資工所碩士班一年級 陳冠軒 112062623

1. How to run my program

先在 **Makefile** 中將 LLVM_CONFIG 的路徑設定好，以及設定要分析的 C code(ex:test1.c、test.c)，然後進入我的 hw1 directories，資料夾結構如下圖 1，在此 directories 下執行 make run 即可 run 我的 pass，執行的結果如下圖 2，執行結果會在 terminal 顯示。

```
ubuntu@ubuntu-MS-7A38:~/advanced_compiler/hw1$ tree
.
├── hw1.cpp
├── Makefile
├── test1.c
└── test2.c

0 directories, 4 files
```

圖表 1

```
ubuntu@ubuntu-MS-7A38:~/advanced_compiler/hw1$ make run
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/clang -Xclang -disable-O0-optnone -fno-discard-value-names -S -emit-llvm -o test1.ll test1.c
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/clang -shared -o hw1.so hw1.cpp `/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --cppflags` -fPIC -fno-rtti `/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --ldflags`
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/opt -disable-output -load-pass-plugin=./hw1.so -passes=hw1 test1.ll
====Flow Dependency====
(i=15, i=19)
A: S1 -----> S2
(i=14, i=18)
A: S1 -----> S2
(i=13, i=17)
A: S1 -----> S2
(i=12, i=16)
A: S1 -----> S2
(i=11, i=15)
A: S1 -----> S2
(i=10, i=14)
A: S1 -----> S2
(i=9, i=13)
A: S1 -----> S2
(i=8, i=12)
A: S1 -----> S2
(i=7, i=11)
A: S1 -----> S2
(i=6, i=10)
A: S1 -----> S2
(i=5, i=9)
A: S1 -----> S2
(i=4, i=8)
A: S1 -----> S2
====Anti-Dependency====
====Output Dependency====
```

圖表 2

2. The case that I can handle

在我自己的測試中助教給的 test1.c 跟 test2.c 都有成功分析出三種 dependency，我寫的 pass，在迴圈中有 A[a*i+b]= B[a*i+b]此種 pattern 的 statement 都可以分析，因為我是用陣列資料結構紀錄 statement 的資訊，目前陣列都是長度為 10，所以可以記錄最多 10 條的 statement，程式會自動將每條 statement 之間的 output/flow/anti dependency 找出。

3. Experiment report

利用 llvm 提供的 api 將 IR 中迴圈的數值、變數的名稱、index 的加減乘除參數取出，我用到的 api 整理如下

Api name	**用途簡述**
getName()	取得 IR 變數名稱
getOperand()	取得 IR 中的運算元
dyn_cast<Inst_type>()	用於識別指令類型或轉換型別
for (BasicBlock &BB : F)	迭代 basic block
for (Instruction &I : BB)	迭代 IR instruction
getSExtValue()	取出 IR 變數中的整數部分
getOpcodeName()	得到 IR intruction 的指令名稱

我的方法是一條一條迭代 IR 指令，迭代過程中取出程式資訊，包含**迴圈起始值**、每個 **statement 的變數名稱**、**變數 index 值**，取出後存到陣列中然後利用這些數值分析 IR。

迭代 instruction 過程中如果遇到 store 的話就當成是一個 statement 的結束，要開始記錄下一個 statement 的資訊，如果遇到 GetElementPtr 的話就是目前 IR 指令要從 source operand 切換到操作 destination operand，我有維護一個變數(sddiff)專門記錄目前是在操作 source 或是 destination operand，以利我都有確實正確記錄到我想記錄的數值。

還有就是處理運算相關的指令(sub、add、mul)，我用 dyn_cast 和 getOperand 將每條運算相關 statemant 的參數的整數部分取出，將這些資訊根據目前是在操作 source operand 還是 destination operand 將他們記錄到不同的陣列中，以利後續利用上課教的 diophantine 公式計算不同 statement 但相同 array 變數之間的 dependency。

我寫的 diophantine function 參數中有兩個 array 中的 index(i)的 mul 數值跟 add、sub 的值、迴圈的起始值跟結束值、要分析的 dependency 類型、還有兩個 array 的變數名稱，有了這些資訊就能用 diophantine function 計算出是否有 dependency 以及他們發生的位置，再透過參考剛剛紀錄參數的陣列就能知道是哪兩個 statement 之間發生 dependency，最後用公式計算出發生 dependency 的位置，一起將這些資訊輸出到 terminal。

以下圖 3、圖 4 分別為 test1.c 和 test2.c 的實驗結果

```
ubuntu@ubuntu-MS-7A38:~/advanced_compiler/hw1$ make run
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/clang -Xclang -disable-O0-optnone -fno-discard-value-
names -S -emit-llvm -o test1.ll test1.c
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/clang -shared -o hw1.so hw1.cpp `/home/ubuntu/advance
d_compiler/llvm_build/bin/llvm-config --cppflags` -fPIC -fno-rtti `/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-conf
ig --ldflags`
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/opt -disable-output -load-pass-plugin=./hw1.so -passe
s=hw1 test1.ll
====Flow Dependency====
(i=15, i=19)
A: S1 -----> S2
(i=14, i=18)
A: S1 -----> S2
(i=13, i=17)
A: S1 -----> S2
(i=12, i=16)
A: S1 -----> S2
(i=11, i=15)
A: S1 -----> S2
(i=10, i=14)
A: S1 -----> S2
(i=9, i=13)
A: S1 -----> S2
(i=8, i=12)
A: S1 -----> S2
(i=7, i=11)
A: S1 -----> S2
(i=6, i=10)
A: S1 -----> S2
(i=5, i=9)
A: S1 -----> S2
(i=4, i=8)
A: S1 -----> S2
====Anti-Dependency====
====Output Dependency====
```

圖表 3

```
ubuntu@ubuntu-MS-7A38:~/advanced_compiler/hw1$ make run
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/clang -Xclang -disable-O0-optnone -fno-discard-value-
names -S -emit-llvm -o test1.ll test2.c
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/clang -shared -o hw1.so hw1.cpp `/home/ubuntu/advance
d_compiler/llvm_build/bin/llvm-config --cppflags` -fPIC -fno-rtti `/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-conf
ig --ldflags`
`/home/ubuntu/advanced_compiler/llvm_build/bin/llvm-config --bindir`/opt -disable-output -load-pass-plugin=./hw1.so -passe
s=hw1 test1.ll
====Flow Dependency====
(i=2, i=2)
A: S1 -----> S2
====Anti-Dependency====
(i=7, i=17)
A: S2 --A--> S1
(i=6, i=14)
A: S2 --A--> S1
(i=5, i=11)
A: S2 --A--> S1
(i=4, i=8)
A: S2 --A--> S1
(i=3, i=5)
A: S2 --A--> S1
====Output Dependency====
(i=18, i=19)
D: S2 --O--> S3
(i=17, i=18)
D: S2 --O--> S3
(i=16, i=17)
D: S2 --O--> S3
(i=15, i=16)
D: S2 --O--> S3
(i=14, i=15)
D: S2 --O--> S3
(i=13, i=14)
D: S2 --O--> S3
(i=12, i=13)
D: S2 --O--> S3
(i=11, i=12)
D: S2 --O--> S3
(i=10, i=11)
D: S2 --O--> S3
(i=9, i=10)
D: S2 --O--> S3
(i=8, i=9)
D: S2 --O--> S3
(i=7, i=8)
D: S2 --O--> S3
(i=6, i=7)
D: S2 --O--> S3
(i=5, i=6)
D: S2 --O--> S3
(i=4, i=5)
D: S2 --O--> S3
(i=3, i=4)
D: S2 --O--> S3
(i=2, i=3)
D: S2 --O--> S3
```

圖表 4

4. Bonus

Mixing pattern

我上網查詢相關的 CRTP 資料發現有以下有點

第一點是程式碼重用性，透過繼承模板類別，我們的 pass class 可以重用共同的功能跟介面，可降低重複的程式碼，並且將 pass 的結構標準化。

第二點是可擴展性，通過繼承 PassInfoMixin 這樣的 class 來撰寫 pass，可以在現有的模板下製作出更多有特定功能的 pass，因此可以說這樣的方法是提升可擴展性。

第三點是將邏輯實作分離，在我們的例子中 PassInfoMixin 這個 class 提供 pass 常用的管理邏輯，HW1Pass 則專注在實作 pass 的功能，將邏輯和實作分開可以提供程式碼的可讀性跟維護性。

最後一點是避免誤用，將常用的邏輯或是功能包裝起來，需要用時再繼承或是呼叫，這樣的方式可以避免使用者誤用。