Université de Strasbourg

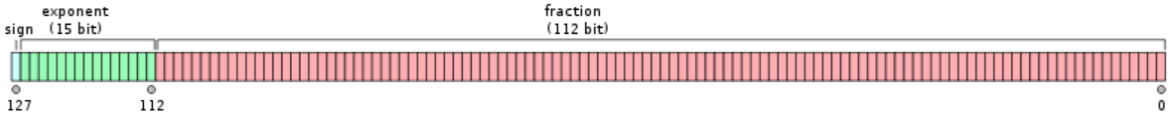
Master Informatique
Projet de Compilation
2018/2019

C2MP : Le calcul flottant multi-précision pour tous !

Le but du projet est de réaliser un compilateur source-à-source d'un programme en C vers un autre programme en C, où certains calculs arithmétiques ont été transformés, afin que ces calculs s'effectuent en virgule flottante de précision arbitraire, grâce à l'utilisation d'une bibliothèque dédiée.

1 Introduction

Il est connu que l'un des problèmes majeurs de la technologie actuelle des ordinateurs est sa capacité limitée de représentation des données numériques, en termes de précision et de grandeur. Même si la norme standard IEEE 754 de représentation en virgule flottante continue à évoluer, proposant aujourd'hui la précision quadruple (128 bits, voir le schéma ci-dessous) et même la précision octuple (256 bits), les compilateurs prennent du temps à adopter ces nouvelles normes. De plus, même la précision octuple peut être insuffisante dans les cas où une précision encore plus importante est nécessaire.



Ainsi, la seule manière de profiter de tels niveaux de précision, et plus encore, est d'utiliser une bibliothèque dédiée. En effet, il existe plusieurs bibliothèques permettant de profiter d'une précision arbitraire, comme par exemple GMP [1], MPFR [2] ou MPC [3]. Il n'y a a priori pas de limite quant à la précision choisie, à l'exception de la quantité de mémoire disponible sur la machine où s'exécute le programme.

Pour ce projet, nous nous limiterons aux bibliothèques MPFR et MPC.

2 GNU MPFR et MPC

La bibliothèque MPFR est dédiée aux calculs sur les nombres réels, alors que la bibliothèque MPC est dédiée aux calculs sur les nombres complexes. La programmation MPFR ou MPC d'une expression arithmétique est très fastidieuse et rebute beaucoup d'utilisateurs potentiels. En effet, chaque opération présente dans l'expression doit être réalisée séparément en une instruction d'appel à la bibliothèque. Par exemple, l'expression arithmétique suivante écrite en C :

```
double complex resultat;
resultat = -(sqrt(-NMAX*(8*pc-4*pow(NMAX,3)-4*pow(NMAX,2)-NMAX-8))-2*pow(NMAX,2)-NMAX)/(2*NMAX);
```

doit être traduite de la manière suivante si on veut utiliser la bibliothèque MPC :

```
double complex resultat;

/* déclarations de variables temporaires et de leur précision (128 dans cet exemple) */
mpc_t T0; mpc_init2(T0,128);
mpc_t T1; mpc_init2(T1,128);
mpc_t T2; mpc_init2(T2,128);
mpc_t T3; mpc_init2(T3,128);
mpc_t T4; mpc_init2(T4,128);
mpc_t T5; mpc_init2(T5,128);
mpc_t T6; mpc_init2(T6,128);
mpc_t T7; mpc_init2(T7,128);
```

1. https://gmplib.org
2. https://www.mpfr.org
3. http://www.multiprecision.org/mpc

```
/*      calculs intermédiaires, avec arrondi vers zéro
        pour les parties réelles et imaginaires dans
        cet exemple (MPC_RNDZZ)                 */
mpc_set_si(T0,NMAX,MPC_RNDZZ);          /* T0 = NMAX            */
mpc_neg(T1,T0,MPC_RNDZZ);               /* T1 = -T0             */
mpc_set_si(T2,8,MPC_RNDZZ);             /* T2 = 8               */
mpc_set_si(T3,pc,MPC_RNDZZ);            /* T3 = pc              */
mpc_mul(T3,T2,T3,MPC_RNDZZ);            /* T3 = T2 * T3         */
mpc_set_si(T4,4,MPC_RNDZZ);             /* T4 = 4               */
mpc_set_si(T5,3,MPC_RNDZZ);             /* T5 = 3               */
mpc_pow(T5,T0,T5,MPC_RNDZZ);            /* T5 = T0^T5           */
mpc_mul(T5,T4,T5,MPC_RNDZZ);            /* T5 = T4 * T5         */
mpc_sub(T5,T3,T5,MPC_RNDZZ);            /* T5 = T3 - T5         */
mpc_set_si(T6,2,MPC_RNDZZ);             /* T6 = 2               */
mpc_pow(T7,T0,T6,MPC_RNDZZ);            /* T7 = T0^T6           */
mpc_mul(T4,T4,T7,MPC_RNDZZ);            /* T4 = T4 * T7         */
mpc_sub(T4,T5,T4,MPC_RNDZZ);            /* T4 = T5 - T4         */
mpc_sub(T4,T4,T0,MPC_RNDZZ);            /* T4 = T4 - T0         */
mpc_sub(T4,T4,T2,MPC_RNDZZ);            /* T4 = T4 - T2         */
mpc_mul(T4,T1,T4,MPC_RNDZZ);            /* T4 = T1 * T4         */
mpc_sqrt(T4,T4,MPC_RNDZZ);              /* T4 = sqrt(T4)        */
mpc_mul(T7,T6,T7,MPC_RNDZZ);            /* T7 = T6 * T7         */
mpc_sub(T7,T4,T7,MPC_RNDZZ);            /* T7 = T4 - T7         */
mpc_sub(T7,T7,T0,MPC_RNDZZ);            /* T7 = T7 - T0         */
mpc_neg(T7,T7,MPC_RNDZZ);               /* T7 = -T7             */
mpc_mul(T6,T6,T0,MPC_RNDZZ);            /* T6 = T6 * T0         */
mpc_div(T6,T7,T6,MPC_RNDZZ);            /* T6 = T7 / T6         */

/* récupération du résultat final */
resultat = mpc_get_ldc(T6,MPC_RNDZZ);

/* désallocation de la mémoire des variables temporaires */
mpc_clear(T0);
mpc_clear(T1);
mpc_clear(T2);
mpc_clear(T3);
mpc_clear(T4);
mpc_clear(T5);
mpc_clear(T6);
mpc_clear(T7);
```

Une telle suite d'instructions est similaire à un code intermédiaire d'instructions à trois adresses des compilateurs. Notons qu'une telle traduction n'est pas unique, et que plusieurs décompositions correctes en opérations élémentaires sont possibles, la précision élevée permettant d'assurer une certaine stabilité du résultat final, quelque soit cette décomposition. On remarquera aussi que dans la traduction proposée, le nombre de variables temporaires (8), est inférieur au nombre d'instructions (24). En effet, certaines variables sont réutilisées dès que leur valeur précédente devient inutile. Une telle optimisation est obtenue par *analyse des durées de vie des variables.*

3 But du projet

L'objectif est de développer un compilateur source-à-source afin de rendre accessible à tous le calcul flottant en précision quelconque. Ce compilateur prendra en entrée un programme C qui contient des blocs d'instructions ciblés par un pragma, et produira en sortie un autre programme C où les calculs arithmétiques de ces blocs ont tous été transformés en appels soit à la librairie MPFR, soit à la librairie MPC. Le pragma permettra à l'utilisateur d'indiquer la bibliothèque qu'il désire utiliser, la précision et le mode d'arrondi. Par exemple, à partir d'un programme C contenant l'extrait suivant :

```
...
#pragma MPC precision(128) rounding(MPC_RNDZZ)
{
  resultat = -(sqrt(-NMAX*(8*pc-4*pow(NMAX,3)-4*pow(NMAX,2)-NMAX-8))-2*pow(NMAX,2)-NMAX)/(2*NMAX);
}
..
```

Le compilateur devra générer un nouveau programme où l'instruction aura été remplacée par une suite d'appels à MPC comme ci-dessus.

Les blocs ciblés par le pragma pourront contenir tout type de structure de contrôle du langage C (for, while, if, ...) ainsi que des appels de fonctions. Les opérandes des expressions arithmétiques ciblées pourront être elles-mêmes des appels de fonctions. Il n'est toutefois pas demandé de modifier les corps des fonctions appelées si elles sont opérandes d'expressions. Celles-ci devront être considérées de la même manière que les variables ou les constantes, via une initialisation d'une variable temporaire MPC ou MPFR à leur valeur de retour.

Tout calcul arithmétique apparaissant dans un bloc d'instructions ciblé devra être converti. Par exemple, si l'on considère l'extrait de programme suivant :

```
#pragma MPC precision(128) rounding(MPC_RNDZZ)
{
  while ( (pow(a,3) - sqr(c) + 12 > 0) && (a * 46 - pow(b,2) < 3.14) ) {
    a = (pow(a,3)+pow(a,2))/2 ;
  }
}
```

Il devra être converti en un programme similaire à celui-ci :

```
mpc_t T0; mpc_init2(T0,128);
mpc_t T1; mpc_init2(T1,128);
mpc_t T2; mpc_init2(T2,128);
mpc_t T3; mpc_init2(T3,128);
mpc_t T4; mpc_init2(T4,128);
mpc_t T5; mpc_init2(T5,128);
mpc_t T6; mpc_init2(T6,128);
mpc_t T7; mpc_init2(T7,128);
mpc_t T8; mpc_init2(T8,128);

/* calcul de l'expression pow(a,3) - sqr(c) + 12 */
mpc_set_si(T0,a,MPC_RNDZZ);
mpc_set_si(T1,3,MPC_RNDZZ);
mpc_pow(T1,T0,T1,MPC_RNDZZ);
mpc_set_si(T2,c,MPC_RNDZZ);
mpc_sqrt(T2,T2,MPC_RNDZZ);
mpc_sub(T2,T1,T2,MPC_RNDZZ);
mpc_set_si(T3,12,MPC_RNDZZ);
mpc_add(T3,T2,T3,MPC_RNDZZ);

/*  calcul de l'expression 0 */
mpc_set_si(T4,0,MPC_RNDZZ);

/*  calcul de l'expression a * 46 - pow(b,2) */
mpc_set_si(T5,46,MPC_RNDZZ);
mpc_mul(T5,T0,T6,MPC_RNDZZ);
mpc_set_si(T6,b,MPC_RNDZZ);
mpc_set_si(T7,2,MPC_RNDZZ);
mpc_pow(T7,T6,T8,MPC_RNDZZ);
mpc_sub(T7,T5,T7,MPC_RNDZZ);

/*  calcul de l'expression 3.14 */
mpc_set_d(T8,3.14,MPC_RNDZZ);

while( mpc_cmp(T3,T4) && !mpc_cmp(T8,T9) ) {

/* calcul de l'expression (pow(a,3)+pow(a,2))/2 */
mpc_set_si(T10,3,MPC_RNDZZ);
mpc_pow(T10,T0,T10,MPC_RNDZZ);
mpc_set_si(T11,2,MPC_RNDZZ);
mpc_pow(T9,T0,T11,MPC_RNDZZ);
mpc_add(T9,T10,T9,MPC_RNDZZ);
mpc_div(T9,T9,T11,MPC_RNDZZ);
a = mpc_get_ldc(T9,MPC_RNDZZ);
```

```
/* calcul de l'expression pow(a,3) - sqr(c) + 12 */
... comme ci-dessus ...

/*  calcul de l'expression a * 46 - pow(b,2) */
... comme ci-dessus ...

}

mpc_clear(T0);
mpc_clear(T1);
mpc_clear(T2);
mpc_clear(T3);
mpc_clear(T4);
mpc_clear(T5);
mpc_clear(T6);
mpc_clear(T7);
mpc_clear(T8);
mpc_clear(T9);
mpc_clear(T10);
mpc_clear(T11);
```

Dans cet exemple, on remarquera que les affectations des variables T4 et T8 ne sont plus effectuées dans la boucle. En effet, ces instructions sont invariantes : leur exécution dans la boucle ne modifierait pas leur résultat. De la même manière, ce programme pourrait encore être amélioré. Par exemple, les instructions servant aux calculs des sous-expressions `- sqr(c) + 12` et `pow(b,2)` pourraient être également évitées, les valeurs de `b` et de `c` n'étant pas modifiées dans la boucle. Mais cela est malheureusement impossible dans la forme actuelle du programme, car les variables qui stockent leurs valeurs sont réutilisées pour d'autres valeurs. Il faut donc les recalculer.

Le compilateur source-à-source C2MP devra, en plus de la traduction vers MPFR ou MPC, effectuer les optimisations suivantes :

1. Minimiser le nombre d'instructions MPFR ou MPC à travers *l'élimination de sous-expressions communes*. Par exemple, le compilateur devra remarquer que l'expression :

   `45*(sqrt(a-pow(b,3))+pow(c,1/3))/(sqrt(a-pow(b,3))+pow(c,1/3))`

   contient deux fois la sous-expression `sqrt(a-pow(b,3))+pow(c,1/3)`, et qu'une seule évaluation suffit.
2. Minimiser le nombre de variables temporaires utilisées dans les instructions MPFR ou MPC, à travers *l'analyse des durées de vie des variables*.
3. Déplacer *les invariants de boucle* en dehors des boucles, en détectant les instructions pour lesquelles leur exécution dans la boucle ne modifie pas leur résultat.

4 Aspects pratiques et techniques

Le compilateur C2MP devra être écrit en C à l'aide des outils Lex et Yacc. Les bibliothèques MPFR et MPC peuvent être installées sous Debian/Ubuntu en installant les packages `libmpfr-dev` et `libmpc-dev`. Tout programme C utilisant ces bibliothèques devra inclure les fichiers d'entête `mpfr.h` et `mpc.h`. Attention, le fichier d'entête `complex.h`, s'il est utile, devra toujours être inclus *avant* les deux fichiers précédents. Les programmes C incluant des appels à MPFR ou à MPC doivent être compilés (avec gcc ou clang) en utilisant les options `-lmpfr` et/ou `-lmpc`. Les documentations décrivant toutes les instructions des bibliothèques MPFR et MPC peuvent être téléchargées à partir de leur site internet.

Le compilateur devra être capable de prendre en compte les structures de contrôle du langage C (for, while, if, ...), ainsi que les appels aux fonctions mathématiques telles que `pow, powf, powl, cpow, cpowf, cpowl, sqrt, sqrtf, sqrtl, csqrt, csqrtf, csqrtl, sin, sinf, sinl, csin, csinf, csinl, log, exp`, ... en les convertissant au mieux en des appels aux fonctions correspondantes des bibliothèques MPFR ou MPC.

Ce travail est à réaliser en équipe dans le cadre du cours de *Compilation* et du cours de *Conduite de Projets*, et à rendre à la date indiquée par vos enseignants en cours et sur Moodle. Une démonstration finale de votre compilateur sera faite durant la dernière séance de TP. Vous devrez rendre sur Moodle dans une archive :

- Le code source de votre projet complet dont la compilation devra se faire simplement par la commande « make ».
- Un document détaillant les capacités de votre compilateur, c'est-à-dire ce qu'il sait faire ou non. Soyez honnêtes, indiquez bien les points intéressants que vous souhaitez que le correcteur prenne en compte car il ne pourra sans doute pas tout voir dans le code.
- Un jeu de tests.

5 Recommandations importantes

Écrire un compilateur est un projet conséquent, il doit donc impérativement être construit incrémentalement en validant chaque étape sur un plus petit langage et en ajoutant progressivement des fonctionnalités ou optimisations. Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis de passer au débogage ! Le résultat de cette démarche serait très probablement nul, c'est-à-dire un compilateur qui ne fonctionne pas du tout ou alors qui reste très bogué.

Par conséquent, nous vous conseillons de développer tout d'abord un compilateur *fonctionnel* mais *limité* à la traduction d'expressions arithmétiques simples, sans structures de contrôle et sans optimisations (par exemple un compilateur qui utilise une variable temporaire pour chacune des instructions MPFR/MPC, et qui recalcule toutes les sous-expressions communes). À partir d'une telle version fonctionnelle, il vous sera plus aisé de la faire évoluer en intégrant telle ou telle optimisation, ou en considérant des expressions plus complexes, ou en intégrant telle ou telle structure de contrôle. De plus, même si votre compilateur ne remplira finalement pas tous les objectifs, il sera néanmoins capable de générer des programmes corrects et qui «marchent» !

6 Précisions concernant la notation

- Si votre projet ne compile pas ou plante directement, la note 0 (zéro) sera appliquée : l'évaluateur n'a absolument pas vocation à aller chercher ce qui pourrait éventuellement ressembler à quelque chose de correct dans votre code. Il faut que votre compilateur s'exécute et qu'il fasse quelque chose de correct, même si c'est peu.
- Si vous manquez de temps, préférez faire moins de choses mais en le faisant bien et de bout en bout : on préférera un compilateur incomplet mais qui génère un programme C compilable et exécutable à une analyse syntaxique seule.
- Élaborez des tests car cela fait partie de votre travail et ce sera donc évalué.
- Faites les choses dans l'ordre et focalisez sur ce qui est demandé. L'évaluateur pourra tenir compte du travail fait en plus (optimisations supplémentaires du programme généré par exemple) seulement si ce qui a été demandé a été fait et bien fait.
- Une conception modulaire et lisible sera fortement appréciée (et inversement).
- Étant donnée l'ampleur de la tâche, on ne demande pas le passage par un arbre de syntaxe abstraite entre analyse syntaxique et génération d'instructions MPFR/MPC, mais ceux qui le feront correctement seront des héros.