Cours 3 : Compilation d'un langage fonctionnel - bibliothèque d'exécution

La principale difficulté de la mise en œuvre d'un langage fonctionnel vient de la traduction du contrôle implicite de l'exécution dans un modèle de calcul où il devient explicite.

La première méthode consiste à réécrire le programme avec un contrôle explicite. La transformation CPS (Continuation Passing Style [Ste78],[AJ89]) réécrit toutes les expressions en expressions closes (sans variables libres) et indique, en le passant comme argument supplémentaire, la continuation (reprise de calcul) à appliquer en fin de calcul d'une fonction.

La deuxième méthode consiste à utiliser une machine abstraite. Il existe deux grandes familles de machines abstraites pour les langages fonctionnels :

- Les machines à environnements construisent des couples (code,environnement) pour les fermetures. La partie code peut contenir des variables libres et la partie environnement les valeurs de ces variables. La machine SECD (Stack Environment Control Dump) a été un des premiers représentants de cette stratégie. Les machines FAM (Functional Abstract Machine) et CAM ([GCS86]) (Categorical Abstract Machine) reprennent ce type d'architecture avec des variantes sur la représentation de l'environnement. La CAM chaîne les environnements alors que la FAM les duplique.
- Les machines à réduction de graphes représentent les expressions par un graphe. Les nœuds des graphes sont des nœuds d'application, et les feuilles sont des combinateurs ou des constantes. La SK machine a été un des premiers représentants. On en rencontre d'autres comme la G-machine ou la TIM .

Les machines à pile et environnement sont plutôt utilisées pour la stratégie d'évaluation stricte et les nmachines à réduction de graphes pour les stratégies paresseuses. Ce n'est pas une conséquence théorique mais une habitude comme le montre le choix de la machine de Krivinne pour compiler le λ-calcul. L'exécution des instructions de ces machines virtuelles repose sur deux modèles : interpréteur ou compilateur.

- Un interpréteur exécute les instructions de cette machine. Cette méthode est, en règle générale, assez peu efficace. On tombe dans les travers des émulations. Néanmoins, certaines mises en œuvre, comme Caml-light ([Ler90]), donnent des temps de calcul raisonnables.
- Un compilateur traduit directement les instructions de la machine virtuelle par une série d'instruction du processeur cible (compilateur natif). Cette solution donne des résultats bien supérieurs à la précédente. La difficulté des compilateurs natifs est d'avoir un générateur de code spécifique à chaque processeur. Elle subsiste même en simplifiant cette traduction par l'emploi d'un langage intermédiaire plus simple .

La figure 1 montre les différentes voies de traduction d'un langage fonctionnel.

Les transformations de programme comme le λ-lifting ([Joh85]) pour la gestion des environnements des déclarations locales, la transformation de programmes CPS (Constinuation Passing Style (CPS[AJ89])) pour le contrôle de l'exécution, seront étudiés dans des prochains cours.

Les machines abstraites comme la machine de Krivine ou la CAM sont prévues effectivement pour la compilation du λ-calcul et des langages fonctionnels. Ces machines ne sont pas les plus répandues, pour cela on cherche à comprendre le modèle de compilation vers des machines à objets ou impératives. On présente dans ce cours un compilateur d'un mini-ML appelé `ml2java` vers du code Java objet. On s'intéresse par la suite à la compilation vers C pour le modèle impératif.

1. Organisation de ml2java

Le programme ml2java, écrit en Caml-light, se décompose de la manière suivante :

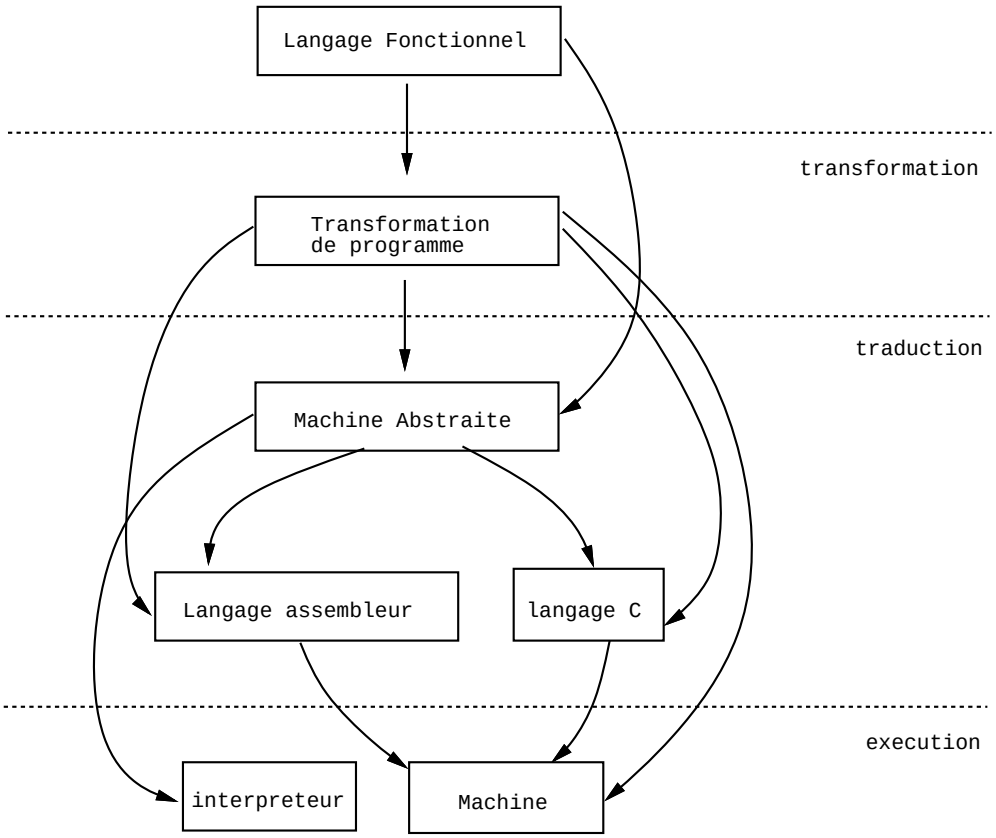

FIG. 1 – Compilation d'un langage fonctionnel

module	fonction	
util.ml	utilitaire	
types.ml	définition des types de mini-ML	
alex.mll	analyseur lexical	
asyn.mly	analyseur syntaxique	
typeur.ml	le typeur	
env_typeur	environnement	
intertypeur.ml	toplevel du typeur	*
eval.ml	évaluateur	*
env_eval	environnement de l'évaluateur	*
intereval.ml	toplevel de l'évaluateur	*
env_trans.ml	environnement du traducteur	
lift.ml	un pseudo λ-lifting	
trans.ml	le traducteur vers un LI	
prod.ml	le traducteur LI vers java	
comp.ml	le compilateur complet	
maincomp.ml	l'analyseur de la ligne de commande	

Les lignes étoilées correspondent à des fichiers inutiles pour la construction du compilateur. Ils permettent de tester le typeur et de construire facilement un évaluateur.

Actuellement le typeur fonctionne uniquement pour la partie fonctionnelle pure. Un projet sera présenté pour intégrer la non généralisation des variables de types pour les expressions expansives. De même la phase de λ-lifting n'est pas complète. Elle n'effectue qu'une globalisation des termes clos. La aussi un projet sera demandé.

2. Bibliothèque d'exécution

L'intérêt de Java comme langage cible d'un compilateur de langage fonctionnel est principalement pédagogique car Java permet d'avoir une bibliothèque d'exécution très réduite comme nous allos le voir. Pour ce qui concerne mini-ML, cela est du principalement au GC inclus dans Java qui évite de devoir en écrire un. Pour un ML plus complet, les exceptions de Java peuvent être utilisées pour implanter les exceptions de ML.

Cette bibliothèque représente les types ML comme sous-classe de la classe abstraite `MLvalue` :

```
abstract class MLvalue extends Object {

  abstract void  print();
}
```

On construit les entiers comme cela :

```
class MLint extends MLvalue
{
   private int val;
   MLint(int a){val=a;}


   public void print(){System.out.print(val);}
   public int MLaccess(){return val;}
}
```

La classe `MLint` est une sous-classe concrète de la classe abstraite `MLvalue`. Elle ne possède qu'un seul champs de données (`val`), un constructeur nécessitant un argument de type `int`, un accesseur `MLaccess` et la méthode `print`. Tous les autres types de base sont construits de cette manière à l'exception des types fonctionnels. Les valeurs des types fonctionnels correspondent aux fermetures de ML. On définit la classe abstraite `MLfun` suivante pour ces valeurs :

```
abstract class MLfun extends MLvalue
{
  public int MLcounter;
  protected MLvalue[] MLenv;

  MLfun(){MLcounter=0;}
  MLfun(int n){MLcounter=0;MLenv = new MLvalue[n];}

  public  void  MLaddenv(MLvalue []O_env,MLvalue a)
   { for (int i=0; i< MLcounter; i++) {MLenv[i]=O_env[i];}
     MLenv[MLcounter]=a;MLcounter++;}

abstract public MLvalue invoke(MLvalue x);

  public void print(){
     System.out.print("<fun> [");
     for (int i=0; i< MLcounter; i++)
           MLenv[i].print();
     System.out.print("]");
  }
}
```

Seule la méthode `invoke` est abstraite. Elle correspond à l'application d'une fermeture `f` à un argument `a` et se traduit par l'envoi de message suivant : `f.invoke(a)`. Le constructeur avec argument entier alloue le nouvel environnement. Le champ `MLcounter` correspond aux nombres de valeurs de l'environnement. La constante `MAX` est l'arité de la fonction.

On créera une sous-classe de cette classe pour chaque fonction définie en ML. La méthode `invoke` sera alors implanter dans la sous classe concrète. Soit la fonction ML `app` suivante :

```
let app = function fx -> function x -> fx x;;
```

Sa traduction est :

```
class MLfun_app___76 extends MLfun {

  private static int MAX = 2;

  MLfun_app___76() {super();}

  MLfun_app___76(int n) {super(n);}

  public MLvalue invoke(MLvalue MLparam){
    if (MLcounter == (MAX-1)) {
      return invoke_real(MLenv[0], MLparam);
    }
    else {
      MLfun_app___76 l = new MLfun_app___76(MLcounter+1);
      l.MLaddenv(MLenv,MLparam);
      return l;
    }
  }

  MLvalue invoke_real(MLvalue fx___77, MLvalue x___78) {
    {
      MLvalue T___79;
      MLvalue T___80;
      T___79=fx___77;
      T___80=x___78;
      return ((MLfun)T___79).invoke(T___80);
    }
  }

}
```

Quand `invoke` est déclenchée, si tous les arguments sont passés, alors la méthode correspondant au code de la fonction est appelée, sinon une nouvelle fermeture est créee avec un environnement agrandi (méthode `MLaddenv`. Les identificateurs ont tous une extension numérique pour garantir l'unicité des noms.

3. Schémas de compilation

On cherche à décrire sous forme de règles la traduction d'une phrase ML dans un langage intermédiaire (LI) simple. La section suivante montre comment traduire les instructions du LI en Java. Un programme ML est une suite de déclarations globales (les expressions ont été transformées en déclarations non fonctionnelles). On ne trouve plus de λ dans une expressions non fonctionnelles, car toutes les fonctions (les fonctions anonymes ont été nommées) ont été closes puis globalisées.

On construit alors les schémas de compilation dans un environnement de compilation E (décrit par la suite) de la manière suivante :

```
[ e ]E   -> INSTRUCTION(param)
```

indiquant que la compilation de `[e]` dans l'environnement `E` se réécrit en `INSTRUCTION(param)`.

On note `c` une constante, `v` une variable et `e` une expression quelconque.

L'environnement de compilation possède 4 composantes : `g` pour l'environnement des noms, `r` indiquant s'il y a un return, `d` indiquant s'il y a une déclaration, et `t` le type de l'expression. Sachant qu'il ne peut y avoir en même un return et une affectation, on obtient alors le système de règles suivant :

```
constante
---------

  [ c ](g,F,"",t)   -> CONST(c,g,t)
  [ c ](g,T,"",t)   -> RETURN(CONST(c,g,t))
  [ c ](g,F,D,t)    -> AFFECT(D,CONST(c,g,t))
```

Une constante ML se traduit par une constante LI. `T` indique qu'il y a un retour (`F` sinon+), `D` indique une déclaration (`""` sinon).

```
variable
--------

  [ v ](g,F,"",t)   -> VAR(v,g,t)
  [ v ](g,T,"",t)   -> RETURN(VAR(v,g,t))
  [ v ](g,F,D,t)    -> AFFECT(D,VAR(v,g,t))
```

Une variable ML se traduit en variable LI en fonction de son nom dans `g` et de son type.

```
conditionnelle
--------------

  [ if v1 then v2 else v3 ](g,r,d,t) ->  IF([v1](g,F,"",bool),
                                            [v2](g,r,d,t),[v3](g,r,d,t))

  [ if e1 then e2 else e3](g,r,d,t) -> [let v1 = e1 in
                                        let v2 = e2 in
                                        let v3 = e3 in
                                         in if v1 then v2 else v3](g,r,d,t)
```

Si on sait traduire le `if then else` avec uniquement des variables, on déduit imem'diatement la compilation générale de cette construction. Les `let in` de la deuxième règle sont en fait des `let and in` .

```
application
-----------

  [ v1 v2 ](g,F,"",t) -> APPLY([v1](g,F,"",_),[v2](g,F,"",_))
  [ v1 v2 ](g,T,"",t) -> RETURN(APPLY([v1](g,F,"",_),[v2](g,F,"",_)))
  [ v1 v2 ](g,F,D,t)    -> AFFECT(D,APPLY([v1](g,F,"",_),[v2](g,F,"",_)))
  [ e1 e2 ](g,r,s,t) -> [let v1 = e1
                         and v2 = e2
                           in v1 v2](g,r,s,t)
```

On traite les trois cas simples avec uniquement des variables, pour ensuite traiter le cas général de l'application.

```
declaration locale
------------------

  [let v1 = \x.e1 in e2](g,r,d,t) -> ERREUR!!!

  [let v1 = e1 in e2](g,r,d,t)   ->
          BLOCK(w1=N(v1),t,[e1](g,F,(w1,t),e2((v1,w1)::g,r,d,t))
```

On crée des nouveaux noms (`N(v)`) pour s'assurer de l'unicité de chacun.

Les expressions globales se réécrivent comme des déclaration globales :

```
  [ e ](g)  -> [let w = e;;](g) avec w=N(v)
```

on obtient ainsi que des déclarations globales. Les règles de traduction des variables globales sont les suivantes :

```
declaration globale
-------------------

 [Let v1 = \x.e1 ](g,t)       ->  FUNCTION(w1=N(v1),t,1,[x],[e1](g,T,"",_))
 [LetRec v1 = \x.e1](g,t)     ->  FUNCTION(w1=N(v1),t,1,
                                           [x],[e1]((x,N(x))::(v1,w1)::g,T,"",_))

 [Let v1 = e1](g,t)             ->    VARGLOBAL(w1=N(v1),t,[e1](g,false,"w1",t))
 [LetRec v1 = e1](g,t)          ->    ERREUR
```

Le langage intermédiaire est le suivant :

```
type LI_const_type =
  INTTYPE
| FLOATTYPE
| BOOLTYPE
| STRINGTYPE
| UNITTYPE
;;

type LI_type =
  ALPHA
| CONSTTYPE of LI_const_type
| PAIRTYPE
| LISTTYPE
| FUNTYPE
| REFTYPE
;;

type LI_const =
  INT of int
| FLOAT of float
| BOOL of bool
| STRING of string
| EMPTYLIST
| UNIT
;;

type LI_instr =
  CONST of LI_const
| VAR   of string * LI_type
| IF  of LI_instr * LI_instr * LI_instr
| PRIM of (string * LI_type) * LI_instr list
| APPLY of LI_instr * LI_instr
| RETURN of LI_instr
| AFFECT of string * LI_instr
| BLOCK of (string  * LI_type * LI_instr) list  * LI_instr
| FUNCTION of string * LI_type * int * (string list * LI_type) * LI_instr
;;
```

On remarque qu'il est plus simple que le langage d'expression de mini-ML. De plus il est complètement indépendant. La construction `PRIM` correspond aux primitives (+,=). La construction `BLOCK` est un bloc pouvant contenir des déclarations locales. Enfin la construction `FUNCTION` sert pour la définition de fonctions. Il ne reste plus qu'à choisir le langage cible final.

4. Production du code Java

Un programme ML est traduit à la phase précédente par une liste d'instructions du LI. La production du code Java s'effectue en trois passes qui chacune parcourt cette liste. La première récupère toutes les définitions de fonctions et produit les classes Java équivalentes. La deuxième effectue les déclarations globales de la classe principale (contenant la méthode `main`) des fonctions, des variables globales non fonctionnelles et des noms associés aux expressions globales. Enfin la troisième passe écrit la fonction `main` correspondant aux différentes expressions globales rencontrées dans le programme. Ces trois passes se retrouvent dans le fichier `prod.ml` qui est un lien symbolique vers le fichier `prodjava.ml`. On peut aisément imaginer de traduire ce LI vers un autre langage.

5. Exemples de compilation

Soitla fonction fib définie en mini-ML de la façon suivante :

```
let rec fib   = function x -> if x < 2  then 1 else (fib(x-1))+(fib(x-2));;
```

Voici le texte de la méthode `invoke_real` correspondante :

```
MLvalue invoke_real(MLvalue x___3) {

  {
    MLvalue T___4;
    {
      MLvalue T___5;
      MLvalue T___6;
      T___5=x___3;
      T___6=new MLint(2);
      T___4=MLruntime.MLltint( (MLint )T___5,(MLint )T___6);
    }
    if (((MLbool)T___4).MLaccess())
      {
        MLvalue T___7;
        T___7=new MLint(1);
        return T___7;
      }
    else
      {
        MLvalue T___8;
        {
          MLvalue T___9;
          MLvalue T___14;
          {
            MLvalue T___10;
            MLvalue T___11;
            T___10=fib.fib___2;
            {
              MLvalue T___12;
              MLvalue T___13;
              T___12=x___3;
              T___13=new MLint(1);
              T___11=MLruntime.MLsubint( (MLint )T___12,(MLint )T___13);
            }
            T___9=((MLfun)T___10).invoke(T___11);
          }
          {
            MLvalue T___15;
            MLvalue T___16;
            T___15=fib.fib___2;
            {
              MLvalue T___17;
              MLvalue T___18;
              T___17=x___3;
              T___18=new MLint(2);
              T___16=MLruntime.MLsubint( (MLint )T___17,(MLint )T___18);
            }
            T___14=((MLfun)T___15).invoke(T___16);
          }
          T___8=MLruntime.MLaddint( (MLint )T___9,(MLint )T___14);
        }
        return T___8;
      }
  }
}
```

C'est probablement le code le plus naif que l'on rencontre heureusement rarement dans des compilateurs, mais il a plusieurs avantages : il correspond aux schémas de compilation décrits, il reste clair et il est propice à des exercices de simplification et d'optimisation.

Références

[AJ89] A. Appel and T. Jim. Continuation-passing style, closure-passing style. *ACM on POPL*, 1989.

[GCS86] M. Mauny G. Cousineau, P.L. Curien and A. Suarez. Combinateurs Catégoriques et Implémentations des Langages Fonctionnels. Technical Report 3, LIENS, 1986.

[Joh85] Thomas Johnsson. Lambda lifting : transforming programs to recursive equations. In *Conference on Functional Programing Languages and Computer Architecture. LNCS 201*, Nancy, 1985. ACM, Springer Verlag.

[Ler90] X. Leroy. The zinc experiment : an economical implementation of the ml language. Technical Report 117, INRIA, February 1990.

[Ste78] G. L. Steele. Rabbit : a compiler for scheme. Technical Report AI-TR-474, MIT, 1978.