Gestion des `isets` en LSD^{12}

Xavier Devroey Alain Solheid Michaël Marcozzi

Année académique 2011-2012

Table des matières

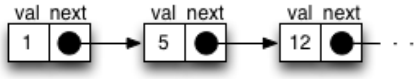
---

1 Introduction

Ce document a pour but de présenter brièvement un proposition de gestion des `isets` en LSD^{12}et plus particulièrement leur transposition en PCode (bien entendu vous êtes libres d'implémenter votre gestion de manière différente).

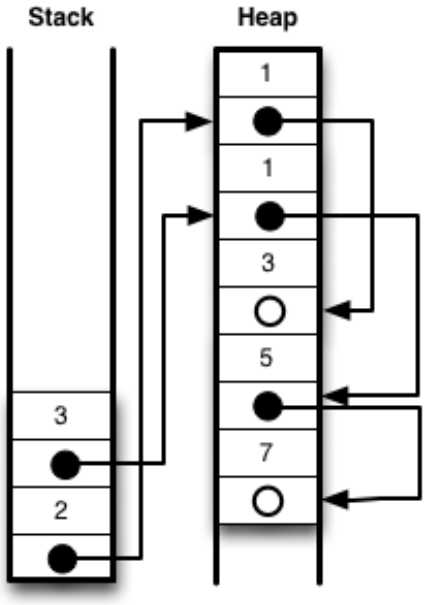
2 Représenter un `iset` en PCode

Le PCode est très basique, il possède les types booléen, entier et adresse. À l'instar du C, il est donc possible de représenter un ensemble sous forme de liste ordonnée chainée :



Contrairement au C, il n'est pas possible de tester la valeur d'une adresse, l'instruction `equ` ne fonctionnant que pour les types scalaires (integer et boolean). Ils est donc impossible de coder la fin de la liste au moyen d'une valeur spéciale (typiquement `NULL`) pour le champ `next` du dernier élément de la liste. En PCode il est donc nécessaire de mémoriser le nombre d'éléments présents dans une liste chainée afin de pouvoir la parcourir.

Stocker l'ensemble de la liste (potentiellement grande) sur le stack (qui correspond à la partie basse du STORE) de la GPMachine n'est pas possible. Il est donc nécessaire de réserver de l'espace mémoire sur le heap (qui correspond à la partie haute du STORE). Par exemple, pour les `isets` `x={1,3}` et `y={1,5,7}` :



Les cercles vides représentant l'adresse 0 du stack.

Un `iset` en PCode est donc codé sur deux cellules mémoire i et $i+1$ [1]. La cellule i est de type adresse et contient l'adresse du premier élément de l'ensemble (si il y en a un). La cellule $i+1$ est de type integer et contient le nombre d'éléments présents dans l'ensemble. À la création d'un ensemble, cette valeur vaut 0.

2.1 Initialisation d'un `iset`

La spécification LSD^{12}précise qu'un `iset` est considéré comme initialisé à l'ensemble vide dès le moment qu'il a été déclaré dans une fonction. Cela se traduit en PCode par une initialisation des cellules i et $i+1$ pour chacun des ensemble déclarés dans une fonction au début de l'exécution de cette fonction. Le PCode d'une fonction (et du programme principal) aura donc la forme :

```
ssp s
; Initilisation des isets
ujp @codeFct
; Declaration des sous-fonctions
define @codeFct
; Corps de la fonction
retf
```

3 Opérations sur les `isets`

3.1 Test de la présence d'un élément

Tester si un élément est présent dans un `iset` revient simplement à parcourir la liste chainée représentant cet `iset`. Si cela devait être fait en C, la fonction suivante pourrait être utilisée. Remarquez bien que comme la comparaison de deux adresse n'est pas possible en PCode, elle n'a pas non plus été utilisée en C.

1. Attention à bien prendre cela en compte lors du calcul du nombre de cellules mémoires nécessaires pour stocker les variables d'une fonction !

```
boolean contains(Set* set, int val){
    boolean contains = false;
    SetElement* current = set->elements;
    int cpt = 0;
    while(! contains && cpt < set->size){
        cpt++;
        contains = current->value == val;
        current = current->next;
    }
    return contains;
}
```

Afin de traduire cet algorithme en PCode, il est nécessaire de réserver un espace mémoire pour les variables qu'il utilise. Cet espace mémoire est arbitrairement situé entre les adresses 0 et 4 du STORE, ce qui signifie que lors du lancement du programme principale, il est nécessaire de décaler les adresses de +5 afin de ne pas utiliser cet espace. La traduction d'un programme LSD^{12}admettant l'utilisation d'`isets` en PCode donnera donc :

```
ssp s;
ujp @begin;
; Sous-fonctions
define @begin;
; Corps du programme
stp
```

Où `s` sera égal au nombre de cellules occupées par les variables du programme principal + 5 (pour les variables de travail de l'algorithme).

L'espace STORE[0-4] est utilisé de la manière suivante :

- STORE[0] = set : Set*
- STORE[1] = val : int
- STORE[2] = current : SetElement*
- STORE[3] = contains : boolean
- STORE[4] = cpt : int

La traduction de l'algorithme C en PCode donnera :

```
;       boolean contains = false;
ldc a 3
ldc b 0
sto b
;       SetElement* current = set->elements;
ldc a 2
ldc a 0
ind a
ind a
sto a
;       int cpt = 0;
ldc a 4
ldc i 0
sto i
```

```
;       while(! contains && cpt < set->size){
define @containsWhile
ldc a 3
ind b
not b
fjp @containsEndWhile
ldc a 4
ind i
ldc a 0
ind a
ldc a 1
add a
ind i
les i
fjp @containsEndWhile
;               cpt++;
ldc a 4
ldc a 4
ind i
ldc i 1
add i
sto i
;               contains = current->value == val;
ldc a 3
ldc a 2
ind a
ldc a 1
add a
ind i
ldc a 1
ind i
equ i
sto b
;               current = current->next;
ldc a 2
ldc a 2
ind a
ind a
sto a
ujp @containsWhile
;       }
define @containsEndWhile
;       return contains;
ldc a 3
ind b
```

Avant d'appeler cet algorithme il est nécessaire de lui passer l'adresse de l'ensemble à traiter, ainsi que la valeur à rechercher. Ceci se fait en initialisant les espaces mémoire STORE[0] et STORE[1]

avec respectivement, l'adresse de l'ensemble à traiter et la valeur à rechercher dans ce ensemble. Une fois que cet algorithme a été appelé, la valeur `contains` se retrouve au sommet du STORE.

3.2 Ajout d'un élément

Comme les `isets` sont représentés par des listes chainées ordonnées, ajouter une valeur à un `iset` revient à insérer une valeur dans la liste. En C, cela se ferait de la manière suivante (toujours sans utiliser la comparaison d'adresses) :

```
void addToSet(Set* set, int val){
        if(contains(set, val)){
                return;
        }
        if(set->size == 0){
                set->elements = malloc(sizeof(SetElement));
                set->elements->value = val;
                set->elements->next = 0;
                set->size = 1;
        } else {
                SetElement* previous =  set;
                SetElement* next = set->elements;
                int cpt = 1;
                while(cpt <= set->size && next->value < val){
                        cpt++;
                        previous = next;
                        next = next->next;
                }
                previous->next = malloc(sizeof(SetElement));
                previous->next->value = val;
                previous->next->next = next;
                set->size = set->size + 1;
        }
}
```

À nouveau, il est nécessaire de réserver les adresses 0 à 4 du STORE afin de stocker les variables locales de l'algorithme. Cet espace est utilisé de la manière suivante :

- STORE[0] = set : Set*
- STORE[1] = val : int
- STORE[2] = previous : SetElement*
- STORE[3] = next : SetElement*
- STORE[4] = cpt : int

La traduction de l'algorithme C en PCode donnera :

```
;       if(contains(set, val)){
;
Utilisation de l'algorithme contains
;
;               return;
not b
fjp @addToSetEndAddToSet
```

```
;       }
;       if(set->size == 0){
ldc a 0
ind a
ldc a 1
add a
ind i
ldc i 0
equ i
fjp @addToSetElseSizeDiffZero
;               set->elements = malloc(sizeof(SetElement));
ldc a 0
ind a
ldc i 2
new
;               set->elements->value = val;
ldc a 0
ind a
ind a
ldc a 1
add a
ldc a 1
ind i
sto i
;               set->elements->next = 0;
ldc a 0
ind a
ind a
ldc a 0
sto a
;               set->size = 1;
ldc a 0
ind a
ldc a 1
add a
ldc i 1
sto i
;       }
ujp @addToSetEndIfSizeZero
;   else {
define @addToSetElseSizeDiffZero

;               SetElement* previous =  set;
ldc a 2
ldc a 0
ind a
sto a
;               SetElement* next = set->elements;
```

```
ldc a 3
ldc a 0
ind a
ind a
sto a
;                int cpt = 0;
ldc a 4
ldc i 0
sto i
;                while(cpt <= set->size && next->value < val){
define @addToSetWhileFindWhereToInsert
ldc a 4
ind i
ldc a 0
ind a
ldc a 1
add a
ind i
leq i
fjp @addToSetEndWhileFindWhereToInsert
ldc a 3
ind a
ldc a 1
add a
ind i
ldc a 1
ind i
les i
fjp @addToSetEndWhileFindWhereToInsert
;                            cpt++;
ldc a 4
ldc a 4
ind i
ldc i 1
add i
sto i
;                            previous = next;
ldc a 2
ldc a 3
ind a
sto a
;                            next = next->next;
ldc a 3
ldc a 3
ind a
ind a
sto a
ujp @addToSetWhileFindWhereToInsert
```

```
;                   }
define @addToSetEndWhileFindWhereToInsert
;                   previous->next = malloc(sizeof(SetElement));
ldc a 2
ind a
ldc i 2
new
;                   previous->next->value = val;
ldc a 2
ind a
ind a
ldc a 1
add a
ldc a 1
ind i
sto i
;                   previous->next->next = next;
ldc a 2
ind a
ind a
ldc a 3
ind a
sto a
;                   set->size = set->size + 1;
ldc a 0
ind a
ldc a 1
add a
ldc a 0
ind a
ldc a 1
add a
ind i
ldc i 1
add i
sto i
;           }
define @addToSetEndIfSizeZero
; }
define @addToSetEndAddToSet
```

Avant d'appeler cet algorithme il est nécessaire de lui passer l'adresse de l'ensemble à traiter, ainsi que la valeur à ajouter. Ceci se fait en initialisant les espaces mémoire STORE[0] et STORE[1] avec respectivement, l'adresse de l'ensemble à traiter et la valeur à ajouter à l'ensemble.

3.3 Retrait d'un élément

Il n'y a pas d'instruction `free` en PCode, ce qui signifie que lorsqu'un élément est retiré d'un ensemble les cellules mémoires occupées par cet élément ne sont pas désalouées, elles sont simplement ignorées et seront libérées à la fin du programme. Bien entendu ce genre de comportement ne fait pas partie des bonnes pratiques de codage. Le langage C dispose bien quant à lui d'une instruction `free`...

Comme les `iset`s sont représentés par des listes chainées ordonnées, retirer une valeur à un `iset` revient à bypasser un élément dans la chaine en le reliant directement avec l'élément suivante. En C, cela se ferait de la manière suivante (toujours sans utiliser la comparaison d'adresses) :

```
void removeFromSet(Set* set, int val){
        if(set->size ==0){
                return;
        } else {
                SetElement* next = set->elements;
                SetElement* previous = (SetElement*) set;
                int cpt = 1;
                while(cpt <= set->size && val != next->value){
                        cpt++;
                        previous = next;
                        next = next->next;
                }
                if( cpt <= set->size && val == next->value ){
                        previous->next = next->next;
                        set->size = set->size - 1;
                }
        }
}
```

Les adresses 0 à 4 du STORE seront utilisées de la manière suivante :

- STORE[0] = set : Set*
- STORE[1] = val : int
- STORE[2] = previous : SetElement*
- STORE[3] = next : SetElement*
- STORE[4] = cpt : int

La traduction de l'algorithme C en PCode donnera :

```
;       if(set->size ==0){
ldc a 0
ind a
ldc a 1
add a
ind i
ldc i 0
equ i
fjp @removeFromSetElseNonEmptySet
;               return;
ujp @removeFromSetEndRemove
;       } else {
```

```
define @removeFromSetElseNonEmptySet
;                SetElement* previous = (SetElement*) set;
ldc a 2
ldc a 0
ind a
sto a
;                SetElement* next = set->elements;
ldc a 3
ldc a 0
ind a
ind a
sto a
;                int cpt = 1;
ldc a 4
ldc i 1
sto i
;                while(cpt <= set->size && val != next->value){
define @removeFromSetWhileNotVal
ldc a 4
ind i
ldc a 0
ind a
ldc a 1
add a
ind i
leq i
fjp @removeFromSetEndWhileNotVal
ldc a 1
ind i
ldc a 3
ind a
ldc a 1
add a
ind i
neq i
fjp @removeFromSetEndWhileNotVal
;                            cpt++;
ldc a 4
ldc a 4
ind i
ldc i 1
add i
sto i
;                            previous = next;
ldc a 2
ldc a 3
ind a
sto a
```

```
;                           next = next->next;
ldc a 3
ldc a 3
ind a
ind a
sto a
ujp @removeFromSetWhileNotVal
;                   }
define @removeFromSetEndWhileNotVal
;                   if( cpt <= set->size && val == next->value ){
ldc a 4
ind i
ldc a 0
ind a
ldc a 1
add a
ind i
leq i
fjp @removeFromSetEndIfValEqu
ldc a 1
ind i
ldc a 3
ind a
ldc a 1
add a
ind i
equ i
fjp @removeFromSetEndIfValEqu
;                           previous->next = next->next;
ldc a 2
ind a
ldc a 3
ind a
ind a
sto a
;                           set->size = set->size - 1;
ldc a 0
ind a
ldc a 1
add a
ldc a 0
ind a
ldc a 1
add a
ind i
ldc i 1
sub i
sto i
```

```
;                   }
define @removeFromSetEndIfValEqu
;           }
; }
define @removeFromSetEndRemove
```

Avant d'appeler cet algorithme il est nécessaire de lui passer l'adresse de l'ensemble à traiter, ainsi que la valeur à retirer. Ceci se fait en initialisant les espaces mémoire STORE[0] et STORE[1] avec respectivement, l'adresse de l'ensemble à traiter et la valeur à retirer à l'ensemble.

3.4 Maximum et minimum

Les `isets` étant représentés par des listes chainées ordonnées, le minimum correspond au premier élément de la liste (si il y en a un) et le maximum correspond au dernier élément de la liste (si il y en a un). Trouver le maximum revient donc à parcourir la liste pour trouver le dernier élément. En C, cela donnerait :

```
int max(Set* set){
        SetElement* current = set->elements;
        int cpt = 1;
        while(cpt < set->size){
                cpt++;
                current = current->next;
        }
        return current->value;
}
```

Les adresses 0, 2 et 4 du STORE seront utilisées de la manière suivante :
- STORE[0] = set : Set*
- STORE[2] = current : SetElement*
- STORE[4] = cpt : int

La traduction de l'algorithme C en PCode donnera :

```
;       SetElement* current = set->elements;
ldc a 2
ldc a 0
ind a
ind a
sto a
;       int cpt = 1;
ldc a 4
ldc i 1
sto i
;       while(cpt < set->size){
define @maxInSetWhileNotEnd
ldc a 4
ind i
ldc a 0
ind a
ldc a 1
```

```
add a
ind i
les i
fjp @maxInSetEndWhileNotEnd
;                   cpt++;
ldc a 4
ldc a 4
ind i
ldc i 1
add i
sto i
;                   current = current->next;
ldc a 2
ldc a 2
ind a
ind a
sto a
ujp @maxInSetWhileNotEnd
;         }
define @maxInSetEndWhileNotEnd
;         return current->value;
ldc a 2
ind a
ldc a 1
add a
ind i
```

Avant d'appeler cet algorithme il est nécessaire de lui passer l'adresse de l'ensemble à traiter en initialisant l'espace mémoire STORE[0]. Après exécution de cet algorithme, le maximum est placé au sommet du STORE.

La recherche du minimum dans un `iset`, de même que la recherche du nombre d'éléments dans un `iset` sont laissés au bon soin du lecteur.