Document Technique - Back End

Nasser ADJIBI

6 avril 2015

Historique :
— 28/04/2015 : création
— 31/04/2015 : ajout specs. de la famille de générateurs
— 04/04/2015 : specification des attentes du back end
— 06/04/2015 : réorganisatioon de la structure du document

Table des matières

1 Rappel de l'enchainement des composants du compilateur

Le compilateur suivra la châine simple de compilation suivante :

1. **Parsage des fichiers dont les noms ont été donnés en argument**
   - **Nom de la fonction** : parse
   - **Paramètres** :
     + KawaTree* : la structure qui contiendra le squelette du programme
     + vector<string> : liste des noms des fichiers
   - **Indication** : échoue si le programme est mal formé. Le KawaTree est alimenté, et est prêt pour l'analyse syntaxique.
2. **Analyse syntaxique**
   - **Nom de la fonction** : syntaxAnalyse
   - **Paramètres** :
     + KawaTree* : la structure qui contenant le squelette du programme
   - **Indication** : si aucune érreur est détectée, le KawaTree a été décorer et contient toutes les informations nécéssaire pour la génération du code intermédiaire.
3. **Génération du code intermédiaire**
   - **Nom de la fonction** : generateIRCode
   - **Paramètres** :
     + KawaTree* : la structure qui contenant le squelette du programme, décoré.
   - **Indication** : un fichier .ll contenant le code intermédiaire est généré.
4. **Optimisation et compilation du code intermédiare.**

2 Architecture du back end

Le back-end est composé de deux parties. La premiere partie dont le rôle est de générer le code intermédiare, et une deuxieme partie dont le rôle est de transformer le code en code machine au format ELF.

2.1 Le générateur du code intermédiaire LLVM

Il traduit l'arbre syntaxique du programme en code intermédiaire LLVM.Le générateur utilise le KawaTree pour obtenir les informations nécéssaires à la génération du programme.
Ce dernier est composé de deux sous-parties :

- Le IRCompiler est le composant responsable de la production du code intermédiare. Il parcours l'arbre syntaxique grâce au KawaTree et appelle les objets objets de production du code intermédiaire.s

— La famille de générateurs, un ensemble classes utilisées pour la génération du code intermédiaire. Il s'agit d'une sur-couche à l'API LLVM déja existante.

2.2 Structure du IRCompiler

Les noeuds de l'abre syntaxique et le génération de code étant découplée, le *patron visiteur* est adapté pour l'implémentation du IRCompiler. Il comprend une table de symbole pour l'aider à la génération du code intermédiaire. (Voir le diagramme dans l'annexe)

Liste des Fonctions :
— LLVM : :Value* getVariable(string) : retourne un LLVM : :Value* associé à la variable
— LLVM : :Value* enterBloc() : notifie l'entrée dans un bloc d'instruction
— LLVM : :Value* leaveBloc() : notifie la sortie d'un bloc d'instruction
— LLVM : :Value* compile(KT_Program) : compile le programme
— LLVM : :Value* compile(KT_Package) : compile un package
— LLVM : :Value* compile(KT_Class) : compile une classe
— LLVM : :Value* compile(KT_Interface) : compile une interface
— LLVM : :Value* compile(KT_Expression) : compile une expression
— LLVM : :Value* compile(KT_Methode) : compile une méthode
— LLVM : :Value* compile(KT_Statement) : compile une instruction
— LLVM : :Value* compileEntier(KT_Entier) : compile un entier
— LLVM : :Value* compileFloat(KT_Float) : compiler un floatant
— LLVM : :Value* compileCallMethode(KT_CallMethode) : compile l'appelle à une méthode
— LLVM : :Value* compileLoadAttribut(KT_LoadAttribut) : compile le chargement d'un attribut
— LLVM : :Value* compileFor(KT_For) : compile une boucle for
— LLVM : :Value* compileWhile(KT_While) : compile une boucle while
— LLVM : :Value* compileAffectation(KT_Affectation) : compile une affectation
— LLVM : :Value* compileDeclaration(KT_Declaration) : compile une declaration
— LLVM : :Value* compileConstructeur(KT_Constructeur) : compile un constructeur
— LLVM : :Value* compileString(KT_String) : compile une chaîne de caractère

2.3 Famille de génération de code

Il s'agit d'une sur-couche aux classes de génération de code déja existantes. Elle est composée d'un ensemble de classes implémentant l'ensemble des spécifications qui suivent. Les spécification pourront être implémentées séparément pour le mode monolithique, et pour le mode partagé :

— **AffectationGenerator** : gère les opérations d'affectation
— **BlockGenerator** : gère la construction des blocs d'instruction
— **CallGenerator** : gère les appels de méthodes et de constructeurs
— **FunctionGenerator** : gère la creation de fonction et de méthodes
— **GlobalVariableGenerator** : gère la creation de variables gloabales néséssaire à la cohésion du programme
— **NameBuilder** : gère le nommage des elements du code intermédiare
— **PrimitiveBinaryOperation** : gère les opérations binaire
— **PrimitiveCreator** : gère la création des primitives
— **PrimitiveValueConverter** : gere la convertion des valeurs primitieves
— **TypeGenerator** : gère le creation et l'accès au typage LLVM

2.4 La chaîne de compilation du code IR

LLVM nous fournit des outils de compilation permettant de traduit du code IR LLVM bien formé en assembleur. Le code assembleur peut ensuite être traduit en code machine. Cette chaine de compilation est composée des éléments suivant :

— **llc** compile le fichier source en un fichier assembleur que l'on peut passer à un linker pour generer un executable
— **clang** compilateur permettant de compiler le code assembleur.

3 Fonctionnment dynamic

3.1 Fonctionnment dynamic du IRCompiler

Le IRCompiler prendra en entrée un arbre syntaxique représentant la structure du programme à compiler. Le noeuds de l'arbre de cet arbre doivent implémenter les interfaces de la famille d'interface AST. (Voir annexe). Au cours de la production, on decorera l'arbre syntaxique avec de objets LLVM de rendre les noeud réutilisables pour la production de code. La production du code LLVM IR d'un noeud de l'arbre syntaxique, entrainera la production en cascade des noeuds fils de ce noeud. Le IRCompiler est responsable des algorithmes de génération de code IR. Selon le mode de compilation, il implémente différent algoriithme.

— MonolithiqueIRCompiler Compilateur en mode monolithique. Il produit un fichier .ll unique contenant le code intermédiaire représentant toute les classes compilées.
— SharedIRCompiler Compilateur en mode partagé, génére plusieurs fichiers .ll représentant les classes à compilées.

3.1.1 Parcours du KawaTree par le compilateur

Le IRCompiler parcourra le composant **KawaTree** de la manière suivante.

1. Compilation du Noeud représentant le programme

- Création de l'objet LLVM : :Module repésentant le programme. Nommage du module au nom du programme
- Parcours des packages

2. Compilation des packages
   - Parcours de la liste d'interfaces
   - Pour chaque interface, génération de la structure la représentant
   - Parcours de la liste de classes
3. Compilation d'une interface
   - Production d'une structure représentant l'interface.
   - Parcours de la liste des méthodes
   - Pour chaque méthode, déclaration de l'entête
4. Compilation d'une classe
   - Compilation de la classe parent si elle ne l'est pas déja.
   - Compilation de la liste d'attribut en une structure représentant la classe.
   - Compilation des entêtes des méthodes de la classe actuelle.
   - Génération des tables AdHoc associées aux méthodes polymorphes.
   - Compilation du corps des méthodes
5. Compilation de la structure représentant la classe
   - Récupération du nom de la classe
   - Récupération de la liste des types
   - Récupération de la liste des noms des attributs
   - Récupération de la liste de booleens spécifiant si un attribut est static.
   - Récupération de la liste de booleens spécifiant si un attribut est final
   - Génération de la structure grâce au composant TypeGénérator.
6. Compilation des entêtes de méthodes. Pour chaque noeud de méthode doit fournir les informations suivantes :
   (a) Le nom de la classe de déclaration
   (b) La liste des types des paramètres
   (c) La liste des noms des paramètres
   (d) Un booleen spécifiant si la méthode est static
7. Compilation des tables AdHoc
   - Table quand la classe statique est aussi la classe dynamique
     La liste de méthode doit être formée de la manière suivante
     (a) *Recuperer la liste des méthodes héritables de la classe parent.(Elle est vide si la classe n'a pas de parent)*
     (b) *Remplacer les méthodes qui ont été redéfinies au bon endroit.*
     (c) *Concatener à cette liste les méthodes nouvellement définies, dans l'ordre lexico-graphic.*
     (d) *Retirer les méthodes statiques et privée. Cette liste sera aussi celle consulter par les classes filles* Compiler de la liste précédente avec le GlobalValueGenerator

— Table pour la classe parent ou interface en static, classe fille en dynamique
Faire correspondre la bonne méthode
(a) *Recuperer la liste des méthodes héritables de la classe parent(construite comme plus haut.)*
(b) *Remplacer les redéfinitions au bon endroit dans la liste* Compiler de la liste précédente avec le GlobalValueGenerator

Le compilation des listes génère les indexes perméttants les appels polymophiques.

8. Compilation du corps des méthodes
   — Initialisation d'un bloc d'instruction.
   — Compilation du bloc d'instruction de la méthode
9. Compilation d'un bloc d'instruction
   Chaque instruction etre capable de recuper d'une manière ou d'un autre la fonction à laquelle elle appartient. Les cas possibles sont :
   — Compilation d'un return
   — Compilation d'une déclation
   — Compilation d'une affection
   — Compilation d'appel de méthode.
   — Compilation d'un appel de méthode static
   — Compilation d'une expression conditionnelle
10. Génération d'une instruction de return
    On utilise l'objet BasicInstructionGenerator
    — Si le noeud est un return void on emploie la fonction createRetunrVoid
    — Si le noeud est un return d'un objet, on génère l'expression à retourner puis l'instruction return
11. Compilation d'une déclation
    — Recupération du type de la variable
    — Récupération du nom de la varible
    — Génération de l'instruction d'allocation avec l'objet AllocationGénérator
    — Empilement de la valeur llvm générée dans la table de symbole du IRCompiler
12. Compilation d'une affection
    On utilisera l'objet BasicInstructionGenerator
    — Chargement de l'adresse pour une affectation d'affectation
    — Génération de la valeur LLVM de l'expression à affecter.
    — Génération du code de l'opération d'affectation
13. Chargement d'une adresse pour une affectation
    — Si il s'agit d'une variable, on récupère sa valeur grâce à son nom et la table de symboles du IRCompiler
    — Si il s'agit d'un accès à l'attribut d'une variable, on utilise le nom de l'attribut et le type d'une variable pour determer l'index de l'attribut dans l'objet.

— Si on a un attribut static, on y accède avec le nom de sa classe et le nom de l'attribut.

14. Génération de la valeur llvm d'une expression à affecter
    — Si l'expression est un noeud représentant une primitive, on utilise l'objet PrimitiveGenérator
    — Si l'expression est un noeud représentant un expression binaire on génère les valeurs LLVM des opérandes puis on appelle la fonction de l'objet BinaryPrimitiveOperation, correspondant à la l'opérateur
    — Si l'expression est un noeud représentant l'accès à la valeur d'une variable, on charge l'adresse de la variable à partir de son nom et de table de symboles du IRCompiler puis on cherche, puis on charge la valeur de la variable.
    — Si l'expression est un noeud d'appel de fonction, on determine l'index de la fonction dans la table adHoc grâce au noeud représentant la méthode déduit statiquement. Ce dernier doit fournir le nom la méthode, la liste des types, le type de retour et la le nom de la classe où la méthode a été trouvée. Il faudra avoir à disposition la valeur LLVM de l'appelleur.
    — Si l'expression à obtenir est un attribut static, le nom de la classe et le nom de la variable est attendu.
    — Si l'expression à obtenir est un attribut, le nom de la classe et le nom de la variable est attendu. Il faudra avoir à disposition le neoud de l'appelleur.
    — Si l'expression à obtenier est appel à une méthode static, le noeud de la méthode ainsi que la liste des noms des variable
15. Compilation d'un instruction conditionnelle Les cas possibles sont :
    — Compilation d'un if
    — Compilation d'un if else'
    — Compilation d'un for
    — Compilation d'un while
16. Compilation d'un if
    — Complation de l'expression représentant la condition
    — Creation du nouveau bloc d'instruction avec BlocGenerator.createIF
    — Initialisation du bloc if
    — Notifier le IRCompiler de l'entrer dans un bloc
    — Appointer le nouveau bloc comme le bloc courant de génération d'instructions
    — Récuper la liste des noeud d'instructions
    — Compiler la liste d'instructions
    — Notificier le IRCompiler de la sortie d'un bloc
    — Fermeture du if avec BlocGenerator : :endIF
17. Compilation d'un if else
    — Complation de l'expression représentant la condition
    — Creation d'un nouveau bloc d'instruction avec BlocGenerator : :createIF

— Creation d'un nouveau bloc d'instruction avec BlocGenerator : :createElse
— Initialisation du bloc pour le if
— Notifier le IRCompiler de l'entrer dans un bloc
— Appointer le nouveau bloc comme le bloc courant de génération d'instructions
— Récuper la liste des noeud d'instructions
— Compiler la liste d'instructions
— Notificier le IRCompiler de la sortie d'un bloc
— Fermeture du if avec BlocGenerator : :endIF
— Initialisation du bloc else
— Appointer le nouveau bloc comme le bloc courant de génération d'instructions
— Récuper la liste des noeud d'instructions
— Compiler la liste d'instructions
— Notificier le IRCompiler de la sortie d'un bloc
— Fermeture du else avec BlocGenerator : :endElse

18. Compilation d'un for
    — Compilation de l'instruction d'initialisation
    — Construction d'un nouveau bloc pour le for
    — Initilisation du bloc For
    — Recupération et compilation de la liste d'instruction
    — Recupération et compilation de l'instruction d'itération
    — Fermeture du for BlocGenerator : :endFor
19. Compilation d'un while
    — Compilation de l'expression conditionnelle
    — Construction d'un nouveau bloc pour le while
    — Initialisation du bloc While
    — Récupération et compilation de la liste d'instruction
    — Fermeture du while avec BlocGenerator : :endWhile

3.2 Fonctionnment des générateurs du code intermédiaire

Les générateurs de code IR sont des classes outils permettant la génération de code. Ces classes contiennent uniquement des fonctions statiques.

3.2.1 Fonctionnement du NameBuilder

L'implémentation des méthodes de cette classe nécéssite la définition de contracts de nommage, permettant d'éviter toutes ambiguté entre les viariables qui seront générées. Ce composant est utilisé par les autres classes de la famille de génération.
Ces fonction :

1. **buildFunctionName** : Construit un nom pour une fonction normale.
   — **Type de retour** : string

— **Paramètres** :

+ string : nom de la classe
+ string : nom de la fonction
+ string : type de retour
+ vector<string> : liste des types des paramètres

— **Indications** :

2. **buildConstructorName** : Construit le nom des fonctions permettant d'obtenir de des instances d'un objet.
   — **Type de retour** : string
   — **Paramètres** :
   + string : nom de la classe
   + vector<string> : liste des types des paramètres
   — **Indications** :
3. **buildSubConstructorName** : construit le nom des fonctions permettant d'initialiser les nouvelles instances d'objets de des instances d'un objet.
   — **Type de retour** : string
   — **Paramètres** :
   + string : nom de la classe
   + vector<string> : liste des types des paramètres
   — **Indications** :
4. **buildFunctionIndexName** : construit le nom d'un index de fonction
   — **Type de retour** : string
   — **Paramètres** :
   + string : nom de la fonction bien contruit par le name builder
   — **Indications** :
5. **buildAdHocTableName** : construit le nom d'une table adHoc
   — **Type de retour** : string
   — **Paramètres** :
   + string : nom de la class static
   + string : nom de la class dynamique
   — **Indications** :
6. **buildStaticVariableName** : construit le nom de la variable globale représentant une variable statique.
   — **Type de retour** : string
   — **Paramètres** :
   + string : classe de la variable statique
   + string : nom de la variable
   — **Indications** :

7. **buildAttributIndexName** : construit le nom de la variable globale représentant l'index d'un attribut.
   — **Type de retour** : string
   — **Paramètres** :
      + string : nom de la classe
      + string : nom de la variable
   — **Indications** : ? ? ?
8. **buildClassTypeName** : construit le nom de la classe pour le module
   — **Type de retour** : string
   — **Paramètres** :
      + string : nom de la classe
   — **Indications** : Exemple : A devient KAWA_CLASS_A
9. **buildClassStructTypeName** : construit le nom de la structure repésentant les membres d'une classe
   — **Type de retour** : string
   — **Paramètres** :
      + string : nom de la classe
10. **LLVMTypeToStr** : construit le nom d'un type à partir d'un type LLVM
   — **Type de retour** : string
   — **Paramètres** :
      + LLVM : :Type* : type llvm à transformer
   — **Indications** : float -> KawaFloat, i32 -> kawaInt, KAWA_CLASS_A -> A
11. **strToKawaType** : construit le nom pleinement qualifié d'un type pour le compilateur kawa
   — **Type de retour** : string
   — **Paramètres** :
      + string : nom du type à transformer
   — **Indications** : int -> KawaInt, float -> KawaFloat

3.2.2 Fonctionnement du TypeGenerator

Ce composant permet de creer et d'obtenir des objets de typages d'un module llvm. Les types primitifs bool, int, char, float, double et string et tableaux représentant respectivement les types llvm i8, i32, i8, float, double, i8*, [nb x type]. On les obtient grâçe à la classe Type de l'api de LLVM. Le structure principale de la classes est composée d'une référence vers un sous type contenant les attributs de la classe et une référence vers un tableau adHoc. Les méthodes et attributs statics sont quant à eux représentés par des variables globales.La génération d'une classe entrainera la génération de variables globales représentant les indexes des attributs dans la structure. Enfin le nommage de ces variables globales sera géré par le NameBuilder. (Voir l'annexe pour un exemple)
Ces Fonctions :

1. **createClassType** : cree une structure représentant une classe
   - **Type de retour** : LLVM : :Structuretype*
   - **Paramètres** :
     + string : nom de la classe
     + vector<string> : liste des noms de attributs
     + vector<string> : liste des noms des types
     + vector<bool> : attributs statiques ou non
   - **Indications** : (Voir l'annexe pour un exemple)
2. **strToLLVMType** : retourne un type LLVM associé à un nom. Un type opaque est crée sinon la structrure n'existe pas.
   - **Type de retour** : LLVM : :Structuretype*
   - **Paramètres** :
     + LLVM : :Module : module dans lequel sera fait l'operation
     + string : nom du type
   - **Indications** : le nom du devra être un nom produit par le NameBuilder

3.2.3 Fonctionnement du PrimitiveCreator

Ce composant permet de creer des objets LLVM représentant des valeurs primitives. Les classes LLVM à utiliser sont : ConstantInt, ConstantFP, ConstantDataArray (Voir l'annexe pour un exemple)

1. **create** : Cree un objet LLVM représentant une valeur primitive.
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + string|float|double|int|char : valeur primitive
     + LLVM : :Context : context de création de l'objet
   - **Indications** : la fonction est à surchargée pour chaque cas

3.2.4 Fonctionnement du PrimitiveConverteur

Ce composant à pour role de creer les opérations de conversion d'un type vers l'autre.Le LLVM IR code ne supporte nativement les opérations binaires intertypées. Il faudra se servir de l'objet IRBuilder pour faciliter l'implémentation de la conversion. (Voir l'annexe pour un exemple)

1. **convertFormTo** : Convertie une valeur primitive d'un type à l'autre
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Type* : type de départ
     + LLVM : :Type* : type cible
     + LLVM : :Value* : valeur convertir
     + LLVM : :BasiBlock* : bloc d'instruction dans lequel s'effectue la conversion.
   - **Indications** :

3.2.5 Fonctionnement du PrimitiveBinaryOperation

Ce composant permet la génération des opérations primitives.L'utilisation d'un IRBuilder est conseillée. (Voir l'annexe pour un exemple)

1. **create[Add|Mul|Sub|Div|Mod]** : cree un[e] [addition|produit|soustraction|division|reste]
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Type* : type vers lequel sera convertie le résultat
     + LLVM : :Value* : opérande gauche
     + LLVM : :Value* : opérande droite
     + LLVM : :BasicBlock* : bloc dans lequel effectuer l'opération
   - **Indications** : le bloc ne devrait jamais être null
2. **create[And|Xor|Not|Eq|Or|Sup|Inf|SOE|IOE]** : cree une opération de comparaison
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Value* : opérande gauche
     + LLVM : :Value* : opérande droite (sauf pour Not)
     + LLVM : :BasicBlock* : bloc dans lequel effectuer l'opération
   - **Indications** : la valeur llvm retournée est toujours un entier llvm. SOE = Superieur ou égal. IOE = Inferieur ou égal.
3. **valToBool** : transforme une valeur en un boolean
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Value* : valeur à transformer
     + LLVM : :BasicBlock* : bloc dans lequel effectuer l'opération
   - **Indications** : false si égal à 0 ou null, sinon true

3.2.6 Fonctionnement du FunctionGenerator

Ce composant permet de declarer et d'avoir accès au composant d'un module. Il utilise Namebuilder pour generer le nom du code de la fonction dans le code IR. La génération d'une méthode non statique ajoute un paramètre supplémentaire représentant l'objet ẗhis: La génération d'un constructeur génére deux fonction llvm. L'une créant et renvoyant une instance de l'objet creer, et une function appelée par la précédente permettant l'initialisation de l'instance. Cette fonction prendra un paramètre supplémentaire représentant l'objet à initialiser. De plus il permet aussi de déclarer et d'obtenir ades fonctions des librairies présentent sur le système comme, strlen, strcpy, strcat..etc (Voir l'annexe pour un exemple)

1. **createFunction** : Cree une fonction dans un module llvm.
   - **Type de retour** : LLVM : :Function
   - **Paramètres** :
     + Module module : module dans lequel est contruit la fonction
     + bool isStatic : la fonction est statique
     + string className : le nom de la classe

+ string ret_type : type de retour de la fonction
+ vector<string> arg_types : liste des types des paramètres
+ vector<string> arg_names : liste des noms des paramètres

— **Indications** : Le nom de la fonction produite est le nom de la fonction produite est générée avec le **NameBuilder**

2. **createContructor** : cree une fonction d'instanciation de classe.
   — **Type de retour** : LLVM : :Function
   — **Paramètres** :
   + Module module : module dans lequel est contruit la fonction
   + string className : nom de la classe
   + vector<string> args_types : types des arguments
   + vector<string> args_names : nom des arguments
   — **Indications** : Le nom de la fonction produite est le nom de la fonction produite est générée avec le **NameBuilder**. La construction d'un objet est faîte par deux fonctions différentes. L'une automatiquement générée, permettant l'allocaction d'un nouvel objet, et une fonction d'initilisation de l'objet. Le corps de cette dernère est à définir.
3. **getFunction** : Retourne une function du module à partir de son nom dans le module, **null** sinon
   — **Type de retour** : LLVM : :Function
   — **Paramètres** :
   + Module module : module dans lequel est contruit la fonction
   + string className : nom pleinement qualifié de la fonction à chercher.
   — **Indications** : le nom de la fonction à retourner devra être contruite avec le **NameBuilder**
4. **getOrCreateMainFunction** : crée ou retourne la fonction **main**
   — **Type de retour** : LLVM : :Function
   — **Paramètres** :
   + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
   — **Indications** : Il s'agit de la fonction de point d'entrer du programme.
5. **getOrCreatePutsFunction** : crée ou retourne la fonction **puts**
   — **Type de retour** : LLVM : :Function
   — **Paramètres** :
   + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
   — **Indications** : La fonction permet de l'affichage de chaînes de caractères. La fonction est implémentée par des librairies locales et chargée par clang.
6. **getOrCreateStrlenFunction** : crée ou retourne la fonction **strlen**
   — **Type de retour** : LLVM : :Function
   — **Paramètres** :
   + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
   — **Indications** : La fonction permet de calculer la longueur d'une chaine de caractère. La fonction est implémentée par des librairies locales et chargée par clang.
7. **getOrCreateStrcatFunction** : crée ou retourne la fonction **strcat**

— **Type de retour** : LLVM : :Function
— **Paramètres** :
+ LLVM : :Module : le module dans lequel creer ou obtenir la fonction
— **Indications** : La fonction permet de concatener deux chaines de caractères. La fonction est implémentée par des librairies locales et chargée par clang.

8. **setFunctionBody** : permet de définir le corps d'une fonction
— **Type de retour** : void
— **Paramètres** :
+ LLVM : :Module : le module dans lequel creer ou obtenir la fonction
+ vector<LLVM : :BasicBlock*> : La liste des blocs llvm comportant les instructions d'une fonction
— **Indications** : cette fonction s'appliquera aussi aux constructeurs.

3.2.7 Fonctionnement du CallGenerator

Ce composant permet de génerer les appels aux méthodes et au chargement d'attributs. Le composant fonctionne de la manière suivante.

— Les appels aux membres static sont directes. Une résolution de nom est suffisante pour obtenir l'adresse de l'objet cible. Les méthodes et les attributs static sont des variables globales dans le module, et repérables par leur nom.
— Les attributs et méthodes non statics sont contenus des structures et tableaux représentant les classe et les tables adHoc permettant d'implementer le principe de polymorphisme. Leur position dans la structure doivent pouvoir être determiner afin de les charger. Pour cela il faut employer la méthode getIndexOfMember de **GloabalGenerator**.
— Les appels aux constructeurs d'objets sont traités comme des appels statics.
— (A completer)

(Voir l'annexe pour un exemple)

1. **callStaticMethode** : cree un appel static sur une méthode
— **Type de retour** : ? ? ? ?
— **Paramètres** :
+ ? ? ?
— **Indications** : ? ? ?
2. **callMethode** : cree un appel non static sur une méthode
— **Type de retour** : ? ? ? ?
— **Paramètres** :
+ ? ? ?
— **Indications** : ? ? ?
3. **loadStaticAttribut** : charge un attribut static
— **Type de retour** : ? ? ? ?
— **Paramètres** :
+ ? ? ?
— **Indications** : ? ? ?
4. **loadAttribut** : charge un attribut non static
— **Type de retour** : ? ? ? ?
— **Paramètres** :
+ ? ? ?
— **Indications** : ? ? ?

3.2.8 Fonctionnement du GlobalVariableGenerator

Ce composant permet la génération des variables globales nécéssaire à la sémentique du programme. Il permet de declarer des variables statics, les tables AdHoc, et les indexes de membres. La compilation en mode Monilithique ét partagé divergera à ce niveau au niveau de atributs des variables. (Voir l'annexe pour un exemple)
Ses fonctions :

1. **getOrCreateStaticAttribut** : cree ou retourne un attribut static
   - **Type de retour** : ? ? ? ?
   - **Paramètres** :
     + LLVM : :Module : module llvm dans lequel effectuer l'opération
     + string : nom de la classe
     + string : nom de l'attribut
     + string : nom du type de l'attribut
   - **Indications** :
2. **createAdHocTable** : cree une table adHoc référençant les fonction correspendant au couple static-dynamique de deux classes
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Module : module dans lequel creer le tableau
     + string : nom de la classe statique
     + string : nom de la classe dynamique
     + vector<LLVM : :Function *> : liste des fonctions de la table
   - **Indications** : le nom des classes doivent avoir été générée avec le NameBuilder. A l'issue de l'opération, chacune des fonction se verra affecté un index par rapport à sa position dans la liste. Il est donc nécéssaire que la liste des fonctions soient ordonnées.
3. **getAdHocTable** : permet d'obtenir une table adHoc
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Module : module dans lequel creer le tableau
     + string : nom de la classe statique
     + string : nom de la classe dynamique
   - **Indications** : le nom des classes doivent avoir été généré avec le NameBuilder.
4. **getAdHocTable** : permet d'obtenir une table adHoc
   - **Type de retour** : LLVM : :Value*
   - **Paramètres** :
     + LLVM : :Module : module dans lequel creer le tableau
     + string : nom de la classe statique
     + string : nom de la classe dynamique
   - **Indications** : le nom des classes doivent avoir été généré avec le NameBuilder.
5. **createIndexOfMember** : permet de creer un index pour un élément global dans un module llvm

— **Type de retour** : LLVM : :Value*
— **Paramètres** :
+ LLVM : :Module : module dans lequel creer le tableau
+ string : nom de l'élément
+ int : index
— **Indications** : le nom de l'élément doit avoir été générée avec le NameBuilder.

6. **getIndexOfMember** : permet d'obtenir un objet llvm représentant un entier correspondant à l'index
— **Type de retour** : LLVM : :Value*
— **Paramètres** :
+ LLVM : :Module : module dans lequel creer le tableau
+ string : nom de l'élément
— **Indications** : le nom de l'élément doit avoir été générée avec le NameBuilder.

3.2.9 Fonctionnnement du BlockGenerator

Ce composant permet d'ajancer les bloc d'instruction LLVM lors de la génération du code d' instructions conditionelles.
Liste des fonctions :

1. **createBloc** : crée un bloc basic
— **Type de retour** : LLVM : :BasicBlock*
— **Paramètres** :
+ LLVM : :Context
— **Indications** : ? ? ?
2. **create[IF|ELSE|For|While]** : crée un bloc [IF|ELSE|For|While]
— **Type de retour** : LLVM : :BasicBlock*
— **Paramètres** :
+ LLVM : :Context
3. **init[IF|ELSE|For|While]** : initialise un bloc [IF|ESLE|For|While]
— **Type de retour** : void
— **Paramètres** :
+ LLVM : :BasicBlock : bloc précédent
+ LLVM : :BasicBlock : bloc à initialiser
+ LLVM : :BasicBlock : bloc suivant
+ LLVM : :BasicBlock : condition de bouclage
4. **end[IF|ELSE|For|While]** : ferme un bloc [IF|ESLE|For|While] et renvoie e bloc suivant
— **Type de retour** : void
— **Paramètres** : LLVM : :
+ LLVM : :BasicBlock : bloc à fermer