**LLVM**

Introduction :

La représentation du code en LLVM est définie pour être utilisé sous trois formes différentes:

-Code Native: Code qui peut directement être exécuté par le processeur

-Byte Code : Code intermédiaire a besoin d’être converti en natif code pour pouvoir être exécuté ou alors par une vm (JVM JAVA).

-LLVM ASM : Un langage assembleur compréhensible pour l homme.

ASM LLVM:

Un module <=> Une unité de compilation.

1) Deux types d'identifiants: global et local

a)Identifiant global: Fonctions et variables globales

Commence par '@'.

b) identifiant local: noms de registre, types

Commence par '%'

2) 3 formats différents pour les identifiants:

-valeur nomme représente par une chaine de caractère (%foo, @DivisionByZero, etc)

-valeur non nommé représenté par un nombre non signé (%12, @2, %44)

-Constants

(Utiliser les valeurs non nomme permets au compilateur d aller plus vite en lui évitant d'avoir a gérer les conflits dans la table des symboles).

Note :

Pas de conflits avec les mots clés (opcode : 'add', 'bitcast', 'ret' // noms de types: 'void', 'i32') grâce aux '@' et '%'.

Exemple :

X\*8: %result = mul i32 %X, 8

alias: %mytype = type { %mytype\*, i32 }

Hello World:

; Declare the string constant as a global constant.

@.LC0 = internal constant [13 x i8] c"hello world\0A\00" ; [13 x i8]\*

; External declaration of the puts function

declare i32 @puts(i8\*) ; i32 (i8\*)\*

; Definition of main function

define i32 @main() { ; i32()\*

; Convert [13 x i8]\* to i8 \*...

%cast210 = getelementptr [13 x i8]\* @.LC0, i64 0, i64 0 ; i8\*

; Call puts function to write out the string to stdout.

call i32 @puts(i8\* %cast210) ; i32

ret i32 0

}

; Named metadata

!1 = metadata !{i32 41}

!foo = !{!1, null}

3) Les fonctions

La définition des fonctions en LLVM consiste en un mot clé "define", le type du linkage (optionnel), le style de visibilité (optionnel), une convention d appelle (optionnel), un type de retour, un attribut au type de retour(optionnel), le nom de la fonction, une liste d'argument(peut être vide)(chacun avec un attribut optionnel),un attribut a la fonction (optionnel), une section(optionnel), un alignement (optionnel), le nom d'un garbage collector(optionnel), an opening curly brace, une liste de block basique, and a closing curly brace.

La déclaration des fonctions en LLVM ressemble a la définition des fonctions a la différence prés que le mot clé est "declare" et qu’il n y a pas de curly brace et de block.

Le premier block d'une fonction est spécial de deux façons: Il est exécuté directement a l'entrée de la fonction et ne peut pas avoir de block qui le précède.

4) Type system:

Classification Types

integer i1, i2, i3, ... i8, ... i16, ... i32, ... i64, ...

floating point float, double, x86\_fp80, fp128, ppc\_fp128

first class integer, floating point, pointer, vector, structure, union, array, label, metadata.

primitive label, void, floating point, metadata.

derived array, function, pointer, structure, packed structure, union, vector, opaque.

a)Integer Type

Syntax: iN

Le nombre de bit que l entier occupera est specifier par la valeur de N

b)Floating Point Types

Type Description

float 32-bit floating point value

double 64-bit floating point value

fp128 128-bit floating point value (112-bit mantissa)

x86\_fp80 80-bit floating point value (X87)

ppc\_fp128 128-bit floating point value (two 64-bits)

c)Void Type

Syntax: void

Le type void ne représente pas de valeur et n a pas de taille.

d) Label Type

Syntax: label

Le type label représente le code label (Comme en ASM?)

e) Metadata Type

Syntax: metadata

Donnée servant à définir ou décrire une autre donnée.

f) Array Type

Syntax: [<# nbelements> x <elementtype>]

Examples:

[40 x i32] Array of 40 32-bit integer values.

[3 x [4 x i32]] 3x4 array of 32-bit integer values.

g) Function Type

Syntax: <returntype> (<parameter list>)

Examples:

i32 (i32) function taking an i32, returning an i32

float (i16, i32 \*) \* Pointer to a function that takes an i16 and a pointer to i32, returning float.

i32 (i8\*, ...) A vararg function that takes at least one pointer to i8 (char in C), which returns an integer. This is the signature for printf in LLVM.

{i32, i32} (i32) A function taking an i32, returning a structure containing two i32 values

h) Structure Type

Syntax: { <type list> }

Examples:

{ i32, i32, i32 } A triple of three i32 values

Note:

Structures in memory are accessed using 'load' and 'store' by getting a pointer to a field with the 'getelementptr' instruction. Structures in registers are accessed using the 'extractvalue' and 'insertvalue' instructions.

i) Packed Structure Type

Structures are accessed using 'load and 'store' by getting a pointer to a field with the 'getelementptr' instruction.

Syntax:< { <type list> } >

Examples:

< { i32, i32, i32 } > A triple of three i32 values

< { float, i32 (i32)\* } > A pair, where the first element is a float and the second element is a pointer to a function that takes an i32, returning an i32.

j) Union Type

Syntax: union { <type list> }

Examples:

union { i32, i32\*, float } A union of three types: an i32, a pointer to an i32, and a float.

union { float, i32 (i32) \* } A union, where the first element is a float and the second element is a pointer to a function that takes an i32, returning an i32.

k) Pointer Type

Note that LLVM does not permit pointers to void (void\*) nor does it permit pointers to labels

(label\*). Use i8\* instead.

Syntax:<type> \*

Examples:

[4 x i32]\* A pointer to array of four i32 values.

i32 (i32\*) \* A pointer to a function that takes an i32\*, returning an i32.

i32 addrspace(5)\* A pointer to an i32 value that resides in address space #5.

l) Vector Type

Syntax: < <# elements> x <elementtype> >

Examples:

<4 x i32> Vector of 4 32-bit integer values.

<8 x float> Vector of 8 32-bit floating-point values.

<2 x i64> Vector of 2 64-bit integer values.

m) Opaque Types

Opaque types are used to represent unknown types in the system. This corresponds (for example) to the C notion of a forward declared structure type. In LLVM, opaque types can eventually be resolved to any type (not just a structure type).

Syntax: opaque

n) Type Up-references

An "up reference" allows you to refer to a lexically enclosing type without requiring it to have a name. For instance, a structure declaration may contain a pointer to any of the types it is lexically a member of. Example of up references (with their equivalent as named type declarations) include:

{ \2 \* } %x = type { %x\* }

{ \2 }\* %y = type { %y }\*

\1\* %z = type %z\*

An up reference is needed by the asmprinter for printing out cyclic types when there is no declared name for a type in the cycle. Because the asmprinter does not want to print out an infinite type string, it needs a syntax to handle recursive types that have no names (all names are optional in llvm IR).

Syntax: \<level>

The level is the count of the lexical type that is being referred to.

Examples:

\1\* Self-referential pointer.

{ { \3\*, i8 }, i32 } Recursive structure where the upref refers to the out-most structure.