你所不知道的 C 語言：編譯器原理和案例分析

直播錄影

如何打造一個具體而微的 C 語言編譯器？

- 700 行系列 與隨感
  - 整整 20 年前，我在苗栗老家用著緩慢的 Pentium 電腦搭配硬碟，在 Slackware Linux 用舊版 gcc 編譯和安裝 gcc-2.8.0 (1998 年 1 月 14 日釋出) 和 gcc-2.8.1 (同年 3 月釋出)。
  - gcc 8.2.0 版於 2018 年 7 月 26 日釋出
  - 你可以想像今日是工業編譯器技術標準的 gcc，在整整 20 年前，原始碼打包後僅 8 MB，全部程式碼不到 53 萬行 (C 語言佔其中近 33 萬行)，而且測試程式碼才 241 行嗎？對比最新的 gcc-8.2.0，用 tar + gzip 壓縮打包後，後者佔了 109 MB，原始程式碼的規模已達到約 752 萬行
  - 20 年的時間，一個專案從「免費又好用」變成「帶領產業變革的標準」，原始程式碼膨漲 14 倍 (過程中移除 gcj 一類的子專案，實際上要大得多)
  - gcc-2.8.0 程式碼中，可見到 Richard Stallman 修改程式碼的蹤跡 (參見 ChangeLog 檔案)。

AMaCC 是由台灣國立成功大學師生開發的 self-compiling 的 C 語言編譯器，可產生 Arm 架構的執行檔 (ELF 格式，運作於 GNU/Linux)、也支援 just-in-time (JIT) 編譯和執行，原始程式碼還不到 1500

行，在本次講座中，我們揭開 AMaCC 背後的原理和實作議題。

預期會接觸到 IR (Intermediate representation), dynamic linking, relocation, symbol table, parsing tree, language frontend, Arm 指令編碼和 ABI 等等。

在 COSCUP 2018 的演講 用1500 行建構可自我編譯的 C 編譯器

- [ ] AMaCC 產生出來的執行檔

> $ file amacc
> amacc: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0, BuildID[sha1]=25cce9ff5ec975579eed9f890511f1c2bd64610d, with debug_info, not stripped

- [ ] 程式碼行數

```
$ cloc amacc.c
       1 text file.
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                1            122            216           1473
-------------------------------------------------------------------------------
```

編譯器和軟體工業強度息息相關

依據 IBM 的簡報指出: [source]

- 1960 年代的 F-4 Phantom 戰鬥機: 僅有 8% 需要軟體控制 (組合語言開發);

- 1970 年代的 F-16 Falcon 戰鬥機: 45% 需要軟體控制 (JOVIAL 程式語言開發，類似 ALGOL，主要針對嵌入式系統用於編寫軍用飛機電子系統);
- 1980 年代的 F-22 Raptor 戰鬥機: 80% 規範仰賴軟體 (Ada83 程式語言開發)，程式碼規模為 170 萬行;
- 2000 年之後的 F-35 Lightning 戰鬥機: 2400 萬行程式碼，用 C/C++ 撰寫;

自動駕駛車的軟體規模預計會百倍於飛行控制系統，而開發階段所必須投入發展的軟體驗證環境和方法對開發都至關重大。韓國的 Hyundai Mobis 計畫與 KAIST 合作開發 MAIST，著眼於後續自駕車的軟體性能及成本的影響非常龐大。

IEEE 1012: 安全相關應用軟體開發
IEC 61508-3: 以每小時危險失效率 (PFH) 或失效危險機率 (PFD) 的形式訂立許多可靠度要求

形式化驗證

背景知識

- 編譯器和最佳化原理篇
- 函式呼叫篇
- 動態連結器和執行時期篇
- 虛擬機器設計與實作

C 程式的解析和語意

- descent 點評幾本編譯器設計書籍 (中文為主)
- 手把手教你建構 C 語言編譯器
- c4 是很好的切入點，原作者 Robert Swierczek 還又另一個更完整的 C 編譯器實作

- AMaCC 在 Robert Swierczek 的基礎上，額外實作 C 語言的 struct, switch-case, for, C-style comment 支援，並且重寫了 IR 執行程式碼，得以輸出合法 GNU/Linux ELF 執行檔 (支援 armhf ABI) 和 JIT 編譯
- 逐步開發編譯器

IR (Intermediate representation)

C 語言程式解析後，會轉為 opcode，可透過 interpreter 執行，或者:

1. JIT 編譯程式碼，透過 mmap 系統呼叫和指定 function pointer 去執行
2. 透過 codegen 搭配 ELF 輸出，過程中需要進行 relocation，以便得知執行時期的地址計算

Interpreter, Compiler, JIT from scratch

AMaCC 的 `-s` 命令列選項可輸出 IR:

```
qemu-arm -L /usr/arm-linux-gnueabihf ./amacc -s tests/hello.c
1: #include <stdio.h>
2:
3: int main()
4: {
5:     printf("hello, world\n");
    ENT  0
    IMM  -161644536
    PSH
    PRTF
    ADJ  1
6:     return 0;
    IMM  0
    LEV
```

```
7: }
    LEV
```

針對 Arm 架構的 JIT 執行:

```
#include <sys/mman.h>
#include <stdlib.h>
#include <stdio.h>

int main(int ac, char **av)
{
    char *jitmem;
    int *je, var;

    jitmem = mmap(0, 256, 7, 0x22, -1, 0);
    je = (int *)jitmem;
    *je++ = 0xe59f000c; // ldr r0, [pc, #12]
    *je++ = 0xe5901000; // ldr r1, [r0]
    *je++ = 0xe2811009; // add r1, r1, #9
    *je++ = 0xe5801000; // str r1, [r0]
    *je++ = 0xe1a0f00e; // mov pc, lr
    *je   = (int)&var;
    __clear_cache(jitmem, je);

    var = ac;
    bsearch(&av, av, 1, 1, (void*) jitmem);
    printf("ac = %d, var = %d\n", ac, var);
    return 0;
}
```

搭配閱讀 How to JIT - an introduction

JIT compiler 的效益: How to write a very simple JIT compiler

程式語言設計和編譯器考量

- 影片: Brian Kernighan on successful language design