Compil ou Face

Notions de compilation pour le reverseur

par *Serge*☘ et *Juan*⚽

☘ sguelton@quarkslab.com

⚽ jmmartinez@quarkslab.com

man sguelton

- Ingénieur R&D à Quarkslab, spécialisé en compilation (Python, LLVM)
- Chercheur associé à Télécom Bretagne

apropos jmmartinez

- Ingénieur R&D à Quarkslab, spécialisé en compilation

Ce Cours (il est long)

- Connaitre une chaîne de compilation et découvire Clang/LLVM
- Comprendre quelques transformations et analyse

Cours et TP entrelacés

Intérêt pour le reverser

- Mieux comprendre le code généré
- Billes de compréhension pour écrire des outils d'analyse
- Culture G

Cherchez l'intru

Lequel de ces outils n'embarque pas de compilateur ?

gcc, clang, tex, sh, javac, firefox, perl, python

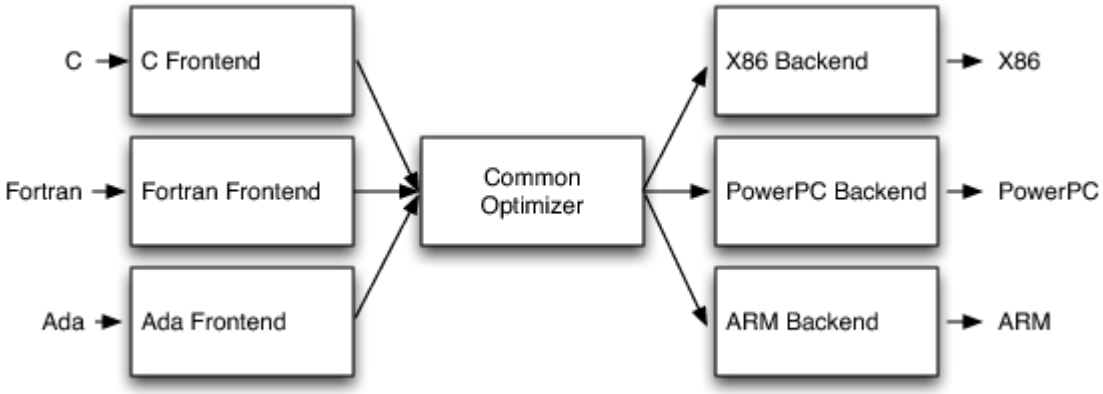
Galerie des horreurs !0

- (O)TCC : http://fr.wikipedia.org/wiki/Tiny_C_Compiler (http://fr.wikipedia.org/wiki/Tiny_C_Compiler)
- Emscripten : http://emscripten.org (http://emscripten.org)
- CompCert : http://compcert.inria.fr/ (http://compcert.inria.fr/)
- NoWeb : http://www.cs.tufts.edu/~nr/noweb/ (http://www.cs.tufts.edu/~nr/noweb/)

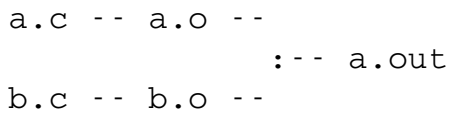
Galerie des horreurs !1

- Astrée : http://www.astree.ens.fr/ (http://www.astree.ens.fr/)
- Splint : http://www.splint.org/ (http://www.splint.org/)
- Sparse : https://sparse.wiki.kernel.org/index.php/Main_Page (https://sparse.wiki.kernel.org/index.php/Main_Page)

Compilateur trois phases


C
C Frontend
Fortran
Fortran Frontend
Ada
Ada Frontend
Common
Optimizer
X86 Backend
X86
PowerPC Backend
PowerPC
ARM Backend
ARM

Chaîne de compilation classique C



In [1]:
```
%%file a.c
#include <stdio.h>
void greet(char const* who) {
    printf("Hello %s!\n", who);
}
```

```
Overwriting a.c
```

In [2]:
```
%%file b.c
extern void greet(char const* who);
int main(int argc, char const* argv[]) {
    if(argc == 1) greet("world");
    else greet(argv[1]);
    return 0;
}
```

```
Overwriting b.c
```

In [3]:
```
%%!
clang a.c b.c
./a.out $USER
```

Out[3]:
```
['Hello serge!']
```

Pas à Pas : Préprocesseur

a.k.a. « le sed du pauvre »

In [4]:
```
!clang -E a.c | wc -l
```

```
740
```

In [5]:
```
!clang -E a.c | head -n 10
```

```
# 1 "a.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 317 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "a.c" 2
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
```

Pas à Pas : La Représentation Interne

Généralement pas exposée à l'utilisateur...

In [6]:
```
!clang -S -emit-llvm a.c
!head -n 16 a.ll
```

```
; ModuleID = 'a.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [11 x i8] c"Hello %s!\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @greet(i8* %who) #0 {
  %1 = alloca i8*, align 8
  store i8* %who, i8** %1, align 8
  %2 = load i8*, i8** %1, align 8
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8], [11 x
i8]* @.str, i32 0, i32 0), i8* %2)
  ret void
}

declare i32 @printf(i8*, ...) #1
```

Pas à Pas : La transformation de RI (olé !)

Ce qui se cache derrière `-O2` et consorts

In [7]:
```
!opt-3.8 -mem2reg a.ll -S | head -n 10
```

```
; ModuleID = 'a.ll'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [11 x i8] c"Hello %s!\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @greet(i8* %who) #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8], [11 x
i8]* @.str, i32 0, i32 0), i8* %who)
  ret void
```

Pas à Pas : La génération de code assembleur

Avec la bonne variante syntaxique...

In [8]:
```
!llc-3.8 a.ll --x86-asm-syntax=intel
!head -n 23 a.s | tail -n 10
```

```
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp2:
        .cfi_def_cfa_register rbp
        sub     rsp, 16
        mov     rcx, rdi
        mov     qword ptr [rbp - 8], rcx
        mov     edi, .L.str
        xor     eax, eax
        mov     rsi, rcx
```

Ou plus simplement :

In [9]:
```
!clang -S  a.c #-masm=intel if needed
```

Pas à Pas : L'assemblage

Génération de code objet, ou `.o`

In [10]:
```
!as a.s -o a.o
!file a.o
```

```
a.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Pas à Pas : L'édition de liens

Le *linker*, les libs statiques, les lib dynamiques…

In [11]:
```
!clang -c b.c # pour avoir le deuxième code objet
```

In [12]:
```
!nm b.o | grep greet
```

```
                 U greet
```

In [13]:
```
!nm a.o | grep greet
!nm a.o | grep printf
```

```
0000000000000000 T greet
                 U printf
```

In [14]:
```
!ldd /bin/ls | grep libc.so
```

```
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1b714fb000)
```

In [15]:
```
!nm /lib/x86_64-linux-gnu/libc.so.6 | grep printf
```

```
nm: /lib/x86_64-linux-gnu/libc.so.6: no symbols
```

In [16]:
```
!file -L /lib/x86_64-linux-gnu/libc.so.6
```

```
/lib/x86_64-linux-gnu/libc.so.6: ELF 64-bit LSB shared object, x86-64, version 1
 (GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, Build
ID[sha1]=2246ba050897f1d98034a7ca4b7ec06b594a373d, for GNU/Linux 2.6.32, strippe
d
```

In [17]:
```
!readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep ' printf'
```

```
   602: 000000000004f160   161 FUNC    GLOBAL DEFAULT   13 printf@@GLIBC_2.2.5
  1499: 000000000004f0b0    31 FUNC    GLOBAL DEFAULT   13 printf_size_info@@GLI
BC_2.2.5
  1911: 000000000004e8c0  2020 FUNC    GLOBAL DEFAULT   13 printf_size@@GLIBC_2.
2.5
```

Pas à Pas : l'exécutable

Pour pondre un joli petit `a.out` tout mignon

In [18]:
```
! ld a.o b.o && ./a.out
```

```
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
a.o: In function `greet':
a.c:(.text+0x20): undefined reference to `printf'
```

In [19]:
```
! ld a.o b.o -lc && file a.out && ./a.out
```

```
ld: warning: cannot find entry symbol _start; defaulting to 00000000004002a0
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib/ld64.so.1, not stripped
/bin/sh: 1: ./a.out: not found
```

In [20]:
```
! clang -v a.o b.o 2>&1 | grep ld
```

```
 "/usr/bin/ld" --hash-style=both --build-id --eh-frame-hdr -m elf_x86_64 -dynami
c-linker /lib64/ld-linux-x86-64.so.2 -o a.out /usr/bin/../lib/gcc/x86_64-linux-g
nu/6.3.0/../../../x86_64-linux-gnu/crt1.o /usr/bin/../lib/gcc/x86_64-linux-gnu/6
.3.0/../../../x86_64-linux-gnu/crti.o /usr/bin/../lib/gcc/x86_64-linux-gnu/6.3.0
/crtbegin.o -L/usr/bin/../lib/gcc/x86_64-linux-gnu/6.3.0 -L/usr/bin/../lib/gcc/x
86_64-linux-gnu/6.3.0/../../../x86_64-linux-gnu -L/lib/x86_64-linux-gnu -L/lib/.
./lib64 -L/usr/lib/x86_64-linux-gnu -L/usr/bin/../lib/gcc/x86_64-linux-gnu/6.3.0
/../../.. -L/usr/lib/llvm-3.8/bin/../lib -L/lib -L/usr/lib a.o b.o -lgcc --as-ne
eded -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/bi
n/../lib/gcc/x86_64-linux-gnu/6.3.0/crtend.o /usr/bin/../lib/gcc/x86_64-linux-gn
u/6.3.0/../../../x86_64-linux-gnu/crtn.o
```

In [21]:
```
! ./a.out 1
```

```
Hello 1!
```

Comprendre l'*Abstract Syntax Tree*

Jouons avec Python et son AST, plus facile que celui de C++

In [22]:
```
import ast
tree = ast.parse("print(1)")
print(tree)
```

```
<_ast.Module object at 0x7fc57063f750>
```

In [23]:
```
ast.dump(tree)
```

Out[23]:
```
'Module(body=[Print(dest=None, values=[Num(n=1)], nl=True)])'
```

In [24]:
```
import astdump
astdump.indented(tree)
```

```
Module
  Print
    Num
```

Compilation de l'AST

Passage d'une représentation proche du langage à une représentation proche de l'interpréteur.

In [25]:
```
code = compile(tree, '<>', 'exec')
```

In [26]:
```
eval(code)
```

```
1
```

Inspection du bytecode

CPython → Interpréteur à pile

In [27]:
```
import dis
dis.dis(code)
```

```
  1           0 LOAD_CONST               0 (1)
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               1 (None)
              8 RETURN_VALUE
```

(Aparté

D'après vous, quels sont les avantages et inconvénients d'une interpréteur à pile par rapport à un interpréteur à registre?

Interpréteur à pile

Facile de conception, peu d'optimisations

Interpréteur à registre

Plus complexe (et pas seulement pour l'allocation de registre) mais permet de modéliser plus d'optimisations

Game of Stack

Écrire un interpréteur qui comprend les instructions suivantes :

- `PUSH <integer>` qui ajoute `<integer>` au dessus de la pile
- `DUP` qui duplique le dessus de la pile
- `ADD` qui enlève les deux premiers éléments de la pile et ajoute `S[0] + S[1]` au dessus de la pile
- `MUL` qui enlève les deux premiers éléments de la pile et ajoute `S[0] * S[1]` au dessus de la pile
- `READ` qui lit un entier sur `stdin` et l'ajoute au dessus de la pile
- `WRITE` qui dépile le premier élément de la pile et l'affiche sudr `stdout`

Par exemple :

```
0 READ
1 DUP
2 ADD
3 WRITE
```

Introduisons maintenant une optmisation (de ouf !). Les deux séquences suivantes sont équivalentes :

```
PUSH 2
MUL
```

et

```
DUP
ADD
```

Ajoutez à votre interpréteur une passe qui effectue de qui s'avère être une *peephole optimisation* en transformant l'une en l'autre.

Et maintenant, introduisez l'instruction

```
JMP <INDEX>
```

Qui saute directement à la <INDEX> ième instruction. Quel impact cela a-t-il sur l'optimisation précédente ?

Enfin, introduisez l'instruction

```
JMP
```

Similaire à la précédente, mais qui lit la valeur de `<INDEX>` sur la pile (en la dépilant). Quel impact cela a-t-il sur l'optimisation précédente ?

Fin de l'aparté)

Continuons à jouer avec l'AST

L'AST Python peut être parcouru grâce à un **visiteur** (qui n'est pas né d'hier)

À lire : https://docs.python.org/3/library/ast.html (https://docs.python.org/3/library/ast.html)

In [29]:
```
class VisitIntegers(ast.NodeVisitor):
    def visit_Num(self, node):
        if isinstance(node.n, int):
            print(node.n)

VisitIntegers().visit(tree)
```

```
1
```

Exo

Écrivez un visiteur qui va trouver tous les appels à la fonction `open`

Pourquoi est-ce en fait impossible en analyse statique ?

En instrumentant

Une sorte d'analyse dynamique ?

In [30]:
```
import __builtin__
real_open = __builtin__.open
def myopen(*args, **kwargs):
    print("hooked:", args, kwargs)
    return real_open(*args, **kwargs)
__builtin__.open = myopen
open("/dev/null")
```

```
('hooked:', ('/dev/null',), {})
```

Out[30]:
```
<open file '/dev/null', mode 'r' at 0x7fc57795e6f0>
```

In [31]:
```
__builtin__.open = real_open
```

Conclusion

- Domaine très vaste dont on a à peine éffleuré la surface
- Présent dans le quotidien de tout informaticien
- Ouverture / question du jour : nouveau langage ou eDSL ?