Kink developpeur

Valvassori Moïse

6 octobre 2000

Résumé

Ce document contient la documentation programmeur pour `Kink`

Table des matières

1 Structures de données

1.1 Dynamic Array

Cette structure implémente des tableaux dynamiques.

Les tableaux dynamiques permettent de gérer des collections d'objets dont le nombres peut varier au cours de temps. J'ai créé cette structure en pensant au coups réel de l'implémentation d'une liste chaînée. En effet, si on a une liste chaînée classique d'entiers, on pense souvent que chaque maillon ne coûte que 8 octets en mémoires mais on oublie que le système utilise de la mémoire à chaque `malloc` (entre 20 et 50 octets). Pour éviter cet inconvénient, on peut réserver un tableau plus grand qui contient un nombre donné d'éléments et sous allouer les éléments à l'intérieur de ce tableau. Si celui ci devient trop petit, on en realloue un autre que l'on chaîne au premier.

1.1.1 Structure

```
#define Util_signature(x) int (*x)(void * cell , DAAction_t action)
typedef struct DArray
{
  struct DArray * next;
  struct DArray * previous;
  unsigned int nb_free_cell;
  unsigned int nb_of_cells;
  unsigned int size_of_cell;
  Util_signature(util);
} darray_t;
```

`next`, `previous`

Ces deux champs sont classiques dans une liste chaînée.

`nb_free_cell`

désigne le nombre de cellule libre à l'intérieurs de ce bloc

`nb_of_cells`

nombre total de cellule dans ce bloc

`sire_of_cell`

Taille d'une cellule

`Util_signature`

Cette fonction permet de réaliser deux opérations au choix :

- dire si la cellule est vide. Pour réaliser cette action, il faut passer `is_null` en paramètre.
- effacer une cellule. Pour réaliser cette action, il faut passer `clear_clean` en paramètre.

La fonction que l'on passe en paramètre peut ressembler à cela :

```
int
util(void * cell , DAAction action)
{
 int ret=0;
 switch (action)
   {
```

```
      case is_null:
        if (*((int*)cell) == 0)
    {
      ret=1;
    }
        else
    {
      ret=0;
    }
        break;
      case clear_cell:
        *(int*)cell = 0;
        break;
      };
     return (ret);
    }
```

ici, on traite un `int`.

1.1.2 interface publique

`darray_t * da_make_dynamic_array (uint size_of_cell,- Util_signature (util)`

Fabrique un nouveau tableau dynamique. On précise la taille et la fonction d'utilitaire des éléments. On notera que le tableau contient que des cellules vide.

`DACell_t da_allocate_cell (darray_t * array)`

Alloue une nouvelle cellule. Cette fonction cherche la première cellule vide et la retourne.

`void * da_cell_ptr (darray_t *array,DACell_t index)`

Renvoie un pointeur sur un élément d'un tableau dynamique désigné par son index dans le tableau.

`void * da_free_cell(darray_t *array,DACell_t index)`

Libère une cellule dans un tableau dynamique

`void da_free_dynamic_array(darray_t * array)`

Libère un tableau dynamique

`DACell_t da_length(darray_t * array)`

Renvoie le nombre d'éléments contenu dans un tableau dynamique

`void * da_cell_ptr_and_next (darray_t **array , DACell_t *index)`

Cette fonction retourne l'élément pointé et modifie les paramètres qu'on lui a passé afin de le faire pointé sur l'élément suivant.

1.2 Cell

Cette structure représente les briques de bases de LISP. C'est dans ces cellules que sont stockées les programmes LISP.

Lors de l'allocation d'une cellule, on utilise l'alloueur du garbage collector (voir section 1.5 page 7). Les cellules ne sont jamais explicitement détruite. C'est le travail du garbage collector.

1.2.1 Structure

```
typedef struct _KCELL
{
  ktype_t type;
  union
  {
    char            boolean;
    long            integer;
    double          floats;
    char          * string;
    ksymbol_t       symbol;
    kpair_t         pair;
    kprimitive_t    primitive;
    klambda_t       lambda;
    struct _KCELL * quote;
  }data;
}kcell_t;
```

type

Le `type` permet de savoir à quel type d'objets on a à faire.

data

En fonction du `type`, l'application choisi le champ à prendre en compte. On y reconnaît tout les types du LISP

1.2.2 Interface publique

L'interface se compose pour l'essentiel de constructeur de cellules.

`kcell_t * cell_make_new()`

Construit une cellule vide. Elle ne fait que l'allouer. Elle n'y met rien dedans.

`kcell_t * cell_make_new_quote(kcell_t *q)`

`kcell_t * cell_make_new_boolean(char b)`

`kcell_t * cell_make_new_integer(long l)`

`kcell_t * cell_make_new_float(double d)`

`kcell_t * cell_make_new_string(char *s)`

`kcell_t * cell_make_new_symbol(char *s)`

`kcell_t * cell_make_new_primitive(kprimitive_t *p)`

`kcell_t * cell_make_new_lambda(klambda_t *l)`

`kcell_t * cell_make_new_pair0()`

`kcell_t * cell_make_new_pair(kcell_t * car)`

`kcell_t * cell_make_new_empty(void)`

Ces fonctions construisent des cellules pleines. C'est à dire qu'elles contiennent déjà une valeur. Cette valeur est passé en paramètre

1.3 Environnement

Cette structure représente les environnements LISP. Un environnement est une liste qui associe un symbole à un objet LISP. Habituellement, on utilise une table de hachage au lieu d'une liste pour des raisons de performance mais j'ai choisi une représentation en liste à cause de la simplicité de la méthode. Les environnements peuvent être imbriqué les un dans les autres ce qui forme une structure arborescente.

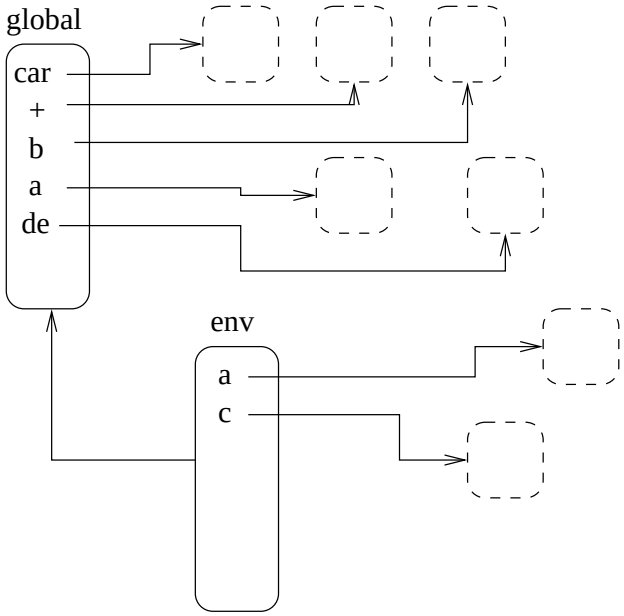

FIG. 1 – Une idée des environnements. On y voit l'environnement global qui contient les symboles *+,a,b,car,de*. Chaque symbole est associé à une cellule. Un environnements *env* est associe à l'environnement global.

1.3.1 Structure

```
typedef struct _ENV
{
  struct _ENV *parent;
  darray_t *table;
}
env_t;
```

Cette première structure représente l'environnement en temps qu'objet. C'est à dire une liste de lien qui peut être lié à une autre.

parent

C'est un pointeur vers l'environnement parent. L'environnement global n'a pas de parent donc dans ce cas, il vaut `NULL`.

table

C'est un pointeur vers le tableau dynamique (voir chapitre 1 page 2) qui contient les associations symboles - cellules. Le type des éléments du tableau est `env_cell_t`.

La structure `env_cell_t` représente un élément du tableau dynamique `table` de la structure précédente.

```
typedef struct
{
  char    * name;
  kcell_t * value;
  char     flag;
}env_cell_t;
```

name

contient le nom du symbole

value

contient la cellule associée au symbole

flag

ce flag sert à différencier les paramètres d'une fonction des autres objets. (voir section 2.3 page 10)

1.3.2 Interface publique

Il y a deux fonctions principales : un accèsseur à l'environnement et un modificateur.

`env_t * environment_make (env_t *parent)`

Fabrique un nouvel environnement. `parent` est l'environnement père auquel le nouvel environnent sera rattaché.

`int environment_update(env_t * e,char *name,kcell_t * val)`

`int environment_update_flag(env_t * e,char *name,kcell_t * val,char flag)`

Rajoute le couple (nom,valeur) l'environnement `e`.

`kcell_t ** environment_lookup(env_t *e,char *name,char * flag)`

Cherche le symbole `name` dans l'environnement `e`. On nous renvoie la cellule qui est associé ainsi que le `flag`.

1.4 Stack

Bien sur, nous avons aussi besoin d'une structure de pile. Celle utilisée par Kink est implémente à l'aide des tableaux dynamiques.

1.4.1 Structure

```
typedef struct
{
  darray_t * base;
  darray_t * current;
  DACell_t cell;
}stack_t;
```

base

pointe sur le premier tableau dynamique de la pile.

current

pointe sur le tableau dynamique courant (le dernier).

cell

l'élément courant dans le tableau dynamique.

1.4.2 Interface publique

`stack_t * stack_new()`

Fabrique une nouvelle pile vide.

`void stack_push(stack_t * s,kcell_t * obj)`

Place un nouvel objet dans la pile.

`kcell_t * stack_pop(stack_t *s)`

Retire le dernier élément de pile.

`int is_empty(stack_t *s)`

Renvoie 0 si la pile est vide.

1.5 Garbage collection

Je n'ai pas implémenté de garbage collector. J'ai utilisé une bibliothèque qui m'en offrait un. Elle est réalisé par

```
Copyright 1988, 1989 Hans-J. Boehm, Alan J. Demers
Copyright (c) 1991-1996 by Xerox Corporation.  All rights reserved.
Copyright (c) 1996-1999 by Silicon Graphics.  All rights reserved.
Copyright (c) 1999 by Hewlett-Packard Company. All rights reserved.
```

On peut trouver la dernière version à l'adresse web suivante :
http://www.hpl.hp.com/personal/Hans_Boehm/gc.

1.5.1 Interface publique

Pour décrire l'interface publique du garbage collector, je préfère recopier la documentation fourni.

```
 GC_malloc(nbytes)
 - allocate an object of size nbytes.  Unlike malloc, the object is
   cleared before being returned to the user.  Gc_malloc will
   invoke the garbage collector when it determines this to be appropriate.
   GC_malloc may return 0 if it is unable to acquire sufficient
   space from the operating system.  This is the most probable
   consequence of running out of space.  Other possible consequences
   are that a function call will fail due to lack of stack space,
   or that the collector will fail in other ways because it cannot
   maintain its internal data structures, or that a crucial system
   process will fail and take down the machine.  Most of these
   possibilities are independent of the malloc implementation.
```

```
GC_free(object)
- explicitly deallocate an object returned by GC_malloc or
  GC_malloc_atomic.  Not necessary, but can be used to minimize
  collections if performance is critical.  Probably a performance
  loss for very small objects (<= 8 bytes).
```

2 Fonctionnement

L'évaluateur est basé sur une boucle *read-eval-print*. Cette boucle est visible dans le fichier `kink.c`.

Voyons de plus prés à quoi ressemble cette boucle.

2.1 Parser

Le programme rentre dans le parser par la fonction `read_`. La saisie du texte est assurée par la fonction `readline`. Cette fonction fait partie de la bibliothèque GNU du même nom.

Ensuite, on initialise l'arbre syntaxique. C'est à dire que l'on met une expression vide dans l'arbre syntaxique, de façon à effacer l'ancienne expression contenue dans celui ci.

Le vrai travail du parser commence ici, dans la fonction `parse_line`. Cette fonction prend la ligne qu'on vient de lire en argument et nous rend un arbre syntaxique. La fonction parcours toute la chaîne caractères par caractères et agit suivant le caractère rencontré :

- si on rencontre un séparateur, on passe au caractère suivant.
- si on rencontre une guillemet, on extrait une chaîne jusqu'au prochain guillemet et on la place dans l'arbre syntaxique.
- si on rencontre une parenthèse ouvrante, on ouvre une nouvelle liste dans l'arbre syntaxique Si on tombe sur une parenthèse fermante, on ferme cette liste.
- Dans tout les autres cas, on extrait a à faire à un mot. Une fonction (`extract_token`) se charge de renvoyer le symbole ou le nombre compris dans la chaîne. Une fois extrait le symbole est inséré dans l'arbre syntaxique

2.2 Représentation des données

Nous sommes entre l'étape de parsing et d'évaluation. Il est temps d'entre voir comment sont représentés les objets LISP en mémoire.

2.2.1 Donnée simple

Les données simples sont codées au travers de la structures `cell_t`. Dans la pratique, on crée un nouvel élément en appelant son constructeur. Par exemple, si on veut représenter l'entier 1, on écrira :

```
kcell_t * toto =  cell_make_new_integer(1);
```

2.2.2 Liste

Les listes sont aussi gérées par la structure `cell_t`. Les cellules sont du type `Pair`. Les données de la liste sont le `car` et le `cdr`. Dans le `car`, on y place un pointeur vers une cellule qui

représente le premier élément de cette liste. Et dans le `cdr`, on y met un pointeur vers le reste de la liste. C'est à dire une cellule de type `Pair` qui contient le reste de la liste. La figure 2 montre le représentation interne d'une liste.

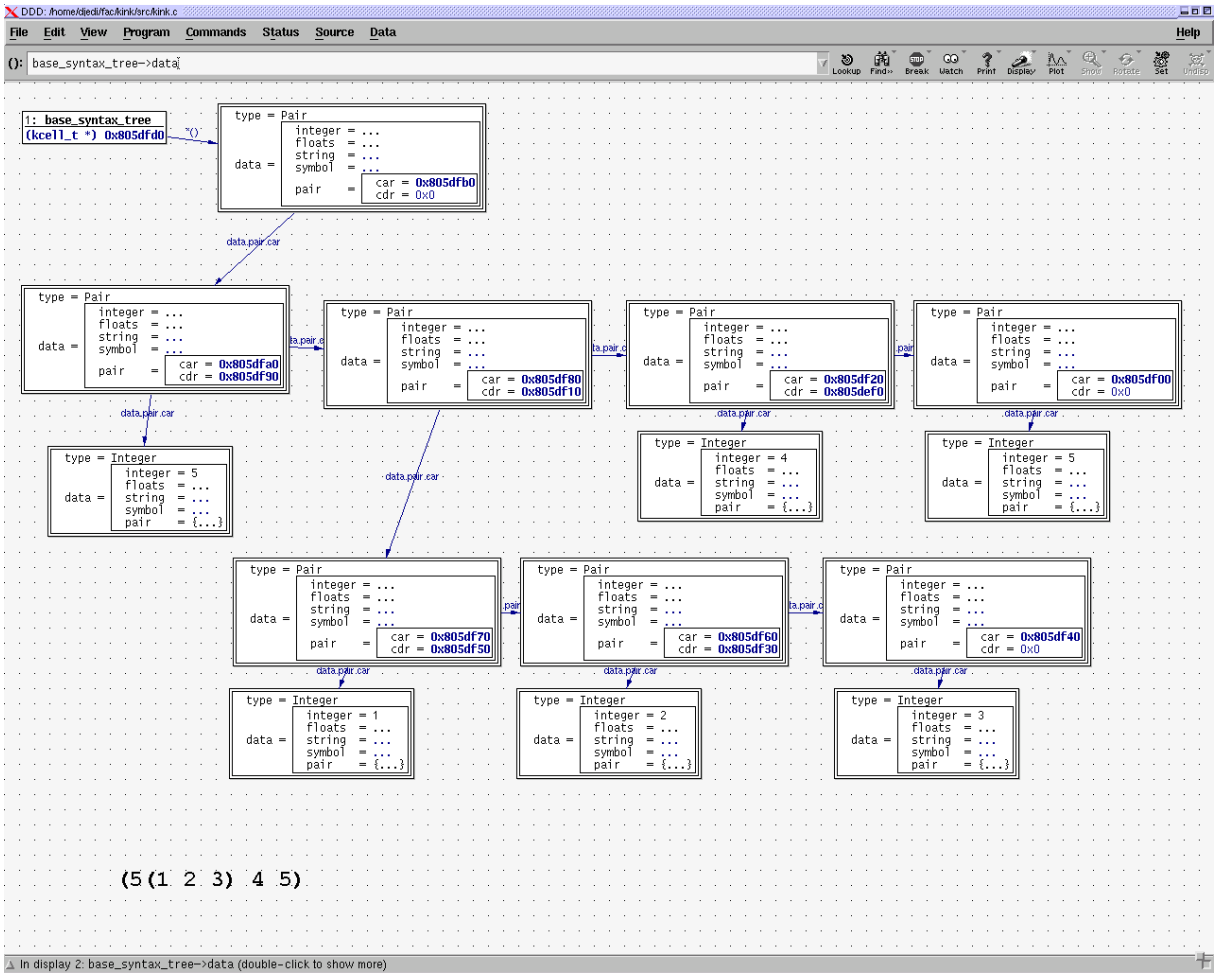

FIG. 2 – Exemple de liste. C'est une copie d'écran du débugger *DDD*. On y voit la représentation de la liste `(5 (1 2 3) 4 5)`. Cette liste est rattachée à l'arbre syntaxique.

2.2.3 Arbre syntaxique

Dans la partie sur le parser, on a vue que celui-ci stockait la ligne que l'on a tapée sous forme d'arbres syntaxiques. Nous allons voir ici comment sont représentées les données.

L'arbre syntaxique n'est qu'une liste (au sens LISP) qui représente l'expression qui vient d'être lu. Pour être plus précis, c'est une liste de liste. Cela permettra de mettre plusieurs expression sur une même ligne.

La construction de l'arbre est assuré par la fonction `make_syntax_tree`

Quand on commence une liste, on crée un nouvelle paire et on la colle sur le noeud courant. On prend la peine de mettre le noeud courant dans une pile pour le retrouver lors de la fermeture de la liste.

La fermeture d'une liste dans l'arbre syntaxique se fait en remontant la liste tout simplement.

Les cellules normales sont collées sur le premiers élément de la liste. Bien sur, si la liste n'existe pas on la crée.

La variable `base_syntax_tree` est un pointeur sur la racine de cet arbre.

2.3 Évaluation

L'arbre syntaxique construit lors du parsing est ensuite évalué dans la fonction `eval`. Cette fonction prend en argument un arbre syntaxique et un environnement. On renvoie l'arbre syntaxique modifié. Cette fonction prend aussi une continuation mais je ne les gère pas encore et elle est ignorée.

Nous allons voir dans ce qui suit comment sont évalués les différents type d'objets LISP.

2.3.1 Atome

La plupart des valeurs atomiques n'ont pas besoin d'être évalué et on renvoie donc le noeud de l'arbre syntaxique tel quel. En effet, une cellule qui contient l'entier 4 est déjà évalué et ne nécessite aucun traitement. Parmi ces atomes non évalué, on retrouve aussi les *primitives* et les *lambdas*.

Les symboles, eux, sont évalués. Il ne faut pas confondre le symbole (`x`) et sa notation quotée (`(' x)`). La notation quotée d'un symbole est évaluée comme tous les autres atomes, on revoit la cellule telle quelle.

Les symboles sont donc évalués dans la fonction `eval_symbol`. La première étape de l'évaluation du symbole consiste à prendre dans l'environnement la cellule associée au symbole grâce à l'environnement. Ensuite suivant la valeur du *flag* associé au nom, on fait :

- on évalue normalement la cellule si le *flag* vaut 1. On remplace la cellule par sa valeur calculée dans l'environnement puis on rend le résultat.
- si le *flag* vaut 2, on construit une *thunk* (fonction sans paramètre) qui renvoie la cellule contenue dans l'environnement. Par exemple, si dans l'environnement la valeur associée au symbole est `(' (1 2 3))` alors on renverra `(lambda () (' (1 2 3)))`. Ce flag est 2 lorsque le symbole est un paramètre d'une *lambda*. Cela permet de l'évaluer paresseusement.

2.3.2 liste

L'évaluation des listes se fait dans la fonction `eval_function`. La première étape consiste à retrouve la fonction. Pour cela, on évalue le premier terme de la liste. Souvent il s'agit de la fonction elle même mais ce n'est pas toujours le cas.

La suite des opérations dépend du type de d'objet que l'on a. On peut avoir une *primitive* ou une *lambda*.

primitive Suivant l'arrité de la primitive, on vérifié si les arguments sont bien définis. Puis on appelle la primitive. Dans la primitive, on évalue les arguments (si on s'en sert) puis on effectue l'opération définit dans la primitive.

Les primitives courantes sont définies dans le fichier `primitive.c`.

lambda La première étape de l'évaluation d'une lambda est la création de l'environnement d'exécution. Cet environnement étend l'environnement courant. Ensuite, on place les arguments dans ce nouvel environnement. Chaque argument formel est associé à la valeur que l'on lui a passé. Si on passe trop ou pas assez d'arguments, il y a une erreur. Si tout c'est bien passé, il nous reste plus qu'à évaluer le corps de la *lambda*.

2.4 Affichage

L'affichage des résultats se fait de façon très simple. Après avoir parser une ligne et l'avoir évaluer, on appelle la fonction `cell_print` dans le fichier `cell.c`. Cette fonction regarde à quel type de cellule on a affaire et affiche la valeur de la cellule. Si l'objet a affiché est une liste, on appelle la fonction d'affichage de liste récursivement.

3 Exemples

Dans cette partie, je vais donner une série d'exemples d'expressions Kink ainsi que le résultat produit par le débugger interne [1] . A chaque fois que l'on rentre dans la fonction `eval`, on affiche à l'écran un crochet ouvrant suivit de l'expression que l'on évalue. Lorsque l'on sort de la fonction `eval` on ferme le crochet et on affiche la valeur de retour.

Le premier exemple montre comment s'effectue la définition d'une variable :

```
Kink> (define a (+ 2 3))
} (define a (+ 2 3))
 } define
  } <Procedure [0x804af24]>
  { <Procedure [0x804af24]>
 { <Procedure [0x804af24]>
{ a
a
Kink> a
} a
 } (+ 2 3)
  } +
   } <Procedure [0x804a73c]>
   { <Procedure [0x804a73c]>
  { <Procedure [0x804a73c]>
  } 2
  { 2
  } 3
  { 3
 { 5
{ 5
5
Kink> a
} a
 } 5
 { 5
{ 5
5
```

[1] Pour activer le debugger interne, il faut configure la compilation comme il suit : `configure ---enable-debug`

La première chose que l'on remarque est que l'expression que l'on affecte au symbole n'est pas évaluée. La procédure que l'on voit apparaître à la ligne 4 est la primitive `define`. Elle provient de l'évaluation du symbole *define*. A la ligne 9, j'évalue `a`. L'expression associé à `a` est évaluée. A la ligne 23, je réévalue la symbole `a`. Cette fois ci, l'expression n'est pas évaluée car le résultat de la précédente évaluation a été conservé.

Dans l'exemple suivant, j'évalue une `lambda` à deux arguments et j'en utilise qu'un seul :

```
1   Kink> ((lambda (x y) (* y y)) 3 (+ 1 2))
    } ((lambda (x y) (* y y)) 3 (+ 1 2))
     } (lambda (x y) (* y y))
      } lambda
5      } <Procedure [0x804b5c4]>
       { <Procedure [0x804b5c4]>
      { <Procedure [0x804b5c4]>
     { <lambda (x y)>
     } (* y y)
      } *
10     } <Procedure [0x804a9f4]>
       { <Procedure [0x804a9f4]>
      { <Procedure [0x804a9f4]>
      } y
15     } (+ 1 2)
        } +
         } <Procedure [0x804a73c]>
         { <Procedure [0x804a73c]>
        { <Procedure [0x804a73c]>
20      } 1
        { 1
        } 2
        { 2
       { 3
25    { 3
      } y
       } 3
       { 3
      { 3
30   { 9
    { 9
    9
```

Donc on remarque que l'argument `x` n'est jamais évalué. De plus, le symbole `y` n'est évalué qu'une seule fois (ligne 14-25) ensuite on utilise la valeur calculée à la première évaluation (ligne 26).