Addenda au sujet de compilation

remplace la versoin du 28 mars 2018

1 Construction de l'arbre de syntaxe

L'analyseur construit pour chaque instruction un arbre de syntaxe qui permettra de produire l'analyse des types, l'interprétation et la compilation.

Le plus simple est de représenter cette syntaxe abstraite sous la forme d'un arbre binaire.

Une structure de donnée simple pour cet arbre pourrait commencer ainsi [1] :

```
record SyntTree {
  tag: (LITERAL, VARIABLE, PLUS, ..., AFFECTATION, ...);
  left, right: ^SyntTree;
  //... con't
}
```

2 Construction des types

L'analyseur construit une représentation abstraite du type de chaque expression.

À nouveau, une structure de donnée simple pour définir un type est un arbre binaire :

```
record Type {
  tag: (CHARACTER, INTEGER, BOOLEAN, POINTER, ARRAY,
   FUNCTION, TIMES, RANGE, ENUMERATE);
  left, right: ^Type;
}
```

Astuce

Il est possible d'enregistrer des informations concernant les types énumérés : un espace de nom (chaque énuméré est identifié par un nombre unique), des valeurs minimales et maximales des intervalles, etc.

1. Pour illustrer nos propos, nous notons des exemples dans un langage qui pourrait correspondre à *Léa*. L'idée est de fournir des éléments d'algorithmique, et non le code C demandé. L'exercice qui consiste à écrire un compilateur dans le langage de ce compilateur est d'ailleurs connu et permet de produire automatiquement des compilateurs testés. Cela s'appelle l'auto-amorçage et est hors sujet du présent document.

3 Analyse de la bonne formation des instructions

On doit enregistrer le type de chaque expression dans l'arbre de syntaxe pour ensuite vérifier sa bonne formation. Une affectation vérifie la compatibilité de type entre sa partie droite et sa partie gauche, un test vérifie que son argument est booléen, etc.

4 Interprétation

En plus d'ajouter à l'arbre de syntaxe le type, nous allons ajouter un ensemble d'éléments qui permettent l'interprétation de cet arbre :

> **Astuce**
> Il est intéressant d'ajouter des éléments permettant de visualiser les résultats au moment de la mise au point : le numéro de la ligne du code source, le dernier terminal lu par l'analyseur lexical, le nom de la variable, etc.

— `Place` Il s'agit pour une variable de l'emplacement dans la mémoire.
— `Value` La valeur d'une expression quelconque. Il faut pouvoir enregistrer la plus grande taille pour un type simple. Dans le cas de *Léa*, il s'agit de 4 octets pour les types pointeur et *integer*.
— `Memory` Il s'agit du type de mémoire pour une variable (statique, la pile ou le tas) (cf plus bas).

Un sommet de l'arbre de syntaxe revient donc à peu près à ceci maintenant :

```
record SyntTree {
  tag: (LITERAL, VARIABLE, PLUS, ..., AFFECTATION, ...);
  left, right: ^SyntTree;
  place: integer;
  value: integer; // 4 bytes
  memory: (STATIC, STACK, HEAP);
}
```

L'interprétation revient donc à parcourir l'arbre de syntaxe et à calculer pour chaque nœud, sa valeur. Ce qui donne de façon incomplète et simplifiée ceci :

```
function interpretation (node : ^SyntTree) : integer
{
switch (node^.tag){
  case VARIABLE:
    switch (node^.tag){
      case STATIC:
        node^.value = staticMemory [ node^.place ];
        break;
      case STACK:
        node^.value = stackMemory [ node^.place ];
        break;
      case HEAP:
        node^.value = heapMemory [ node^.place ];
        break;
```

```
      }
    }
    break;
  case _PLUS_:
    node^.value = interpretation (node->^.lhs) +
   interpretation(node^.rhs);
    break;
    // ... con't
  }
  return node^.value;
}
```

5 Gestion de la mémoire au moment de l'interprétation

L'interprétation d'une `variable_access` exploite trois données :

1. Son `type` (qui permet de connaître sa taille et aussi le traitement qui lui est réservé)
2. Sa `place` qui permet de trouver son contenu
3. L'espace mémoire dans laquelle elle est stockée (en mémoire statique, dans la pile ou dans le tas)

Comment sont enregistrées ces informations ?

1. Chaque déclaration `IDENTIFIER ':' type` permet d'associer au nom de la variable un type. Il s'agit donc de construire des environnements et d'y enregistrer cette information. Quand l'analyseur syntaxique construit l'arbre d'analyse, il y associe le type connu de chaque variable.
2. Si les variables sont organisées dans la mémoire dans l'ordre où elles apparaissent, il est simple d'utiliser un compteur (*offset*) incrémenté à chaque fois de la taille du dernier type.
3. Au moment de l'analyse syntaxique, chaque déclaration `IDENTIFIER ':' type` se trouve à l'un des deux endroits possibles : a) dans la partie de déclaration des variables globales, b) à l'intérieur de la définition ou de la déclaration d'une fonction. Dans le premier cas, la variable est statique, dans le second cas, la variable est stockée dans la pile. Seules les `variable_access` de type `variable_access '^'` ont leur contenu dans le tas.

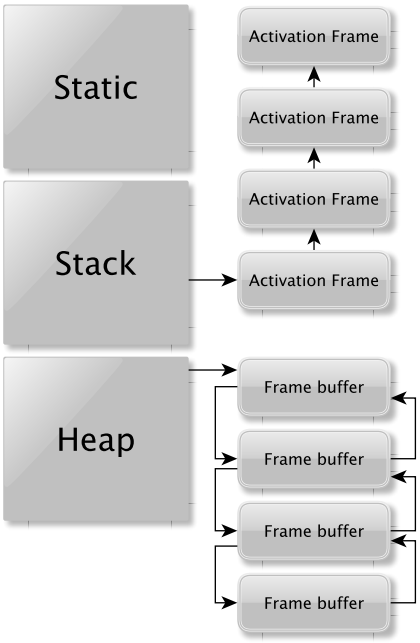

FIGURE 1 – Espace mémoire

5.1 Espace statique, pile et tas

Astuce

En langage C, une bonne facon d'avoir un tableau d'octets est d'utiliser `u_int8` défini dans `<stdint.h>`

La mémoire de la machine peut se représenter par le schéma de la figure 1

Trois espaces sont utilisés : La partie statique (`Static`) pour les variables globales, la pile (`Stack`) pour l'espace alloué lors des appels de fonctions et le tas (`Heap`) pour les allocations dynamiques.

Chacun de ces espaces est un tableau d'octets. Seule la taille du premier est connu lors de la compilation, les autres sont dynamiquement augmentés lors de l'exécution du programme *Léa*.

Espace statique (*Static*)

Les variables déclarées statiquement y sont rangées par ordre croissant. La somme des octets occupés par ces variables représente la taille en octets de cet espace. Les variables ont une place qui correspond au décalage à partir de zéro.

Mettons un peu de code au format *Léa* dans un programme `yacc` pour illustrer ceci :

```
// On met le compteur de place `a z'ero
// au d'ebut du programme
program:
  {inheritedOffset = 0;} type_declaration_part
   variable_declaration_part
   procedure_and_function_definition_part TOKEN_BEGIN
   statement_part TOKEN_END;

// On ajoute la place occup'ee de chaque variable
variable_declaration:
  identifier_list ':' type ';'
  {
    foreach i in $1 do
      add(inheritedEnvironment, i^.name,
        inheritedOffset);
    // On incr'emente le compteur de place
    inheritedOffset += size($3);
  };
```

Pile (*Stack*)

La pile est également un tableau d'octets organisé en liste simplement chaînée de blocs d'activation (*activation frame*). Chacun de ces blocs contient tout ce qui est nécessaire au fonctionnement d'une fonction ou d'une procédure interprétée :

Astuce

Si la taille de l'espace mémoire est insuffisante, il suffit de la doubler, et de réallouer avec une commande comme celle-ci (en C) : `size*=2; buffer=(uint8_t *)realloc(buffer,size);`

Désignation	Nom	nombre d'octets
Pointeur vers le bloc d'activation précédent	*previous*	4
Pointeur vers le bloc d'activation suivant	*next*	4
Adresse de l'argument en cours pour cet enregistrement	*offset*	1
Registre où sera enregistré la valeur de retour	*return*	1
Arguments de la fonction	*args*	taille des arguments
Variables locales à la fonction	*vars*	taille des variables locales

Comme tous les types sont connus lors de la compilation, la taille de ce bloc est connu au moment de l'appel de la fonction.

Appel d'une fonction par l'interpréteur

L'algorithme suivant peut être appliqué :

Soit `foo` une fonction dont le type est $\tau_1 \rightarrow \tau_2$ et dont la taille des variables locales est k.

L'appel $foo(x_1, x_2, ...x_k)$ produit l'effet suivant :

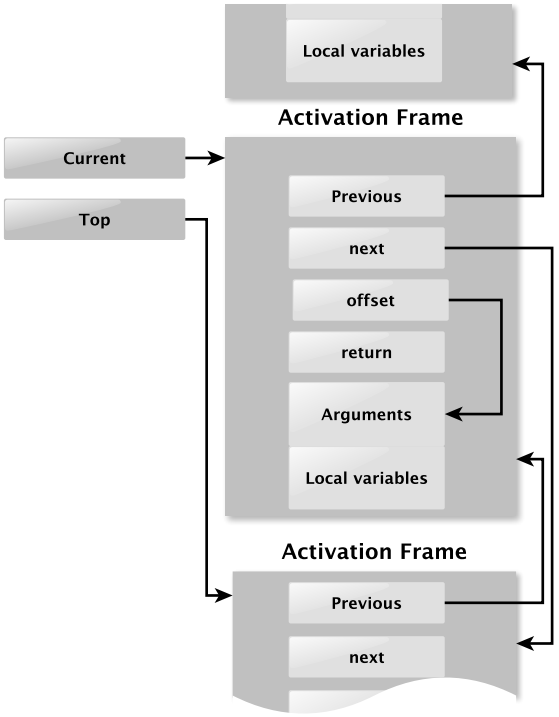

FIGURE 2 – Bloc d'activation (*Activation frame*)

1. Créer un nouveau bloc d'activation dont la taille est $10 + size(\tau_1) + size(\tau_2) + k$. Conserver sa place dans New
2. Affecter *previous* vers le bloc précédent : $New\hat{}.previous := Actual$
3. Affecter *next* vers le bloc suivant : $New\hat{}.next := Actual{+}10{+}size(\tau_1){+}size(\tau_2) + k$
4. Initialiser le compteur $offset$ à 10 (position relative du premier argument)
5. Pour i allant de 1 à k, Interpréter l'expression x_i et placer sa valeur à $New + offset$, incrémenter $offset$ de la taille du type de x_i [2]
6. Affecter *Actual* à la valeur de New

2. À ce moment, le bloc d'activation actuel est toujours inchangé et les évaluations des arguments se font bien dans la procédure appelante.

7. Interpréter le code de foo : Les valeurs locales n ont leur place en $Actual + n\hat{}.place$, l'instruction $return(e)$ copie la valeur de e dans le registre *return*
8. Dépiler le dernier bloc d'activation en exploitant le pointeur vers le bloc précédent

Astuce

L'appel à une fonction fait intervenir une commande qui "empile" les arguments. L'appel d'une fonction $foo(arg_1, arg_2, \ldots, arg_k)$ correspond à un arbre de syntaxe comme ceci : $CALL("foo", PUSH(arg_1), PUSH(arg_2), \ldots)$ L'interprétation de $PUSH(arg_i)$ est la suivante : a) Évaluer arg_i dans le bloc d'activation courant b) Le copier dans le nouveau bloc $+offset$ c) Incrémenter $offset$ de la taille de arg_i

Prenons un exemple. Soit le code suivant :

```
function foo (a: integer, b : boolean): character
var
c : integer;
begin
  return 'X';
end

[...]

begin
[...]
println(foo(32, true));
[...]
end
```

Soit π l'adresse de l'actuel bloc d'activation.

L'expression `foo(32, true)` aura comme premier effet d'allouer un nouveau bloc d'activation de 17 octets dans l'espace mémoire *Stack*.

Ce bloc d'activation sera le suivant :

	nombre d'octets	place relative	valeur
Previous	4	0	π
Taille du bloc	4	4	18
a	4	8	*default value*
b	1	12	*default value*
c	4	13	*default value*
Valeur de retour	1	17	*default value*

Ensuite les arguments seront évalués dans le bloc d'activation *Actual* toujours inchangé.

Le bloc d'activation sera le suivant :

	nombre d'octets	place relative	valeur
Previous	4	0	π
Taille du bloc	4	4	18
a	4	8	**32**
b	1	12	**1**
c	4	13	*default value*
Valeur de retour	1	17	*default value*

Puis le bloc d'activation actuel devient celui-ci et l'évaluation du code affecte la valeur de retour.

	nombre d'octets	place relative	valeur
Previous	4	0	π
Taille du bloc	4	4	18
a	4	8	32
b	1	12	1
c	4	13	*default value*
Valeur de retour	1	17	**'X'**

La valeur de retour est donnée comme valeur pour l'expression à afficher.

Tas (*Heap*)

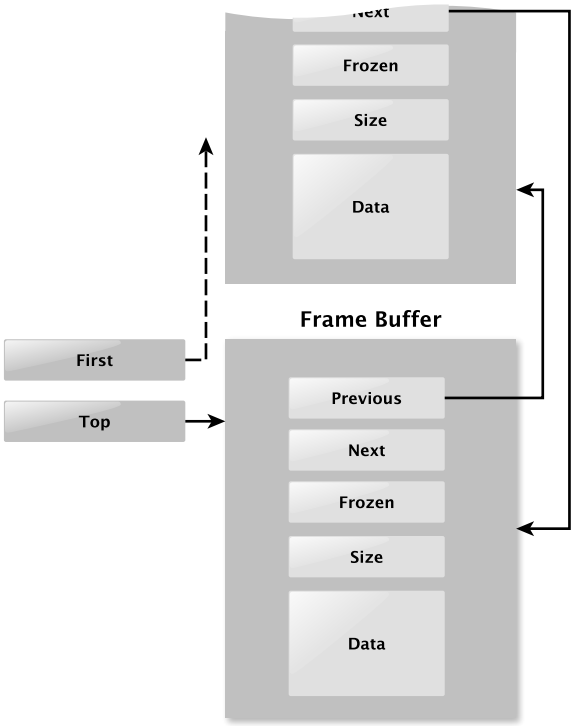

FIGURE 3 – Bloc (*Frame buffer*)

Le programmeur utilisant le langage *Léa* peut allouer dynamiquement de la mémoire explicitement grâce à la commande `new(p)`. Il faut pour cela que `p` soit une variable déclarée de type $\hat{}\tau$.

L'effet de cette commande est d'allouer un nouvel espace de la taille du type τ.

Le tas est un tableau d'octets organisé en une liste doublement chaînée de

blocs.

Chacun des blocs contient :

1. Les pointeurs vers les blocs précédents et suivants.
2. Un booléen qui indique si ce bloc est libre ou non (*frozen*)
3. La taille en octets du bloc
4. Un espace contigu d'octets pour y placer les données

L'algorithme suivant peut être appliqué à l'interprétation de la commande $new(p);;$

1. Chercher en parcourant tous les blocs à partir du premier, un bloc qui serait libre (*frozen* = *true*) et dont la taille serait supérieur ou égale à la taille de $p\hat{}$.
2. S'il existe, affecter à p l'adresse de ce bloc et mettre *frozen* à `false`
3. Sinon, créer un nouveau bloc de la taille de $p\hat{}$, y affecter le pointeur vers le précédent, sa taille, *frozzen* à false. Affecter le pointeur `next` du bloc précédent à celui-ci. Affecter à p l'adresse de ce bloc.

L'algorithme suivant peut être appliqué à l'interprétation de la commande $dispose(p);;$

1. Mettre *frozen* à `false` dans le bloc pointé par p
2. Affecter *null* à p

Cette organisation du tas est suffisante pour développer un ramasse-miettes que nous ne demandons pas d'implémenter.