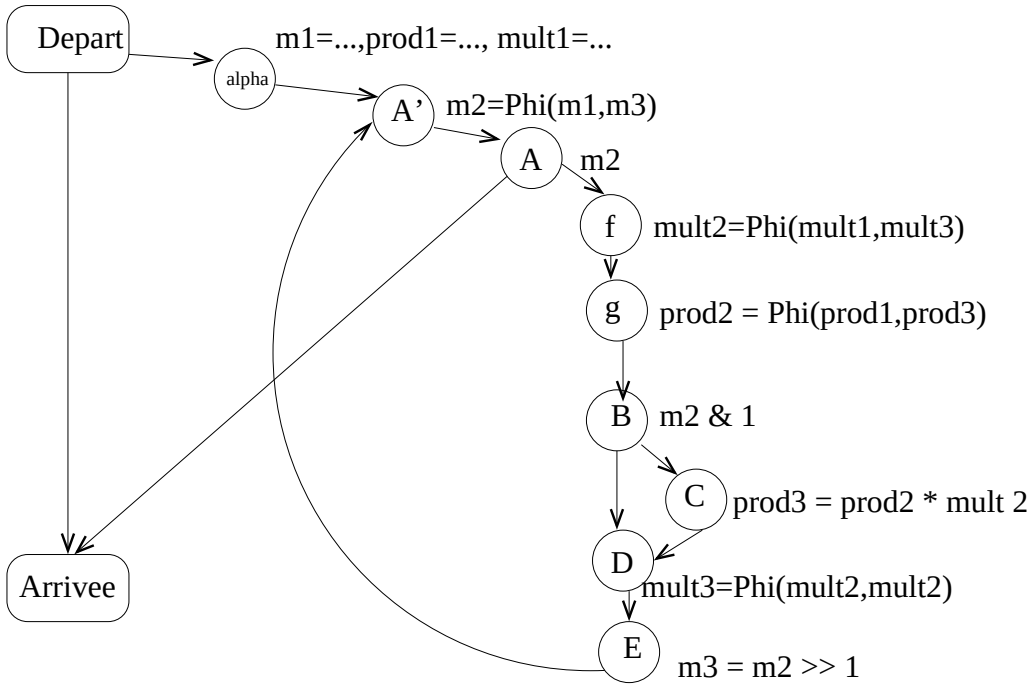
Conversion vers SSA: combinaison

```
     float x;
     int m;
     float prod = 1;
     float mult = x;
A    while ( m ) {
B      if ( m & 1){
C        prod *= mult;
       }
D    mult *= mult;
E    m>>=1 ;
     }
```

```
α    int m1;
     float prod1 = 1;
     float mult1 = x;
A    while ( m2 = Φ(m1,m3) ) {
f      mult2 =Φ(mult1,mult3);
g      prod2 =Φ(prod1,prod3);
B      if ( m2 & 1){
C        prod3 = prod2 * mult2;
       }
D    mult3 = mult2 * mult2;
E    m3 = m2 >>1 ;
     }
```

Conversion vers SSA: combinaison

```
        int m1 ;
        float prod 1 = 1;
        float mult 1 = x;
A       while ( m2 = Φ(m1,m3) ) {
f         mult2 =Φ(mult1,mult3);
g         prod2 =Φ(prod1,prod3);
B         if ( m2 & 1){
C          prod3 = prod2 * mult2;
          }
D       mult3 = mult2 * mult2;
E       m3 = m2 >>1 ;
        }
```



SSA et algorithmique

Le procédé de traduction vers la forme SSA a été formalisé sous une forme efficace du point de vue algorithmique. ([SSA]). En particulier, la question du placement des fonctions Φ a été ramenée à des problèmes d'algorithmique sur les graphes. Il en ressort que la borne de complexité dans le pire des cas est $\mathbf{O}(R^3)$ mais que le comportement mesuré sur des programmes réels est $\mathbf{O}(R)$, où R représente la taille du problème.

On dispose des résultats suivants concernant les algorithmes d'analyse et d'optimisation:

- Elimination des redondances partielles, (expressions calculées plusieurs fois dans au moins un chemin d'exécution): coût linéaire [PRE] en la taille du graphe de flot de contrôle.

L'extension de l'approche SSA aux cas:

- Pointeurs: voir [Ptr]
- Parallélisme: voir [ASSA]

Utilisation de SSA pour l'analyse du parallélisme

1. Initialement SSA a été mis au point pour les variables scalaires, avec une application simplifiée au tableaux. (fonctions Access(A,i) et Update(A,i,x) dans [SSA]. Conforme à l'état de l'art vers '90, avec la séparation de la compilation **scalaire** et des aspects de **parallélisation**.

2. L'extension aux tableaux est le point de départ pour l'utilisation de cette approche en parallélisation. Nous la décrivons conformément a [ASSA].

   - Clarification du rôle des fonctions Φ: celles-ci servent à résumer l'ensemble des *choix possibles pour le sens d'une variable*. Elles sont placées, en essayant de minimiser leur nombre, aux positions stratégiques.
   - Pour les tableaux, l'analyse DEF-USE pose problème en raison de la possibilité de mise à jour partielle d'un tableau. On conserve la valeur pour une partie des indices, lors de certaines assignations.

Extension de SSA aux tableaux selon [ASSA]

Les principes de cette extension sont les suivants:

1. toute assignation d'une variable indexée A(i)[1] est une mise à jour partielle de la variable A. Une fonction Φ de **définition** décrit cette assignation partielle,

2. on insère des fonctions Φ aux frontières de domination dans le graphe de contrôle comme dans le cas scalaire, pour tenir compte des variables dont la définition peut provenir de plusieurs assignations conformément au graphe de contrôle.

Concernant les assignations partielles, on utilise l'information des tableaux @:

- $@X_k[j]$ représente l'instant dans l'espace d'itérations où l'élément de X_k d'indice j a été modifié la dernière fois.

- les expressions en $@X_k[j]$ sont utilisés pour définir les fonctions Φ.

[1] ou B(i,j,...) car on traite également les cas multiindexés

Exemple très facile d'application à la parallélisation

```
1    if (c){
2       for (int i=0; i < lg ; i++)
3          A[i] = B[i];
4       } else {
5       for (int i=0; i < lg; i++)
6       A[i] = C[i];
        }
```

```
1    if (c){
2′      i1 = 0;
2       for( ;i2 = Φ(i1,i3),i2 < lg; i3 = i2 + 1)
3          A1[i2] = ...
        } else {
5′      i4 = 0;
5       for( ;i5 = Φ(i4,i6),i5 < lg; i6 = i5 + 1)
6          A2[i5] = ...
        }
7    A3 = Φ(A1,A2)
```

Suite: expliciter les @ et les Φ

```
1     if (c){
2′      i1 = 0;
2       for( ;i2 = Φ(i1,i3),i2 < lg; i3 = i2 + 1)
3         A1[i2] = ...
      } else {
5′      i4 = 0;
5       for( ;i5 = Φ(i4,i6),i5 < lg; i6 = i5 + 1)
6         A2[i5] = ...
      }
7     A3 = Φ(A1,A2)
```

```
1     if (c){
2′      i1 = 0;
2       for( ;i2 = Φ(i1,i3),i2 < lg; i3 = i2 + 1)
3         A1[i2] = ...
3′        @A1[i2] = <i2>;
      } else {
5′      i4 = 0;
5       for( ;i5 = Φ(i4,i6), i5 < lg; i6 = i5 + 1)
6         A2[i5] = ...
6′        @A2[i5] = <i5>;
      }
7     A3 = Φ(A1,A2);
7′    @A3 = max(@A1,@A2);
```

Suite: simplifier et conclure

```
1    if (c){
2′     i1 = 0;
2      for( ;i2 = Φ(i1,i3),i2 < lg; i3 = i2 + 1)
3        A1[i2] = ...
3′       @A1[i2] = <i2>;
     } else {
5′     i4 = 0;
5      for( ;i5 = Φ(i4,i6),i5 < lg; i6 = i5 + 1)
6        A2[i5] = ...
6′       @A2[i5] = <i5>;
     }
7    A3 = Φ(A1,A2);
7′   @A3 = max(@A1,@A2);
```

On va utiliser les informations pour optimiser

1. i_2 est une expression à incrément constant, et on peut remplacer i_2 par sa valeur. Idem pour i_5.
2. on propage ces valeurs dans 3′ et 6′.
3. on connait explicitement @A_1 et @A_2 en fonction de c.
4. leur calcul se fait en **parallèle**.
5. ceci permet d'expliciter la fonction $\Phi(A_1,A_2)$.
6. on élimine la fonction Φ en utilisant une exécution conditionnelle.

Suite: apres simplification

```
1    if (c){
3      A1[0 : (lg − 1)] = ...
3′     // @A1[0 : (lg − 1)] = c ? [0:(lg-1)] : ⊥;
     } else {
6      A2[0 : (lg − 1)] = ...
6′     // @A2[0 : (lg − 1)] = ¬ c ? [0:(lg-1)] : ⊥;
     }
7    A3[0:(lg-1)] = c ? A1[0:(lg-1)] : A2[0:(lg-1)];
7′   // @A3[0:(lg-1)] = [0:(lg-1)];
```

1. assignation parallèle ou vectorielle à A_1 et A_2.
2. les valeurs de i_* ne sont pas utilisées après, élimination de leur calcul.
3. on calcule symboliquement les valeurs des fonctions Φ et des @
4. on a placé en commentaire les informations sémantiques qu'il n'est pas utile de calculer à l'exécution.
5. on peut encore propager les valeurs de $A_{[1,2]}$[0:(lg-1)] dans une expression conditionnelle.

Evidemment, on n'a pas tout dit dans le transparent précédent

Comment a t'on résolu le cas précédent ?

- On peut effectuer la propagation des constantes, rechercher les expressions communes, éliminer le code mort en ce qui concerne les @ et les valeurs calculées par les Φ,
- le calcul sur les @ et les Φ est explicité dans [ASSA]. Par exemple il y est montré que pour des raisons sémantiques d'interprétation de code, les Φ possèdent des propriétés "algébriques" (associativité)
- l'élimination des expressions Φ est discutée dans [SSA], et de nombreuses améliorations sont disponibles. Ceci se fait en générant des instructions dans les blocs de code en relation avec la définition.
- dans certains cas , la parallélisation demande de calculer Φ à l'exécution, on gagne alors en étant capables de paralléliser dans des situations complexes.

Un second exemple un peu plus complexe

```
1     for (int i=0; i < lg; i++)
2       if (c[i]){
3         A[i] = B[i];
4       }else {
5         A[i] = C[i];
6       }
```

```
1'    int i_1=0;
1     for ( ; i_2= Φ(i_1,i_3), i_2 < lg; i_3=i_2+1){
2       if (c[i_2]){
3         A_1[i_2] = B[i_2];
3'        @A_1[i_2]=< i_2 >;
4       }else {
5         A_2[i_2] = C[i_2];
5'        @A_2[i_2]=< i_2 >;
        }
6       A_3=Φ(A_1,A_2);
6'      @A_3 = max(@A_1,@A_2);
7     }
```

Calcul des Φ et @

```
1′    int i1=0;
1     for ( ; i2= Φ(i1,i3), i2 < lg; i3=i2+1){
2         if (c[i2]){
3             A1[i2] = B[i2];
3′            @A1[i2]=< i2 >;
4         }else {
5             A2[i2] = C[i2];
5′            @A2[i2]=< i2 >;
          }
6         A3=Φ(A1,A2);
6′        @A3 = max(@A1,@A2);
7     }
```

1. On se rend compte que i est une expression à incrément constant, on connait sa valeur que l'on propage.
2. On calcule explicitement (symboliquement) @A_1[i]=c[i]?i:$\perp$, @A_2[i]=$\neg$c[i]?i:$\perp$.
3. On explicite la fonction $\Phi(A_1,A_2)$ de telle sorte que A_3[i] = c[i] ? A_1[i] : A_2[i].
4. Il apparaît maintenant que le code est parallélisable, la stratégie pouvant dépendre de l'architecture. Il suffit de posséder à l'exécution c[i] pour pouvoir calculer la fonction Φ à l'exécution. On a donc affaire à des instructions avec masque.....
5. Si besoin est on propage les valeurs de A_1 et A_2.

Synthèse du code à partir des Φ et @

3 where(c[0:(lg-1)]) A_1[0:(lg-1)] = B[0:(lg-1)];

3′ // @A_1[0:(lg-1)]= c[0:(lg-1)] ? [0:(lg-1)] : $\perp$;

5 where($\neg$c[0:(lg-1)]) A_2[0:(lg-1)] = C[0:(lg-1)];

5′ // @A_2[0:(lg-1)]=$\neg$c[0:(lg-1)] ? [0:(lg-1)] : $\perp$;

6 A_3= c ? A_1 : A_2;

6′ // @$A_3[0:(lg-1)]$ = [0:(lg-1)];

1. En vert: valeurs @ qui ne sont pas utilisées dans la suite, simple information
2. En rouge: valeur @ qui n'a d'intérêt que si elle est utilisée par la suite, sinon on peut l'oublier
3. Suivant le type d'architecture machine, il faut choisir une forme d'implantation parallèle.
4. Si on calcule tous les expressions second membre, au lieu d'utiliser une technique "scatter-gather", il peut être nécessaire de masquer les exceptions pour les termes "supplémentaires". On peut utiliser avec profi t les valeurs @ pour préciser cela.

Supprimer les fonctions Φ ?

Il y a deux options:

1. les éliminer:
   - solution proposée dans l'article original [SSA], consiste à faire remonter des assignations dans les divers blocs de base qui branchent vers la définition de la fonction Φ
   - divers algorithmes existent, se méfier des solutions naïves.
   - Le gain est qu'entre leur introduction et leur suppression on a réalisé un nombre de transformations et d'optimisations.
2. les calculer à l'exécution:
   - permet de représenter certaines situations requises pour le parallélisme, dont les instructions avec prédicats (vectoriels),
   - optimisation étudiée dans [ASSA].

Faire le point

- On a réussi à représenter la sémantique opérationnelle d'une manière qui se prête à l'application d'une grande variété d'algorithmes,
- Le formalisme vu ci-dessus est applicable aussi bien à du code de relativement haut niveau (fonctions) ou de bas niveau (code d'une machine virtuelle),
- Largement langage indépendant: le frontal peut traduire depuis le code d'origine dans le code de la machine virtuelle..
- Bonne efficacité algorithmique, même pour des algorithmes d'optimisation "globale"

Représentation SSA dans GCC

```
float puiss(const float x, int m){
  float prod = 1;
  float mult = x;
  while ( m ) {
    if ( m & 1){
      prod *= mult;
    }
    mult *= mult;
    m>>=1 ;
  }
  return prod;
}
```

```
        float puiss(float, int) (x, m){
        float mult; float prod;
        float D.30414; bool D.30413; int D.30412;
<bb 0>:
        prod_5 = 1.0e+0;
        mult_7 = x_6;
        goto <bb 4> (<L3>);
<L0>:;
        D.30412_11 = m_4  1;
        D.30413_12 = (bool) D.30412_11;
        if (D.30413_12) goto <L1>; else goto <L2>;
<L1>:;
        prod_15 = prod_2 * mult_3;
        -- prod_1 = PHI <prod_2(1), prod_15(2)>;
<L2>:;
        mult_13 = mult_3 * mult_3;
        m_14 = m_4 >> 1;
        -- m_4 = PHI <m_8(0), m_14(3)>;
        -- mult_3 = PHI <mult_7(0), mult_13(3)>;
        -- prod_2 = PHI <prod_5(0), prod_1(3)>;
<L3>:;
        if (m_4 != 0) goto <L0>; else goto <L4>;
<L4>:;
        D.30414_9 = prod_2;
        return D.30414_9;
}
```

GCC: Après optimisation et suppression des Φ

```
    float puiss(float, int) (x, m){
    float mult; float prod; float D.30414;
    bool D.30413; int D.30412;
<bb 0>:
    prod_5 = 1.0e+0; mult_7 = x_6;
    goto <bb 4> (<L3>);
<L0>:;
    D.30412_11 = m_4  1;
    D.30413_12 = (bool) D.30412_11;
    if (D.30413_12) goto <L1>; else goto <L2>;
<L1>:;
    prod_15 = prod_2 * mult_3;
    -- prod_1 = PHI <prod_2(1), prod_15(2)>;
<L2>:;
    mult_13 = mult_3 * mult_3; m_14 = m_4 >> 1;
    -- m_4 = PHI <m_8(0), m_14(3)>;
    -- mult_3 = PHI <mult_7(0), mult_13(3)>;
    -- prod_2 = PHI <prod_5(0), prod_1(3)>;
<L3>:;
    if (m_4 != 0) goto <L0>; else goto <L4>;
<L4>:;
    D.30414_9 = prod_2; return D.30414_9;
}
```

```
float puiss(float, int) (x, m){
    float mult;
    float prod;
<bb 0>:
    if (m != 0) goto <L14>; else goto <L15>;
<L15>:;
    prod = 1.0e+0;
    goto <L4>;
<L14>:;
    mult = x;
    prod = 1.0e+0;
<L0>:;
    if ((m  1) != 0) goto <L1>;
       else goto <L2>;
<L1>:;
    prod = mult * prod;
<L2>:;
    mult = mult * mult;
    m = m >> 1;
    if (m != 0) goto <L0>;
       else goto <L4>;
<L4>:;
    return prod;
}
```

Catalogue de transformations (1)

Transformations à haut niveau

- Inlining
- Elimination de la récursion
- Parallélisme
- Nids de boucles

Catalogue de transformations (1)

Transformations de base

- Simplifications algébriques
- Calcul des expressions constantes
- Elimination des redondances
  - déplacer les invariants de boucles
  - élimination des expressions communes
  - élimination des redondances partielles (calculées plusieurs fois sur un chemin d'exécution)
- Calcul des expressions à incrément constant (en vue du parallélisme et de la vectorisation).

Voir: [Dragon], [GnuSSA], [PFA], [PRE]

Troisième partie

Génération de code

On doit, *in fine*, générer du code.

Les développements précédents ont apporté:

1. une meilleure connaissance de la sémantique du code,
2. de nombreuses restructurations ont été faites,
3. de nombreuses optimisations ont été faites, essentielles, en particulier, pour supporter des langages de haut niveau (objet,....)

Néanmoins, il faut maintenant gérer toutes les contraintes liées à la machine, son architecture et ses performances.

Problèmes et méthodes

- Description et sélection des instructions
  - Méthodes: reconnaissance de “patterns” d’arbres et programmation dynamique. [CGGen]
  - En pratique, on construit un outil à partir d’une description du jeu d’instructions.
  - Le même type d’outil permet d’effectuer les optimisations “peephole” du code généré.
- Allocation des registres, Ordonnancement (cf. [Dragon]
  - ces deux problèmes sont NP-Complets (même pris séparément), en fait ils interfèrent et diverses méthodes heuristiques sont utilisées.
  - **allocation**: on utilise, par exemple, le coloriage du graphe d’interférence qui représente les interactions entre les durées de vie des valeurs.[CACC+]
  - Avec l’arrivée des processeurs à fort taux de parallélisme interne, l’ordonnancement interfère considérablement avec l’allocation des registres, demandant de nouvelles méthodes. [SP]

Portion de la description des instructions dans GCC

```
;; General case of fullword move.
(define_expand "movsi"
  [ (set (match_operand:SI 0 "nonimmediate_operand" "")
         (match_operand:SI 1 "general_operand" ""))
  ]
  ""
  "ix86_expand_move (SImode, operands); DONE;"
)

(define_expand "adddi3"
  [ (set (match_operand:DI 0 "nonimmediate_operand" "")
         (plus:DI (match_operand:DI 1 "nonimmediate_operand" "")
             (match_operand:DI 2 "x86_64_general_operand" "")))
    (clobber (reg:CC FLAGS_REG))
  ]
  ""
  "ix86_expand_binary_operator (PLUS, DImode, operands); DONE;"
)
```

```
(define_insn "*adddi3_1"
  [ (set (match_operand:DI 0 "nonimmediate_operand" "=r,o")
         (plus:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
             (match_operand:DI 2 "general_operand" "roiF,riF")))
    (clobber (reg:CC FLAGS_REG))
  ]
  "!TARGET_64BIT  ix86_binary_operator_ok (PLUS, DImode, operands)"
  ""
)
```

Pattern : à reconnaître

Effet de bord : effet de bord de l'action sémantique associée.

Informations d'ordonnancement générées par GCC

```
;;   ==========================================================
;;   -- basic block 2 from 114 to 37 -- after reload
;;   ==========================================================

;;   0--> 112  si=[bp+0xc]                          :decodern,p2
;;   0--> 114  dx=cx*0x4                            :decodern,p0
;;   0--> 34   {cx=cx+0x1;clobber flags;}           :decodern,p0|p1
;;   1--> 36   flags=cmp(bx,cx)                     :decodern,p0|p1
;;   4--> 32   ax=[si+dx]                           :decodern,p2
;;   8--> 33   [di+dx]=ax                           :decoder0,(p4+p3)
;;   8--> 37   pc={(flags>0x0)?L26:pc}              :decodern,p1
;;   total time = 8
```

Code machine produit par GCC 4.0.0

```
float puiss(const float x, int m){
   float prod = 1;
   float mult = x;
   for ( ; m ; mult *= mult, m>>=1 ){
      if ( m & 1)
         prod *= mult;
      }
   }
return prod;
```

```
_Z5puissfi:
.LFB1724:
            pushl    %ebp
.LCFI3:
            movl     %esp, %ebp
.LCFI4:
            fld1
            movl     12(%ebp), %edx
            flds     8(%ebp)
            testl    %edx, %edx
            je       .L16
            .p2align 4,,7
.L14:
            movl     %edx, %ecx
            andl     $1, %ecx
            testb    %cl, %cl
            je       .L8
            fmul     %st, %st(1)
.L8:
            sarl     %edx
            fmul     %st(0), %st
            jne      .L14
.L16:
            fstp     %st(0)
            popl     %ebp
            ret
```

Support des caractéristiques architecturales

- Vectorisation
- Conditionnement
- Prefetch
- Spéculation

Tout ce que vous voulez apprendre sur ces sujets dans l'exposé de F. Bodin, et bien plus.

Les nouveaux challenges

1. facilité de programmation, au moins pour l'utilisateur,
   - langages évolués,
   - types abstraits, encapsulation, énéricité,
   - orientation objet,
   - bibliothèques de composants génériques.
2. portabilité, outils de construction de bibliothèques,
3. prise en compte de l'architecture,
4. accès à toutes les formes de parallélisme.

Et les performances ?

Possibilité de performances importantes avec des langages évolués, savoir faire réparti entre compilateur, run-time, bibliothèques.

Exemples (en C++):

1. template meta programming, illustré dans PETE[a],
2. implémentation d'une bibliothèque à l'aide des extensions SSE, sans impact sur l'interface.

[a] LANL

Exemple de performances avec les SSE et GCC 4.0

Produits scalaires, vecteurs alignes 16 octets

Pentium 4 Mobile Intel(R) Pentium(R) 4 - M CPU 1.80GHz

Cache size : 512 KB

iterations=250000 longueur=800

3333.33 MOp flottantes with SSE (hand unrolled x 2)/s

3333.33 MOp flottantes with SSE (hand unrolled x 2 + prefetch)/s

2857.14 MOp flottantes with SSE prodscal2/s

253.165 MOp flottantes without SSE (deroule main)/s

130.293 MOp flottantes without SSE/s

Le code généré par GCC 4.0 pour la boucle interne (version la plus rapide...)

Boucle interne:

```
.L23:   movss   (%ebx), %xmm1
        mulss   (%esi), %xmm1
        movss   4(%eax), %xmm0
        mulss   4(%edx), %xmm0
        addss   %xmm0, %xmm1
        addss   %xmm1, %xmm2
        addl    $2, %edi
        addl    $8, %esi
        addl    $8, %ebx
        addl    $8, %edx
        addl    $8, %eax
        movl    -32(%ebp), %ecx
        addl    %edi, %ecx
        cmpl    -36(%ebp), %ecx
        jl      .L23
        movl    %ecx, -32(%ebp)
        movl    %ecx, %ebx
```

Le programme source:

```
float prodscal(Vect_Float_Aligned a, Vect_Float_Aligned  b)
    long longueur = a.size();
    float accum= 0.0;
    long position = 0;
    if (longueur >= 20){
        vect_4 load_a1, load_a2, load_b1, load_b2,
           mult_1, mult_2, accu_1 , accu_2;
        for (position=0; position < longueur-7; position+=8){
             load_a1.load_aligned(&a[position]);
             load_b1.load_aligned(&b[position]);
             mult_1.mul(load_a1, load_b1);
             load_a2.load_aligned(&a[position+4]);
             accu_1.add(accu_1,mult_1);
             load_b2.load_aligned(&b[position+4]);
             mult_2.mul(load_a2, load_b2);
             accu_2.add(accu_2,mult_2);
        }
        accu_1.add(accu_1, accu_2);
        vecteur4 v4;
        accu_1.store(v4);
        accum = v4[0]+v4[1]+v4[2]+v4[3];
    }
    for( ; position< (longueur-1); position+=2){
        accum+=(a[position]*b[position]+a[position+1]*b[position+1]
    }
    if ( position < longueur){
        accum += a[position]*b[position];
    }
    return accum;
}
```

En fait, une grande partie de la construction de ce programme a consisté à faire, à la main, le travail d'un vectoriseur. Ceci peut donc être automatisé également.

Conclusion

J'espère avoir contribué a démystifier un peu ces logiciels

- Voir bibliographie.
- Evolution rapide des produits commerciaux et des logiciels libres.
- Domaines d'applications variés, vont des aides à la programmation à la synthèse de systèmes spécialisés et de réalisations "flexibles" (FPGA).

Bibliographie (1)

Les principaux ouvrages

- [MA] **E. G. Manes & M. A. Arbib**,*Algebraic Approaches to Program Semantics*,Springer Verlag,1986
- [Dragon] **A.V. Aho, R. Sethi, J. D. Ullman**,*Compilers: principles, techniques and tools*,Addison Wesley,1986
- [PFA] **S. S. Muchnick & N. D. Jones eds.**,*Program Flow Analysis*,Prentice Hall Software Series,1981

Bibliographie (2)

Et quelques articles

- [SSA] **R. Cytron, J. Ferrante, B. K. Rosen & M. N. Wegman**,*Efficiently computing the Static Single Assignment Form and the Control Dependence Graph*,ACM TOPLAS, Vol 13 No 4,1991
- [ASSA] **K. Knobe & V. Sarkar**,*Array SSA form and its use in Parallelization*,ACM POPL-98,1998
- [Ptr] **C.Lapkowski & L.JL Hendren**,*Extended SSA numbering: Introducing SSA numbering to languages with multi level pointers*,,
- [PRE] **F. Chow, S. Chan, K. Kennedy, S-M. Liu, R. Lo & P. Tu**,*A New Algorithm for Partial Redundancy Elimination based on SSA Form*,ACM PLDI'97,1997
- [CGGen] **C.W. Fraser, D.R. Hanson & T.A. Proebsting**,*Engineering a Simple, Efficient*

Code Generator Generator,ACM Letters Programming Languages & Systems, Vol 1, No 3,1992

- [SP] **S. S. Pinter**,*Register Allocation with Scheduling: a new approach*,SIGPLAN PLDI '93,1993
- [CACC+] **G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins & P.W. Markst**, *Register Allocation and Spilling via Graph Coloring*,Computer Languages, 6:47-57,1981
- [ASched] **V.N. Makarov**,*The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC*,,
- [EGS] **C. Eisenbeis, F. Gasperoni & U. Schwiegelshohn**,*Allocating registers in multiple instruction issuing processors*,Rapport Recherche INRIA, No 2628,1995

Bibliographie (3)

Sur le Net

- [Sassa] **M. Sassa**,*Static Single Assignment Form*,*(Cours détaillant les transformations pour former la SSA et générer le code ensuite, en éliminant les fonctions Φ. Références au projet COINS.)*, http://www.is.titech.ac.jp/ sassa/coins-www-ssa/english/ssa-lecture-80.pdf
- [GnuSSA] **D. Novillo**,*Tree SSA - A high level optimization framework for the Gnu Compiler Collection*,*(Implémentation SSA dans GCC, de nombreux détails sur l'architecture de GCC.)*, http://www.linux.org.uk/ ajh/gcc/gccsummit-2003-proceedings.pdf
- [CC] **Compiler Connection**,*A resource for compiler developpers*,*(Beaucoup de liens sur les domaines académiques et professionnels.)*, http://www.compilerconnection.com/
- [WGCC] **Page de liens du projet GCC**,,*()*, http://www.gnu.org/software/gcc/readings.html
- [HIPEAC] **Projet Européen HIPEAC**,*European Network of Excellence on High*

Performance Embedded Architecture and Compilation,(), http://escher.elis.ugent.be/hipeac/

- [CUJ] ,*C/C++ Users Journal*,(), http://www.cuj.com