Institut de Recherche en Informatique de Toulouse
CNRS
INPT
UPS
UT1

Normalisation de boucles pour l'évaluation du nombre maximum d'itérations

Marianne de Michiel, Armelle Bonenfant, Pascal Sainrat, Hugues Cassé
Institut de Recherche en Informatique de Toulouse
118, route de Narbonne
31062 Toulouse Cedex
michiel@irit.fr

Rapport interne n° : IRIT/RR--2008-3--FR

1 Introduction

Une étape importante de l'analyse du pire temps d'exécution d'un programme (WCET) est de déterminer les bornes maximales du nombre d'itérations des boucles. Ces bornes sont cruciales afin de vérifier qu'un système temps-réel satisfait ces contraintes temporelles. A notre connaissance, il n'existe pas actuellement de méthode automatique permettant de déterminer ces bornes pour toutes les boucles. C'est pourquoi, les outils d'analyse du WCET permettent d'introduire des annotations manuelles pour indiquer ces bornes. Cependant, il est souvent laborieux et source d'erreurs de fournir manuellement ces annotations. Des études montrent qu'il est important de développer des méthodes d'analyse afin de calculer aussi automatiquement que possible ces informations afin de réduire les besoins en annotations manuelles.

Nous décrivons ici les travaux relatifs au calcul de bornes de boucles, puis les principes de notre propre méthode avant d'en justifier l'étape préliminaire : la normalisation de boucles.

1.1 Etat de l'art

Plusieurs travaux se sont intéressés à trouver des bornes de boucles. Ces méthodes diffèrent par :

- le langage source auquel elles s'appliquent (C, RTL, Fortran...),
- l'intégration ou non d'instructions de rupture de séquence,
- les types de conditions de boucle prises en compte (<, <=, >, >=, = , !=, &&, ||),
- les types d'incréments considérés (+k ou -k seuls, incréments multiples...)
- la prise en compte ou non du contexte (sans contexte, pas d'appel de fonction et seules les bornes constantes sont considérées),
- le traitement de boucles imbriquées (nids de boucles).

Dans [1] , Healy et al présentent comment ils analysent une représentation RTL d'un programme. Ils construisent des expressions des bornes de boucles et les évaluent en utilisant un solveur algébrique. Ils prennent en compte les multiples sorties (exit) des boucles mais restreignent leur étude aux variations constantes de la variable de boucle de type +k ou –k (avec k constante) mais ne savent pas résoudre les variations de type ×k ou /k (avec k constante). Ils prennent en compte les conditions de boucle utilisant les opérateurs <, <=, >, >=, = et !=, mais ni des && ni des ||. Leur analyse se restreint aux boucles dont le nombre d'itérations est bornée par une constante ou directement une/des variable(s) de boucle(s) englobante(s). Les valeurs initiales et limites des variables de boucle sont des constantes ou des variables de boucles englobantes. Les formules des boucles internes sont des sommations.

Les travaux de [1] ont été repris par Kiner [2] afin de les adapter à l'analyse de programmes sources en C. Il ne considère cependant pas les nids de boucles. Il introduit des annotations dans le code source permettant de donner pour chaque boucle ses bornes pour une itération de la boucle englobante. Si aucune borne n'a été trouvée, l'analyse peut se référer aux annotations de l'utilisateur.

[1] et [2] recherchent des expressions de bornes de boucle et les évaluent en utilisant un solveur algébrique. Cependant leurs évaluations sont indépendantes du contexte dans lesquels les nids de boucles sont appelés.

Bound-T [3], un outil industriel, propose une analyse des bornes de boucle basée sur les résultats du projet Omega. Cependant, ce projet étudie les nids de boucle Fortran (boucles for) mais ne prend pas en compte les appels de fonction ni les flots de contrôle arbitraires.

aiT [4], est un autre outil industriel. Il utilise l'interprétation abstraite sur des intervalles et l'analyse

des flots de données mais il se limite à des variables de boucle modifiées par des incréments de type +k ou –k (avec k constante).

Ermedahl et al [5], proposent d'appliquer l'interprétation abstraite à des intervalles représentant les valeurs possibles d'une variable. Par exemple, i=[1..20] représente tous les états de 1<=i<=20 (i pouvant être une entrée du programme). L'utilisateur peut lui-même fournir un intervalle de variation pour une variable si il ne peut pas être déterminé automatiquement. Ils combinent cette analyse avec une interprétation des variations des variables de boucle appelée analyse des congruences afin de prendre en compte certains des incrément variables.

Les méthodes qui utilisent l'interprétation abstraite présentent l'avantage de pouvoir prendre en compte au moins partiellement le contexte d'appel des boucles et donc ne se limitent plus à des bornes constantes même s'il existe toujours des restrictions notamment par rapport aux types d'incrément possibles, aux types de conditions...

1.2 Notre méthode

Nous avons développé une méthode statique de recherche des bornes de boucles.

1.2.1 Principe

Cette méthode se base sur l'analyse du flot d'exécution du programme et sur l'interprétation abstraite. Elle intègre donc le contexte (appels de fonctions mais non récursives). Elle traite les programmes C corrects par hypothèse, ne prend pas en compte les instructions de rupture de séquence. Ces restrictions, pourront être relaxées par l'utilisation du simplificateur de code Calipso [6].

Cette analyse prend en compte les incréments de boucle de type (+, -, ×, /) constante de boucle c'est-à-dire non modifiés par la boucle mais non nécessairement constant pour le programme, les nids de boucles, les conditions de boucle de type ==, !=,<, <=, >, >=, et certains &&.

Nous nous situons ici dans un contexte où la boucle n'est pas interrompue par une instruction de rupture de séquence (exit...) (utilisation de Calipso). Comme nous recherchons un nombre maximum d'itérations des boucles, cette limite n'est pas pénalisante.

1.2.2 Description

Notre méthode est basée sur la recherche d'une expression du nombre d'itérations de chaque boucle indépendamment de ces boucle englobantes. Cette expression se rapproche de l'expression construite dans [1-2] mais peut comporter des variables. A partir de cette expression on construit une forme normalisée des boucles. Ceci constitue la première étape de notre méthode d'évaluation.

La seconde étape construit une expression représentant le nombre d'itération de chaque boucle d'un nid de boucle en fonction de leur dépendance et utilise l'interprétation abstraite pour connaître les valeurs possibles (expressions) des variables en fonction du contexte d'appel du nid. Ceci constitue l'étape 2 de notre analyse.

Enfin la dernière étape correspond à l'évaluation des expressions ainsi obtenues.

Cette méthode est donc constituée de trois phases :

• Phase 1, nous construisons l'arbre de contexte annoté du programme. Un arbre de contexte est un arbre donc les nœuds sont soit des boucles, soit des appels de fonction. Les boucles dans notre arbre sont des boucles normalisées, les annotations de l'arbre sont utilisées à l'étape suivante pour évaluer les bornes de la boucle pour chaque appel.

• Phase 2, nous parcourons cet arbre afin d'associer à chaque boucle de l'arbre une expression de ces bornes en fonction de son contexte (appel de fonction, boucle englobantes ...). L'expression de cette

borne est construite en combinant l'expression de la borne déterminée lors de la première phase avec celle d'éventuelles boucles englobantes et en considérant les différents contextes d'appel au moyen de l'interprétation abstraite.

• Phase 3, pour chaque appel de boucle, on s'attache à évaluer les expressions construites.

1.2.3 La normalisation de boucle

La première étape de ce travail consiste donc à associer à chaque boucle une forme normalisée permettant de définir une expression de son nombre d'itérations indépendamment de son contexte. Lors de cette phase ne sont recherchés que la valeur initiale de la variable de boucle, son incrément et sa valeur limite en utilisant des résultats d'interprétation abstraite. L'avantage de la normalisation est de transformer une boucle initiale en une boucle équivalente mais pour laquelle la variable de boucle parcourt toutes les valeurs de l'intervalle [0..borneMax] avec un incrément constant de 1. Ceci permet en particulier de simplifier les calculs dans les cas de nids de boucles dépendantes.

Pour valider la normalisation et en exposer les limites, nous allons, dans un premier temps, étudier les boucles `for` ayant un incrément de type `+k` ou `-k`, puis étendre les résultats aux boucle `while` et `do ...while`. Nous étudierons ensuite les incréments de type `×k` ou `/k`. Enfin nous présenterons quelques extensions.

2 Caractérisation des boucles non imbriquées

Dans une première partie, nous allons étudier les boucles for de type

```
for (i = k; i <= n; i = i+c)
```

avec comme hypothèses : `k>=0, n>=0, c>0` ainsi que les boucles assimilables à celles-ci. Ces boucles sont appelées **boucles arithmétiques**.

Dans une seconde partie, nous allons étudier les boucles

```
for (i = k; i <= n; i = i*c)
```

avec comme hypothèses : `k >0, n>=0, c>1` et `c` entier ainsi que les boucles assimilables à celles-ci.

Ces boucles sont appelées **boucles géométriques**.

3 Etude des boucles arithmétiques

Notre premier objectif est de nous ramener d'une boucle de type
`for (i = k; i <= n; i += c)` avec comme hypothèses : `k>=0; n>=0 c>0`
à une boucle de type
`for (j = 0; j <= nombreIT-1; j++)` dite boucle normalisée.

Soit une boucle du type : `for (i = k; i <= n; i = i+c)`,
l'écriture `for (i = k; i <= n/c ; i++)` n'est pas équivalente en terme de nombre d'itérations.
Par exemple, les boucles `for (i=1; i<=9 ; i+=5)` et `for (i=1; i<=9/5 ; i+=1)` n'ont pas le même nombre d'itérations. Les valeurs de `i` pour lesquelles on passe dans la boucle pour la première sont 1 et 6, on a donc 2 itérations alors que pour la seconde boucle, on a une seule itération, la valeur de `i` étant 1.

Avant de modifier le pas de boucle, il faut donc procéder à tous les changements de repères nécessaires de manière à commencer le nombre d'itérations à 0 .

Ici, nous proposons une méthode de normalisation pour les boucles `for` du type

`for (i=k; i<=n; i+=c)` puis `for (i=k; i<n; i+=c)`.

3.1 Normalisation d'une boucle for d'incrément + valeur

Nous proposons donc une méthode de transformation pour les boucles `for` du type :
for (i=k; i<=n; i+=c) avec comme hypothèses : `k >=0, n>=0, c >0` en une boucle
for (j = 0; j<=nombreIT-1; j++) ayant le même nombre d'itérations.

- **Démarrage de la valeur initiale à 0**

  Etudions le nombre d'itérations des boucles for (i=k; i<=n ; i+=c)
  et for (i = 0; i<=n-k; i = i+c)

Le nombre d'itérations de la boucle for (i = k; i<=n; i +=c) est :

- **si k > n : 0**
- **si k = n : 1**
- **si k < n:**

  Soit it entier positif (it est un entier car it +1 est le nombre d'itérations) tel que
  $k + c \times it <= n$ et $k + c \times (it+1) > n$
  => $(n-k)/c - 1 < it <= (n-k)/c$ $(c>0)$
  => $(n-k)/c < it + 1 <= (n-k)/c + 1$
  - si $(n-k)/c$ est entier alors le nombre d'itérations est : $(n-k)/c + 1$
    sinon le nombre d'itérations est : $\lfloor (n-k)/c + 1 \rfloor$.
  - On remarquera que cette formule est également valable pour k=n.

- **En conclusion, le nombre d'itérations de la boucle for (i = k; i<=n; i +=c) est :**
  si k>n : 0 sinon : $\lfloor (n-k)/c + 1 \rfloor$.

  Montrons que dans les deux cas le nombre d'itérations est identique.
  On applique la formule ainsi obtenue à la seconde boucle en remplaçant n par n-k et k par 0, on obtient ainsi quand k<=n : $\lfloor (n-k-0)/c + 1 \rfloor$ et 0 sinon. Le nombre d'itérations **nombreIT** de ces boucles est donc identique et est :
  nombreIT = si k>n : 0 sinon : $\lfloor (n-k)/c + 1 \rfloor$

- **Transformation de l'incrément**

  Transformation de for (i=0; i<=n-k; i +=c) en for (i=0; i<=(n-k)/c; i++)
  Montrons que dans les deux cas le nombre d'itérations est identique :
  $\lfloor (n-k)/c + 1 \rfloor$ itérations pour la première et $\lfloor ((n-k)/c)/1 + 1 \rfloor$ pour la seconde donc les deux boucles ont le même nombre d'itérations. Le nombre d'itérations **nombreIT** de ces boucles est **nombreIT = si si k>n : 0 sinon : $\lfloor (n-k)/c + 1 \rfloor$**

  Conclusion :
  On conserve le nombre d'itérations en passant d'une boucle de type : for (i=k; i<=n; i+=c) avec : `k>=0, n>=0, c>0` en boucle for (i = 0; i<=(n-k)/c; i++).
  Le nombre d'itérations **nombreIT** de ces boucles est :
  nombreIT = si k>n : 0 sinon : $\lfloor (n-k)/c + 1 \rfloor$.

- **Remplacement du < par <=.**

  La transformation de for (i = k; i<n; i +=c) en for (i=k; i<=n-ε ; i+=c) avec c>= **ε >0** , n >= **ε** (pour conserver une borne positive) conserve-t-elle le nombre d'itérations ?

Le nombre d'itérations de la boucle for (i = k; i<=n-ε; i +=c) est :

si k>n-ε : 0 sinon : **⌊(n-ε-k)/c +1⌋.**

Le nombre d'itérations de la boucle for (i=k; i<n ; i+=c) :

- **si k >n -ε : 0 (k>n car ε>0)**
- **si k = n -ε : 1 (k = n -ε =>** on a donc k-ε <n (car ε>0 : au moins une itération) et k+c =c+n-ε >=n(c>=ε>0) on a donc une et une seule itération).
- **si k<n-ε** :
    1. i<n-**ε** => i<n (car ε>0) (on aura au moins le bon nombre d'itérations).
    2. i>=n-**ε =>** i+c>=n>n (car c >=ε **> 0)**) (on aura au plus le bon nombre d'itérations).

Conclusion :

La transformation de la boucle for (i=k; i<n; i+=c) en for (i=k; i<=n-ε; i+=c) **conserve le nombre d'itérations**. Le nombre d'itérations **nombreIT** de ces boucles est :

nombreIT = si k>n-ε : 0 sinon : **⌊(n-ε-k)/c +1⌋**

3.2 Transformation d'incrément – valeur en incrément + valeur

Montrons que la boucle `for (i=sup ; i>= inf; i-=dec)` a le même nombre d'itérations que la boucle `for(i=inf; i<=sup; i+=dec);`

D'après la démonstration précédente, le nombre d'itérations de la seconde boucle est :

si inf>sup : 0 sinon : **⌊(sup-inf)/dec+1⌋.**

Pour la boucle `for (i = sup ; i >= inf; i-=dec)`

- **si inf>sup : 0** (car sup>=inf est faux)
- **si inf= sup : 1**
- **si inf< sup:**

    Soit it un entier positif tel que sup - dec×it >= inf et sup - dec ×(it +1) < inf
    => (sup-inf)/dec -1 <= it < (sup-inf)/dec (cf. hypothèses)
    Le nombre d'itérations est alors de it +1
    (sup-inf)/dec -1 <= it < (sup-inf)/dec
    => (sup-inf)/dec < = it +1 < (sup-inf)/dec +1
    le nombre d'itérations est donc
    ⌊(sup-inf)/dec +1⌋.
    On remarquera que cette formule est également valable pour **inf= sup**.

Conclusion : ces deux boucles possèdent le même nombre d'itérations **nombreIT**.
Le nombre d'itérations de ces boucles est :

nombreIT = si inf>sup : 0 sinon : ⌊(sup-inf)/dec+1⌋

3.3 Les boucles while assimilables aux boucles for précédentes

Par définition, une boucle while de type :

```
i=inf;
while (i <= sup)
{
        instructions ne modifiant ni i, ni inf, ni inc, ni sup;
        i+=inc;
```

```
}
```

avec sup, inf et inc constants pour la boucle est équivalente en nombre d'itérations à la boucle for (i=inf; i<=sup;i+=inc).

Une boucle while de type :

```
i=sup;
while (i >= inf)
{
        instructions ne modifiant ni i, ni inf, ni dec, ni sup;
        i-=dec;
}
```

avec sup, inf et dec constants pour la boucle est équivalente en nombre d'itérations à la boucle for (i=sup; i >=inf; i-=dec) laquelle est équivalente à for(i=inf; i<=sup; i+=dec) (voir section précédente).

3.4 Les boucles do while assimilables aux boucles précédentes

Une boucle do ...while de type :

```
i=inf;
do
{
        instructions ne modifiant ni i, ni inf, ni inc, ni sup;
        i+=inc;
} while(i <= sup);
```

Cette boucle est équivalente à l'écriture suivante :

```
i=inf;
instructions ne modifiant ni i, ni inf, ni inc, ni sup;
i+=inc;
while(i<=sup)                                                    -- B1
{
        instructions ne modifiant ni i, ni inf, ni inc, ni sup;
        i+=inc;
}
```

Soit à :

```
i=inf;
instructions ne modifiant ni i, ni inf, ni inc, ni sup;
for (i+=inc;i<=sup; i+=inc)                                      -- B1
{    instructions ne modifiant ni i, ni inf, ni inc, ni sup;     }
```

Son nombre d'itérations est donc : 1 + le nombre d'itérations de la boucle B1.

Le nombre d'itérations **nombreIT** de ces boucles est donc :

nombreIT = si inf+inc >sup : 1 sinon : ⌊(⌊sup⌋-inc-inf)/inc +1⌋ +1.

Une boucle do ...while de type :

```
i=sup;
do
{
        instructions ne modifiant  ni i, ni inf, ni inc, ni sup;
        i-=inc;
}while(i >= inf);
```

est équivalente à l'écriture suivante :

```
i=sup;
instructions ne modifiant ni i, ni inf, ni inc, ni sup;
i-=inc;
while(i>=inf)                                                    -- B1
```

```
{
        instructions ne modifiant  ni i, ni inf, ni inc, ni sup;
        i-=inc;
}
```

Soit à :

```
i=sup;
instructions ne modifiant ni i ni inf ni inc ni sup;
for (i-=inc;i>=inf; i-=inc)                                          -- B1
{   instructions ne modifiant ni i ni inf ni inc ni sup        }
```

Son nombre d'itérations est donc : 1 + le nombre d'itérations de la boucle B1.

Le nombre d'itérations **nombreIT** de ces boucles est donc :

nombreIT = si inf-inc >sup : 1 sinon : ⌊(sup-inc -inf)/inc+1⌋+1.

3.5 Bilan sur les boucles arithmétiques étudiées

Pour l'ensemble des boucles précédentes, nous nous sommes ramenés à des boucles `for` de type `for(i=0;i<=n;i++)`. Ces boucles correspondent à une variable de boucle suivant une suite arithmétique de raison 1 et de valeur initiale 0.

3.6 Les boucles géométriques

Etudions maintenant la transformation d'une boucle géométrique de type `for (i=inf; i<=sup-ε; i*=cte)` avec inf>0, **sup>ε** >0 et cte un entier strictement positif.

Soit la boucle `for (i=1; i<=sup-ε; i*=cte)` avec sup strictement positif et cte un entier strictement positif.
A l'itération k, i=cte^(k-1)

Rappel sur les logarithmes : $cte^{(k-1)} = x \Rightarrow \log_{cte}(x)=k-1$
$i<=sup-\varepsilon \Rightarrow cte^{(k-1)} < sup-\varepsilon \Rightarrow k-1<\log_{cte}(sup-\varepsilon)$ (avec cte un entier positif)
Cette remarque permet de supposer que la boucle for (i=1; i<=sup-ε; i*=cte) est équivalente en nombre d'itérations à la boucle `for (k=0;k<=log`$_{cte}$`(sup-ε) ;k++)`.
Essayons de le montrer.

3.7 Transformation d'une boucle géométrique croissante en boucle arithmétique

Nous allons tout d'abord considérer les boucles commençant à 1 puis nous intéresser à celles commençant à une borne inf supérieure ou égale à 1.

3.7.1 Pour une boucle géométrique commençant à 1

Montrons que `for (i=1; i<=sup-ε; i*=cte)` est équivalente en nombre d'itérations à `for (k=0;k<=log`$_{cte}$`(sup-ε) ;k++)` (avec **sup>ε**>0 et cte entier >1 sinon boucle infinie).

Nombre d'itérations de la boucle `for (i=1; i<= sup-ε; i*=cte)` :

- **si sup-ε < 1 : 0**
- **si sup-ε = 1 : 1**
- **si sup-ε > 1 :**
  soit it tel que $cte^{it} <= sup-\varepsilon$ et $cte^{(it+1)} > sup-\varepsilon$
  On a donc $cte^{it} < =sup-\varepsilon < cte^{(it+1)}$.
  Soit it $<=\log_{cte}(sup-\varepsilon) < it+1$ le nombre d'itérations de la boucle est it +1 . Ce nombre

d'itérations est $\lfloor \log_{cte}(\mathbf{sup-\varepsilon}) +1 \rfloor$.

Nombre d'itérations de la boucle for (k=0;k<=□($\log_{cte}$(**sup-ε**)□;k++) :

- **si sup -ε < 1 : 0**
- **si sup-ε = 1 : 1**
- **si sup-ε > 1 :** le nombre d'itérations de la boucle est
  $\lfloor (\log_{cte}(\mathbf{sup-\varepsilon}) -0)/1 +1 \rfloor = \lfloor \log_{cte}(\mathbf{sup-\varepsilon})+1 \rfloor$.

Conclusion :
Le nombre d'itérations de ces boucles est : si **sup-ε < 1 : 0 sinon :** $\lfloor \log_{cte}(\mathbf{sup-\varepsilon}) +1 \rfloor$

3.7.2 Pour une boucle géométrique commençant à inf >=1

Etude de la transformation de suite géométrique de type `for (i=inf; i<= sup-ε; i*=cte)` avec inf >0 , sup >0 et cte un entier positif en `for (i=1; i<= sup-ε/inf; i*=cte)`.
Nombre d'itérations de la boucle `for (i=inf; i<= sup-ε; i*=cte)` :

- **si sup-ε < inf : 0**
- **si sup-ε = inf : 1**
- **si sup-ε > inf :**
  soit it tel que $inf \times cte^{it} <$ **sup-ε** et $inf \times cte^{(it+1)} >=$ **sup-ε**
  On a donc $inf \times cte^{it} <=$ **sup-ε** $< inf \times cte^{(it+1)}$.
  On a donc $cte^{it} <=$ (**sup-ε**)/inf $< cte^{(it+1)}$ car inf >0.
  Soit it $<= \log_{cte}(sup/inf) < (it+1)$ le nombre d'itérations de la boucle est it+1. Ce nombre d'itérations correspond à : si **sup-ε <= inf : 0 sinon :** $\lfloor \log_{cte}((\mathbf{sup-\varepsilon})/inf) +1 \rfloor$

Nombre d'itérations de la boucle `for (i=1; i<=(sup-ε)/inf; i*=cte)` :

- si **sup-ε < 1 : 0 sinon :** $\lfloor \log_{cte}((\mathbf{sup-\varepsilon})/inf) +1 \rfloor$

  Conclusion : les deux boucles ont le même nombre d'itérations et ce nombre est : **si sup-ε < inf : 0 sinon :** $\lfloor \log_{cte}((\mathbf{sup-\varepsilon})/inf) +1 \rfloor$

Conclusion : Une boucle de type `for (i=inf; i<=sup-ε; i*=cte)` avec inf strictement positif, sup positif et cte un entier positif est donc équivalente en nombre d'itérations à une boucle `for (k=1;k<=`$\lfloor \log_{cte}$`(sup-ε/inf)`$\rfloor$`;k++)`.
Le nombre d'itérations de ces boucles est donc :
si sup < inf : 0 sinon : $\lfloor \log_{cte}(\mathbf{sup-\varepsilon}/inf) +1 \rfloor$

3.8 Transformation d'une boucle décroissante en boucle croissante

Montrons qu'une boucle de type `for(i=sup; i>=inf; i/=cte)` est équivalente en nombre d'itérations à la boucle `for(i=`$\lfloor$`inf`$\rfloor$`; i <=sup; i*=cte)` avec inf>=0, sup >0 et cte entier >0.

Nombre d'itérations de la boucle for(i=sup; i>=inf; i/=cte) :

- **si sup < inf : 0**
- **si sup = inf : 1**
- **si sup** > inf **:**
  soit it tel que $sup \times cte^{(-it)} >= inf$ et $sup \times cte^{(-it-1)} < inf$
  On a donc $sup \times cte^{(-it-1)} < inf <= sup \times cte^{(-it)}$.
  On a donc $cte^{(-it-1)} < inf/sup <= cte^{(-it)}$. (sup >0).
  On a donc $l/cte^{(it+1)} < inf/sup <= 1/cte^{(it)}$.
  On a donc $\log_{cte} l -(it + 1) < \log_{cte}(inf/sup) <= \log_{cte} l -it.$ ($\log_{cte} l =0$)
  On a donc $-(it + 1) < \log_{cte}(inf) - \log_{cte}(sup) <= -it.$

On a donc (it + 1) >= $\log_{cte}$(sup) - $\log_{cte}$(inf) > it.
Le nombre d'itérations est donc : $\lfloor \log_{cte}(sup/inf) + 1 \rfloor$.

Nombre d'itérations de la boucle `for(i=⌊inf⌋; i <=sup; i*=cte)` :

si sup < inf : 0 sinon : $\lfloor \log_{cte}(sup/inf) + 1 \rfloor$

Conclusion : Ces deux boucles ont le même nombre d'itérations. Ce nombre est défini par :
si **sup < inf : 0, sinon** $\lfloor \log_{cte}(sup/inf) + 1 \rfloor$

Quelque soit la constante, +/-k ou ×// k on peut se ramener à une constante égale à un et donc tant que l'on parcourt un intervalle, on peut se ramener pour les boucles for avec une condition simple à une suite arithmétique.

3.9 Les boucles while assimilables aux boucles for précédentes

Une boucle while de type :

```
i=inf;
while (i <= sup)
{...
      i*=inc;
}
```

avec sup et inf constants pour la boucle est équivalente en nombre d'itérations à la boucle for(i=inf; i<=sup;i*=inc).

Une boucle while de type :

```
i=sup;
while (i >= inf)
{...
      i/=inc;
}
```

avec sup et inf constants pour la boucle est équivalente en nombre d'itérations à la boucle `for(i=i=⌊inf⌋; i<=sup;i/=inc)`.

3.10 Les boucles do while assimilables une des boucles précédentes

Une boucle do ...while de type :

```
i=inf;
do
{    instructions ne modifiant ni i ni inf ni inc ni sup;
       i*=inc;
} while(i <= sup);
```

Cette boucle est équivalente l'écriture suivante :

```
i=inf;
instructions ne modifiant ni i ni inf ni inc ni sup;
i*=inc;
while(i<=sup)                                                    -- B1
{      instructions ne modifiant ni i ni inf ni inc ni sup;
       i*=inc;
}
```

Soit à :

```
i=inf;
instructions ne modifiant ni i ni inf ni inc ni sup;
for (i*=inc;i<=sup; i*=inc)                                      -- B1
{    instructions ne modifiant ni i ni inf ni inc ni sup;        }
```

Son nombre d'itérations est donc : 1 + le nombre d'itérations de la boucle B1.

Le nombre d'itérations de ces boucles est donc :

si sup ×inc< inf : 1 sinon: $\lfloor \log_{inc}(sup \times inc/inf) + 1 \rfloor$.

Une boucle do ...while de type :

```
i=sup;
do
{     instructions ne modifiant ni i ni inf ni inc ni sup
      i/=inc;
}while(i >= inf);
```

Cette boucle est équivalente l'écriture suivante :

```
i=sup;
instructions ne modifiant ni i ni inf ni inc ni sup;
i/=inc;
while(i>=inf)                                                    -- B1
{      instructions ne modifiant ni i ni inf ni inc ni sup
       i/=inc;
}
```

Soit à :

```
i=sup ;
instructions ne modifiant ni i ni inf ni inc ni sup;
for (i/=inc;i>=inf; i/=inc)                                      -- B1
{       instructions ne modifiant ni i ni inf ni inc ni sup      }
```

Son nombre d'itérations est donc : 1 + le nombre d'itérations de la boucle B1.

Le nombre d'itérations **nombreIT** de ces boucles est donc :

si sup/inc < inf : 1, sinon $\lfloor \mathbf{log_{cte}(sup/inc \times inf) + 1} \rfloor$.

3.11 Boucle avec condition constante

Les boucles for et while s'exécutent jamais ou toujours en fonction de la valeur de la constante, les do ...while s'exécutent 1 fois ou toujours.

donc par exemple `for (...; cond; ...)` a un nombre d'itérations égal à `for(i=0; cond; i++)`

3.12 Boucle avec condition == et !=

- `for (i=k;i==N; i+=c)` (respectivement `for(i=k;i==N; i-=c)`)

  si c>0 si k=N : 1 sinon 0 cette boucle à le même nombre d'itérations que

  `for (...; (k==N?1:0); ...)` (condition constante)

  si c= 0 si k=N toujours sinon 0 à le même nombre d'itérations que for (...; k==N ; ...) (condition constante)

- for (i=k; i==N; i*=c) (respectivement for(i=k; i==N; i/=c))

  si c>1 cette boucle à le même nombre d'itérations que for (...; si(k==N?1:0); ...) (condition constante)

  si c = 1 si k=N toujours sinon 0 à le même nombre d'itérations que for (...; k==N; ...) (condition constante)

- for (i=k; i!=N; i+=c) à le même nombre d'itérations que for (i=k; i<N; i+=c) si c>0
- for (i=k; i!=N; i-=c) à le même nombre d'itérations que for (i=k; i>N; i-=c) si c>0

- for (i=k; i!=N; i*=c) à le même nombre d'itérations que for (i=k; k<N; i*=c) si c>1
- for (i=k; i!=N; i/=c) à le même nombre d'itérations que for (i=k; k<N; i/=c) si c>1

3.13 Boucle avec incréments multiples

Non traité pour le moment.

3.14 Boucle avec conditions multiples

for(i=k; i<N && i<T; i+=c) équivalent en nombre d'itérations à for (...; i< min(N, T); i+=c)

for (i=k; i>N && i>T; i-=c) équivalent en nombre d'itérations à for (...; i> min(N, T); i-=c)

for (i=k; i<N && i>T; i+=c) équivalent en nombre d'itérations à for (i=max(k,T); i< N); i+=c)

... à développer

3.15 Prise en compte des bornes des tableaux afin de permettre le calcul du nombre d'itérations

Remarque : En ajoutant des tests sur les bornes valides d'accès aux tableaux, on peut arriver à évaluer une borne pour les boucles qui prendra l'hypothèse que les index sont corrects. Lorsque dans le programme, on accède à des indices incorrects, le nombre d'itérations estimé et celui obtenu seront différents mais notre travail se situe dans un contexte temps réel strict et on fait l'hypothèse de programmes corrects.

3.16 Normalisation des boucles

Pour toutes les boucles, nous transformons la boucle initiale en une forme normalisée en utilisant les réécritures dépendant du type de boucle.

Lorsque **nombreIT** n'est pas défini pour une boucle, c'est une expression indéfinie.

Pour normaliser les boucles nous introduisons pour chaque boucle une variable varB propre à chaque boucle avec varB + 1 qui représente le numéro de l'itération courante. Si **nombreIT** est définie, varB∈ [0..**nombreIT -1**] sinon varB∈ [0..infinie[.

3.17 Boucle for

`for (exp1, exp2, exp3) {inst}` est transformé en

- Lorsque **nombreIT** est définie :

```
exp1;
for (varB=0; varB<=nombreIT -1; varB++) {inst; exp3;}
```

- Lorsque **nombreIT** est n'est pas définie

```
exp1;
for (varB=0; exp2; varB++) {inst; exp3;}
```

3.18 Boucle while

`while(exp2) {inst}` est transformé en :

- Lorsque **nombreIT** est définie :

(en particulier avec inst contenant une expression d'affectation (exp3) de la variable de boucle ou exp2 condition constante (le cas incrément multiple n'est pour le moment pas étudié),

```
for (varB=0; varB<=nombreIT -1; varB++) {inst}
```

- Lorsque **nombreIT** est n'est pas définie :

```
for (varB=0; exp2; varB++) {inst}
```

3.19 Boucle do while

`do {inst} while (exp2);` est transformé en :

- Lorsque **nombreIT** est définie (mêmes remarques que pour le cas précédent) :

```
for(varB=0; varB<=nombreIT -1; varB++) {inst};
```

- Lorsque **nombreIT** est n'est pas définie :

```
for (varB=0; exp2; varB++) {inst}
```

4 Conclusion

Nous nous situons ici dans un contexte où le nombre d'itérations de la boucle n'est pas interrompue par une instruction de rupture de séquence (exit...). Sous ces restrictions, nous avons donc montré que la transformation de la boucle initiale en la boucle for (varB=0; varB<= **nombreIT -1**; varB++) { inst} conserve le nombre d'itérations (si **nombreIT** est évalué).

Si **nombreIT** est indéfini, la condition initiale restant inchangée, le nombre d'itérations des deux boucles est aussi égal.

De plus, les instructions de la boucle initiale et leur ordre d'exécution est conservé, la variable varB n'interfère que sur le nombre d'itérations.

Les transformations conservent donc la sémantique du programme initial.

Dans l'ensemble notre méthode couvre plus de critères que les méthodes actuelles (combine contexte, nid de boucles ...). Il est possible d'apporter encore des améliorations.

On peut utiliser Calipso pour transformer les ruptures de séquence en introduisant des conditionnelles qui pour la plupart pourront être traitées ultérieurement. Cela améliorera l'expression des bornes. Les incréments multiples pourront également être traités s'ils sont monotones.

5 Références

[1] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. V. Engelen. Supporting timing analysis by automatic bounding of loop iterations. Real-Time Syst., 18(2-3):129–156, 2000.

[2] M. Kirner. Automatic loop bound analysis of
programs written in C. Master's thesis,
Technische Universität Wien, Institut für
Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[3] Bound-t tool homepage, http://www.tidorum.fi/bound-t/, 2005.

[4] ait tool, http://www.absint.com. 2007.

[5] Frontc homepage, http://www.irit.fr/frontc, 2007.

[6] H. Cassé, L. Féraud, C. Rochange, and P. Sainrat. Une approche pour reduire la complexité du flot de contrôle dans les programmes C. Technique et Science Informatiques, 21(7):1009–1032, 2002.