

國立交通大學

網路工程研究所

碩士論文

使用 LLVM 中間表示式產生腳本語言之技術

Generating Lua Code from LLVM IR



研究生：劉道源

指導教授：楊 武 教授

中華民國 102 年 11 月

使用 LLVM 中間表示式產生腳本語言之技術

Generating Lua Code from LLVM IR

研究生：劉道源

Student : Tao-Yuan Liu

指導教授：楊 武

Advisor : Wu Yang

國立交通大學



Submitted to Institute of Network Engineering  
College of Computer Science  
National Chiao Tung University  
in partial Fulfillment of the Requirements  
for the Degree of  
Master  
in

Computer Science

November 2013

Hsinchu, Taiwan, Republic of China

中華民國 102 年 11 月

# 使用 LLVM 中間表示式產生腳本語言之技術

學生：劉道源

指導教授：楊 武 博士

國立交通大學網路工程研究所

## 摘要

在現在的商業軟體，特別是遊戲開發上，腳本語言逐漸扮演著一個越來越重要的腳色。這同時也代表著腳本語言與傳統程式語言之間的溝通以及轉譯成為一件重要的事，特別是在開發前期這種程式架構還不太穩定的時候。

然而，現在這部分的工作仍然大多由程式設計師手動進行處理，這使得軟體開發前期會消耗大量的開發人力，並且產生許多廢棄的程式碼。然而，如果有適當的工具來處理這些工作，那麼就可以大幅減少這種無謂的浪費，提昇軟體開發的效率。

在本篇論文中，我提出一個結合 LLVM 而生的轉譯系統，並藉由此系統探討相關的優化技術。讓轉譯後的程式碼擁有更高的執行效率，更精簡的程式內容，以及更高的可讀性。期望軟體轉譯的程式碼變得更接近人工轉譯的版本。

# Generating Lua Code from LLVM IR

**Student : Tao-Yuan Liu**

**Advisor : Dr. Wu Yang**

Institute of Network Engineering  
National Chiao Tung University

## Abstract

In the business software, especially on game development, script languages take more and more important roles. It represents that translation and communication between scripting languages and programming languages and translation as an important thing become an important part of business software development, especially at the early stage of development – the system architecture is not stable yet.

Until now, this work is mostly handled by the programmer, this situation wastes a lot of time for software developers at early stage of development, and produce many useless codes. However, if we have an software to deal with these work, it is possible to reduce this unnecessary wasting, and improve the efficiency of software development.

In this paper, we propose a translating system which combines with LLVM system, and discussed related optimization techniques in this system. Let target source code have higher efficiency, simpler instruction statement, and higher readability. Expected source code which is translated by software become more similar to human version.

## 致謝

關於這篇論文，首先我要感謝我的指導教授楊武教授，他給我很大的自由空間，讓我可以投入腳本語言這個我之前就很有興趣的領域。此外，老師對於作研究的嚴謹要求與態度，是讓我的這篇論文不斷進步的重要因素。

另外，我也要感謝博班的俊宇學長與碩班的家倫同學。俊宇學長常常在我遇到困難時給與我指導，每次向他請教都讓我獲益良多；家倫同學則會和我分享許多作研究和報告的小技巧，讓我省下不少功夫。沒有他們的幫助，我的研究一定會遭遇更多的險阻。

除此之外，我還要感謝碩班的柏亨同學，正是因為他所提供的一篇 Paper，才會讓我產生靈感來研究這個題目。

最後，我要感謝我的家人在背後支持我，讓我有勇氣辭去工作，回歸學校研讀碩士，最後終於完成這篇論文。僅以此文獻上對他們由衷的感謝！

The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 100-2218-E-009-010-MY3 and NSC 100-2218-E-009-009-MY3.

# 目錄

摘要.....	i
Abstract.....	ii
致謝.....	iii
目錄.....	iv
圖表目錄.....	vi
第一章 緒論.....	1
1.1 背景與動機.....	1
1.2 研究目標.....	2
1.3 論文架構.....	3
第二章 相關研究.....	4
2.1 腳本語言介紹.....	4
2.2 Lua 介紹.....	6
2.3 Low Level Virtual Machine.....	6
2.4 LLVM Backend.....	7
2.5 Emscripten.....	9
第三章 系統設計與實作.....	11
3.1 簡介.....	11
3.2 架構.....	11
3.2.1 Load Module.....	12
3.2.2 Initialization.....	12
3.2.3 Function Generalize.....	13
3.2.4 Function Initialization.....	13
3.2.5 Loop Recovery.....	14
3.3.3.1 Branch Analyze.....	15
3.3.3.2 CFG Rebuild.....	15
3.3.3.3 Reloop.....	15
3.2.6 Basic Block Initialization.....	16
3.2.7 Instruction Process.....	16
3.3 實作細節.....	17
3.3.1 Function Generalize.....	18
3.3.2 Loop Recovery.....	25
3.3.3 Implementation of Instruction translation.....	37
3.3.3.1 Pointer variable process and Instruction optimization.....	37
3.3.3.2 Special instruction process.....	43
第四章 成果與分析.....	47

4.1 環境設置.....	47
4.2 實驗方法.....	48
4.3 成果展現.....	48
4.3.1 啟用 Instruction Optimization 造成之影響.....	48
4.3.2 啟用 Loop Recovery 造成之影響.....	49
4.4 整體成果.....	51
4.5 Lua 後端與 Emscripten 的比較.....	53
第五章 結論與未來展望.....	54
5.1 結論.....	54
5.2 未來發展與改進方向.....	54
參考文獻.....	56



# 圖表目錄

圖 2-1 各種語言的比較 .....	5
圖 2-2 LLVM 前端(Frontend)與後端(Backend)架構 .....	11
圖 3-1 系統架構 .....	12
圖 3-2 LLVM 前後端對 Template 函式的處理 .....	18
圖 3-3 Preliminary Function Match .....	19
圖 3-4 Nested Function Analyze .....	19
圖 3-5 遞迴函式範例 1 .....	23
圖 3-6 遞迴函式範例 2 .....	23
圖 3-7 Shape 結構範例 .....	26
圖 3-8 Simple Shape 範例 .....	27
圖 3-9 Loop Shape 範例 .....	27
圖 3-10 Multiple shape 範例 .....	28
圖 3-11 Reloop 運行步驟 1 .....	31
圖 3-12 Reloop 運行步驟 2 .....	34
圖 3-13 Reloop 運行範例 1 .....	35
圖 3-14 Reloop 運行範例 2 .....	36
圖 3-15 Reloop 運行範例 3 .....	37
圖 3-16 Instruction translation 執行步驟 .....	38
圖 3-17 轉譯與執行順序不同之問題 .....	39
圖 3-18 啟用 Preliminary variable process .....	40
圖 3-19 Instruction translation 範例 1 .....	41
圖 3-20 Instruction translation 範例 2 .....	42
圖 3-21 Instruction translation 範例 3 .....	42
圖 3-22 PHI 指令 .....	44



圖 3-23 分解 PHI 指令.....	44
圖 3-24 轉譯 PHI 指令範例.....	45
圖 3-20 節點管理範例 1 .....	錯誤! 尚未定義書籤。
圖 3-21 節點管理範例 2 .....	錯誤! 尚未定義書籤。
圖 3-22 新 Static Subscribe Server 註冊 .....	錯誤! 尚未定義書籤。
圖 3-23 Tunnel Setup 節點選取 .....	錯誤! 尚未定義書籤。
圖 3-24 Tunnel Setup 封包 .....	錯誤! 尚未定義書籤。
圖 3-25 Tunnel Setup 示例 1 .....	錯誤! 尚未定義書籤。
圖 3-26 Tunnel Setup 示例 2 .....	錯誤! 尚未定義書籤。
圖 3-27 Tunnel Setup 示例 3 .....	錯誤! 尚未定義書籤。
表 3-1 indexMap 以及 varMap 結構.....	43
圖 4-1 有無 Instruction Optimization 機制對 Statement 數之影響.....	48
圖 4-2 有無 Instruction Optimization 機制對執行時間之影響.....	49
圖 4-3 Loop Recovery 機制影響範圍 .....	50
圖 4-4 有無 Loop Recover 機制對 Statement 數之影響 .....	50
圖 4-5 有無 Loop Recover 機制對執行時間之影響 .....	51
圖 4-6 整體優化機制對 Statement 數之影響 .....	52
圖 4-7 整體優化機制對執行時間之影響 .....	52
圖 4-8 Lua 後端與 Emscripten 對 Statement 數之比較.....	53
表 4-1 測試環境 .....	47

# 第一章 緒論

## 1.1 背景與動機

在現代的商業軟體開發上，除了程式的執行效率以外，開發的時間長短也是極為重要的一件事。因為如此，在軟體開發上常常使用多人分層開發以及部分外包的方式來進行。

然而，使用這種開發方式，固然會增加開發的效率，卻也會產生新的問題：其一是需要更多的專業程式人員；其二是會有外包人員外洩原始碼的風險。為了解決這些問題，腳本語言(Script Language)在某一層面上已經開始取代傳統的程式語言，成為商業軟體開發的新寵兒。

由於腳本語言擁有[1]：

- 語法和結構通常比較簡單
- 學習和使用通常比較簡單
- 通常以容易修改程式的「直譯」作為執行方式，而不需要「編譯」
- 程式的開發產能優於執行效能
- 容易與其他程式元件搭配

的特性，使得使用腳本語言的門坎遠低於傳統程式語言。這樣一來，就可以將使用腳本語言撰寫的部分交給其他非專業的程式人員來負責，如同變相增加了開發人手。而只有將腳本語言的部分外包，同時也保障了傳統程式語言部分原始碼的安全性。因為如此，越來越多的商業軟體(特別是遊戲)開發都採用「程式語言為主體+部份功能使用腳本語言撰寫」的模式來進行。

不過，即使使用腳本語言輔助開發，在開發早期，仍然可能由於整體的軟體架構設計還不穩定，導致腳本語言部分的程式與傳統程式語言部分發生變動的情形。在這種狀況下，一般的處理方式是：程式人員將已經製作完成的傳統程式版本的原始碼放棄，然後訂定相對應的規格文件，最後再將規格文件交由腳本人員製作腳本版本的原始碼。但是這種方式，除了會有重覆發明輪子的老問題之外，增加的溝通行為，也會減少開發人員的專注力與工作效率。因此，如果有適當的工具來處理這些工作，那麼不但可以減少開發人員不必要的溝通，更可以回收原本被廢棄的程式碼，提昇整體開發的效率。

## 1.2 研究目標

基於以上的理由，我提出一種與 LLVM 結合的轉譯器，能夠將 LLVM 的中間表示式，轉譯成 Lua 這種腳本語言。這樣不但可以使用到 LLVM 本身的優化能力，還可以利用 LLVM 所具有的跨平台特性，讓轉譯器能擁有更大的可擴展性。

除此之外，由於腳本語言具有許多高階語言的特性，我也藉由此轉譯器探討相關的優化技術。讓轉譯後的程式碼不僅僅是由中間表示式直接轉譯成的程式碼，更能利用腳本語言的特性進行優化，讓程式碼擁有更高的執行效率，更精簡的程式內容。同時由於腳本語言的程式碼仍然需要搭配傳統程式語言來執行，也有可能之後會讓程式人員進行維護，因此對於可讀性方面，也要進行優化，方能減少之後的維護成本。至於最終目標，則是希望達到我轉譯器轉譯出的程式碼，能變得更為接近人工轉譯的版本。

本篇論文主要即是描述轉譯系統整體的架構，並且針對 LLVM 中間表示式

與 Lua 腳本語言之間的轉譯跟優化所需建立各種機制進行說明以及介紹。

### 1.3 論文架構

在本章裡，我對於腳本語言進行了簡單的介紹，並且解釋了整體的研究動機以及目標。而在第二章中，將會介紹轉譯器會使用到的背景技術，以及相關工具。第三章則會描述整體的系統設計，以及各部份的實作內容。在第四章會介紹我的研究成果，還有相關的實驗數據。在最後的第五章會對我目前的研究結果做總結，以及提出未來還能夠改進的目標。



## 第二章 相關研究

### 2.1 腳本語言介紹

腳本語言是為了縮短傳統的「編寫、編譯、連結、執行」(edit-compile-link-run)過程而建立的電腦編程語言。通常都有簡單、易學、易用的特性，目的就是希望能讓程式設計師快速完成程式的編寫工作[1]。簡單來說，與其說腳本語言是從頭開始編寫應用程序，還不如說是結合現有的組件產生出符合目標的功能[2]。

一般而言，腳本語言具有以下幾種特性：

- 語法和結構通常比較簡單
- 學習和使用通常比較簡單
- 通常以容易修改程式的「直譯」作為執行方式，而不需要「編譯」
- 程式的開發產能優於執行效能
- 容易與其他程式元件搭配

由於腳本語言是源自於早期作業系統的 shell 指令，因此執行方式大多為直譯式執行，基本的設計也是較偏向於一般的電腦使用者，而不是專門的程式設計師。主要目的也是簡化代碼編寫時間，像是 Garbage Collection、Memory Map 以及 Overflow Check 等問題皆是自動處理，以減少撰寫人的負擔。

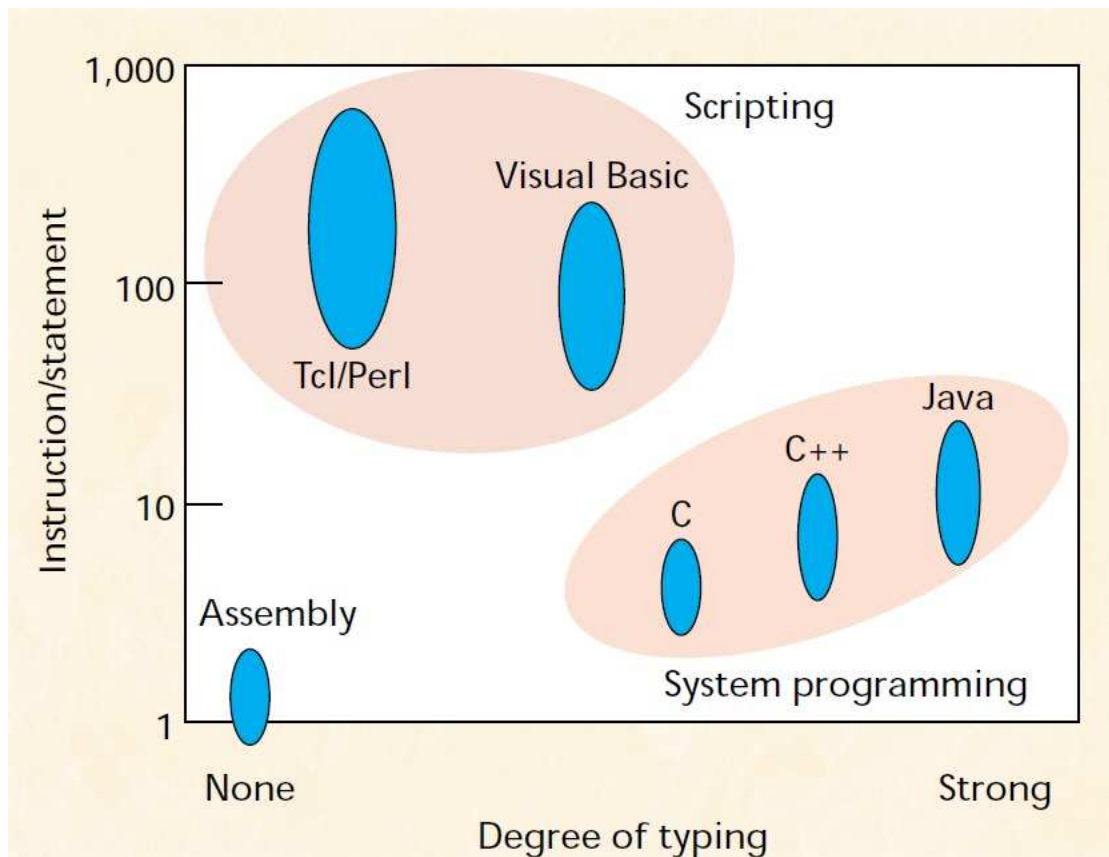


圖 2-1 各種語言的比較[2]

也因為如此，腳本語言要與其他程式元件之間進行通訊，所需要進行的設定也要簡單的多，並且會提供許多可以進行高階的操作的指令碼。整體而言，就如同圖 2-1 所顯示的，每一個腳本語言的 **Statement** 大約可以轉換成 100 到 1000 個低階指令，而傳統的程式語言每個 **Statement** 只能轉換成 5 到 10 個低階指令[2]。

如同以上所言，腳本語言擁有開發速度更快，而且檔案大明顯小於傳統程式語言的檔案。相對的，它在執行速度上遠遠不如傳統程式語言。腳本語言通常是直譯式執行，速度不僅慢，而且記憶體使用量也更多。然而，由於硬體速度的大幅演進，如果是編寫小型程式，兩者的差距並不大。即使是大型程式，也可搭配傳統程式語言使用來縮小兩者之間的差距。在軟體程本趨高而硬體成本趨低的現今，擁有低開發門坎的腳本語言的確在軟體設計上開拓了一片新領域。

## 2.2 Lua 介紹

在眾多腳本語言中，Lua 是一個功能強大，速度快，重量輕，可嵌入的腳本語言[3]。其中最為特殊的一點，就是他非常容易嵌入其它程式語言中使用。只要是宿主語言所實作出來的功能，Lua 都可以輕易的使用他們來擴充自己的功能性。而實際上，Lua 也已經擁有許多成熟的擴充功能模組讓使用者能夠更為方便來使用。

如同其他的腳本語言，Lua 也具有 Dynamically Typed，以及 Garbage Collection 自動處理等特性。而且相對於其他腳本語言，Lua 有著更為輕量的記憶體使用量以及更快的執行速度，有許多的 Benchmark 測試皆顯示 Lua 的執行速度在腳本語言中可算是最快的[3]。也因為如此，使得許多商業軟體皆選擇 Lua 作為編寫程式的主要語言之一。



## 2.3 Low Level Virtual Machine

Low Level Virtual Machine，即 LLVM 是一種 Compiler 的基礎平台。其最大的特點就是在編譯程式碼的過程中，提供了中間表示式這種獨立的中間層[4]。利用這種機制，LLVM 可對整個編譯過程的編譯時期、鏈結時期、執行時期以及閒置時期進行各自的最佳化。

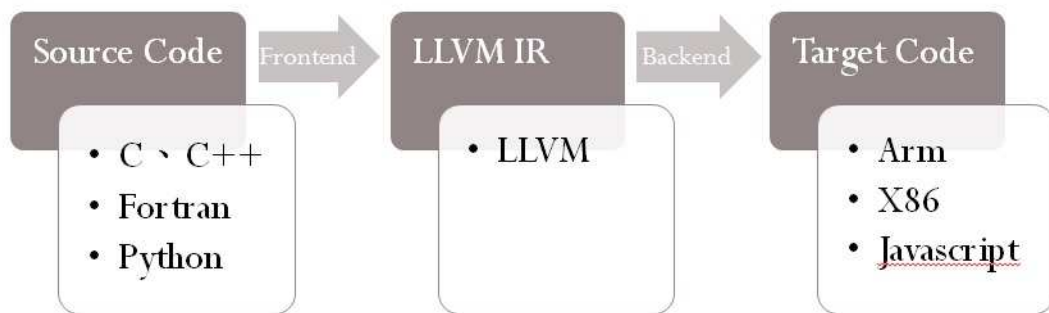


圖 2-2 LLVM 前端(Frontend)與後端(Backend)架構

也因為這種設計，使得 LLVM 可使整個編譯器模組化，將編譯器分為前端 (Frontend)與後端(Backend)各別設計。使用者可以自行撰寫各種的前端來將高階語言轉譯成中間表示式；也可以撰寫後端將中間表示式轉譯成各種機器碼。截至目前為止，LLVM 前端已經可以支援 ActionScript、Ada、D、Fortran、GLSL、Haskell、Java bytecode、Julia、C/C++、Python、Ruby、Rust、Scala 以及 C#[4]。LLVM 後端則可以支援 Alpha、ARM、Blackfin、CellSPU、JavaScript、MBlaze、MIPS、MSP430、PowerPC、PTX、Sparc、SystemZ、X86 以及 XCore 等眾多機器碼，這也顯示了 LLVM 擁有非常強大的跨平台能力。

## 2.4 LLVM Backend

LLVM 後端在 LLVM 整體架構中扮演者一個十分重要的角色。由於中間表示式無法被直接執行，故需要先由 LLVM 後端將中間表示式轉換成機器碼或指定的語言，才能讓實體或虛擬機器進行執行工作。

因此，在一般狀況下，要實作一個 LLVM 後端，必須進行以下幾個步驟：



### **註冊目標語言 (Target Registration) :**

當實作一個新 LLVM 後端時，首先是要向 LLVM 本體註冊。這樣以後在轉譯中間表示式時，LLVM 才能選擇此種語言作為目標語言。

### **定義暫存器 (Defining Register) :**

由於每種指令集架構的不同，我還必須設定目標語言的暫存器資訊。其中包括暫存器的名稱、數量以及類型(整數、浮點數、或是向量暫存器)。

### **指令運算元對映(Instruction Operand Mapping) :**

在指令集方面，首先要先定義每種指示所需使用的運算元數量，以及每個運算元所代表的意義。舉例來說，SPARC 語言的 XNORrr 指令需要三個運算元，而第三個運算元為接收前兩個運算元進行 NOT XOR 計算的結果，這都必須在 LLVM 後端裡事先定義好。

### **指令關係對映(Instruction Relation Mapping) :**

除此之外，如果有些中間表示式的指令必須用複數個目標語言的 Statement 來實作；或者是多個中間表示式的指令可以用單個目標語言的 Statement 來實作時，就必須設定兩者之間指令關係的對映。

### **Branch 指令縮簡以及 IF 指令轉換(Branch Folding and If Conversion) :**

如果目標語言有擁有特殊的 Branch 指令，則 LLVM 允許後端利用此功能進行 Branch 指令的縮簡以及轉換。此外，LLVM 也提供數個 API 讓後端進行 Control Flow Graph 的分析，讓後端能進行相關的優化作業。

### **Call 指令設定 (Call Convention) :**

由於中間表示式裡有 `Call` 這種函式呼叫指令，所以也要在目標語言中設定相關的 `Flag` 以及指令來實作對應的行為。

以上便是建立一個 LLVM 後端所大致需要實作的幾個部分。然而，實際上所需實作的內容會根據後端的目標語言而有所不同。特別是當目標語言不是機器碼時，有些步驟則會有所不同。

## 2.5 Emscripten

Emscripten[6]是一個將 LLVM 中間表示式轉譯成 JavaScript 的轉譯系統。其主要功能是利用 LLVM 可以支援多種語言的特性，以及大多數網頁瀏覽器皆支援 JavaScript 的特點。讓許多並非使用 JavaScript 撰寫的程式，在不需要大幅改寫的狀況下，也能夠在網頁上執行。其主要可分為三個部分：

### Intertyper

將輸入的 LLVM 中間表示式轉換成 Emscripten 自己的內部格式。

### Analyzer

對 Emscripten 內部格式資訊進行分析，進一步產生的各種有用的訊息，像是變數類型和資訊，堆疊使用狀況，以及相關的優化資訊等。

### Jsifier

根據 Emscripten 內部格式資訊以及 Analyzer 分析出的資訊，產生最後的 JavaScript 程式碼。

與我的轉譯器直接與 LLVM 搭配不同，Emscripten 是一獨立運作的轉譯系統。而其主要優化功能則是集中在 Loop 結構的重建。由於其優秀且穩定的演算法，我在實作 Loop Recovery 演算法時，也是根據 Emscripten 的演算法而進行修改。



## 第三章 系統設計與實作

### 3.1 簡介

如同前一章所言，當目標語言不是機器碼，反而是一般的程式語言或腳本語言時，LLVM 的後端所需實作的內容會與一般的後端有所不同。有些步驟，如定義暫存器會省略掉；然而相對的，要撰寫這種後端，也會有一般的後端所不會遇到的問題，例如 SSA Form 的轉換，以及指標變數的處理等問題。而這些問題，並不像一般的後端有許多現成的資訊可以參考，必須從 LLVM 所提供繁多的 API 中來想辦法解決。

同時，也不同於一般的後端只需要單純的轉換就可以執行，當目標語言是腳本語言時，單純轉換出來的程式碼，往往只類似於低階的機器碼，並不能發揮出高階語言應該要有的高可讀性與效率。這樣一來，便失去了轉譯成腳本語言的意義。因此我的後端還必須要加上許多的優化功能，讓轉譯出來的程式碼更接近一般腳本語言應該有的風格。

### 3.2 架構

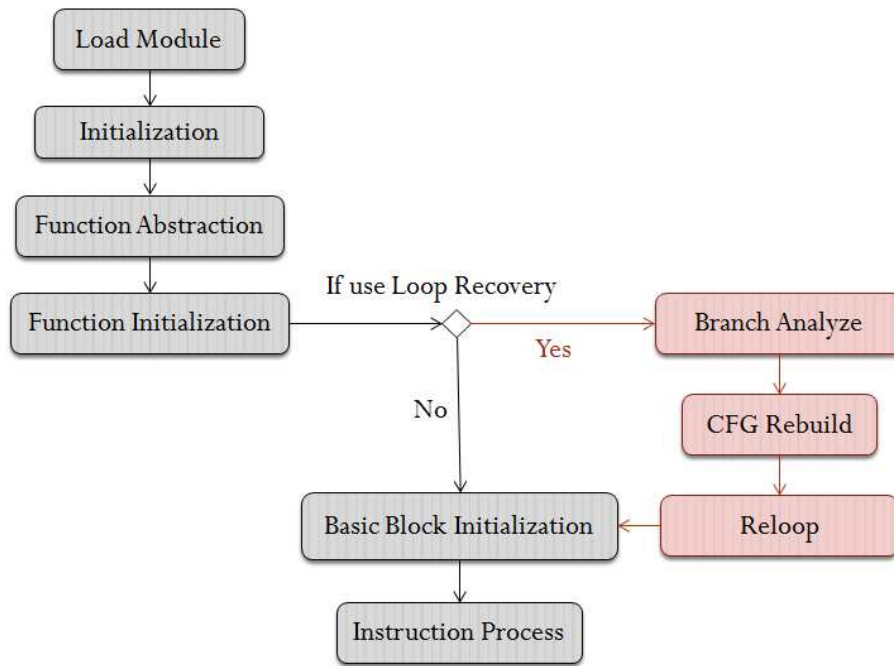


圖 3-1 系統架構

圖 3-1 是我的 LLVM 後端的基本架構。基本上，系統將中間表示式轉譯成 Lua 時，會分為數個步驟來處理：

### 3.2.1 Load Module

一開始，我的 LLVM 後端必須從 LLVM 本體載入 Module，Module 中包含著所輸入的原始碼的所有相關資訊。所以要先對 Module 進行分析，以方便之後步驟的處理。

### 3.2.2 Initialization

取得 Module 的相關資訊後，我首先要對 Global variable 以及 System function 進行處理。由於 Lua Script 是直譯式執行的，在設定上，不論是 Local variable 還是 Global variable 都必須先進行宣告後方能使用。所以一開始我就必須將 Global

variable 先宣告好，以方便之後的 statement 使用。

至於 System function 則是來自於 C 的內建函式庫。由於 LLVM 允許中間表示式直接使用 C 內建函式庫的函式，我所接受的原始碼中往往會有 C 內建函式庫的函式的宣告，卻沒有相關的定義。然而這些函式在 Lua Script 中大多是沒有，或者是格式不同。因此我必須要製作這些函式的定義，才能讓轉譯出來的程式正常運作。

### 3.2.3 Function Generalize

初步的設定完成之後，接下來就是進行函式(Function)層的處理。在這裡，我會對函式層進行優化作業。

由於 LLVM 的中間表示式也是採用 Strong Type 型式，所以前端在建立函式時，即使函式內部是進行相同的工作，只要函式的引數(Arguments)是不同型別的，LLVM 前端依然會各自建立互相獨立的函式。

然而，這些互相獨立函式對於 Weak Type 型式的腳本語言而言，卻大多是多餘的。所以 Function Generalize 這步驟就是負責檢查所有的函式，找出對於腳本語言而言是功能雷同的部分，並建立對照表以方便之後進行轉譯時使用。至於詳細的內容，會在實作細節中做說明。

### 3.2.4 Function Initialization

Function Initialization 步驟，基本上又可分為兩部份，其一是對每個函式的

引數進行初始設定，包括可變引數(variable number of arguments)的處理，以及為了之後指標變數的處理，預先檢查各個引數的型別，並分別對指標型別的引數以及一般引數進行相關處理。

第二部分則是預先對函式底下的指令進行初步的檢查，對函式中所有指令會用到的變數建立對應的基礎資訊表，以方便之後的 **Instruction Process** 步驟進行處理。

### 3.2.5 Loop Recovery

處理完函式層面的問題之後，我接著要對 **Control Flow** 的基礎結構，LLVM 中間表示式的 **Basic Block** 層進行分析以及優化。

由於 LLVM 中間表示式是屬於類似組合語言的低階語言，所以當前端把原本的程式碼轉譯成中間表示式後，一切的迴圈結構都會被破壞，**Control Flow** 會變成由基本的 **branch** 指令來組成。

然而，這樣子雖然對執行速度上可能會有提昇(但是對於腳本語言而言，這種提昇微乎其微)；相對的，只使用基本 **branch** 指令的程式碼不僅對 **statement** 數會有負面的影響，更重要的是會大幅增加人類閱讀的困難性。這對於程式以後的維護會造成很大的麻煩。

因此，將已經被摧毀的迴圈結構重新建立起來，對於我的後端而言，具有很高的必要性。

### 3.3.3.1 Branch Analyze

要重新建立迴圈結構，首先就是要搜尋函式底下的所有 Basic Block。首先先確定函式的進入點，之後則是分析每個 Basic Block 中的 branch 指令，以了解每個 Basic Block 之間的關係。

### 3.3.3.2 CFG Rebuild

確認每個 Basic Block 之間的關係後，則要依據關係建立整個函式內部的 Control Flow Graph，以方便 Reloop 步驟進行處理。

### 3.3.3.3 Reloop

最後則是對 Control Flow Graph 的結構進行優化，並且依此重新建立迴圈結構。在這一步驟，需要十分可靠的演算法，以確定重新建立迴圈結構後整體的 Control Flow 依然是一致的。

因此，我在這裡採用 Emscripten 的 Relooper 演算法[6]。Emscripten 是已經公開發布相當時間的軟體，其演算法也經過許多實際的商業軟體測驗過，在可靠性方面可以算是無庸置疑。而為了配合 Lua Script 的語法結構，我再對它進行以下修改，方能使用在我的後端上進行重新建立迴圈結構的工作。

- [1] 在每層 Loop 的尾端增加 goto 指令所能使用的 Label。
- [2] 由於 Lua 腳本語言並沒有 break 到指定某層 Loop 的功能，故將此種 break 指令使用 goto 指令取代。



[3] 由於 Lua 腳本語言並沒有 `continue` 指令，故改為使用 `goto` 指令取代。

### 3.2.6 Basic Block Initialization

有時候，LLVM 中間表式會有一些對於中高階語言而言是不必要的指令，例如：在使用 `malloc` 取得記憶體後，LLVM 中間表式會使用數個指令來確保記憶體的指標的確是指在正確的位址。然而，這些指令對於中高階語言，特別是腳本語言而言，卻是沒有意義的指令。

Basic Block Initialization 步驟的主要工作，就是在 Basic Block 中檢查這些不必要的指令，並且事先將之排除。如此一來，不但可以增加轉譯後的程式的效率，還能減少 Instruction Process 步驟的負擔。

### 3.2.7 Instruction Process

最後的 Instruction Process 步驟，就是進行指令間實際的轉換。不過，就如同本章一開始所提的，我的後端並不只是進行單純的轉換。為了發揮出高階語言應該要有的高可讀性與效率，以及配合 Lua 腳本語言的特性，Instruction Process 步驟還包括了特殊功能，簡述如下：

#### Instruction optimization

由於 LLVM 中間表示式是類似組合語言的低階語言，會有 `load` 以及 `store` 這種指令負責將數值在記憶體以及暫存器之間存取。然而，這些指令一旦轉換成腳本語言，就變成是單純的參數之間數值的傳遞。而這些數值的傳遞指令，對腳本語言而言，有相當部分是可以省略的。

因此，我的後端會對這些指令進行一定程度的優化，這樣一來，轉譯出來的腳本語言不管是執行速度還是總 `statement` 數都會有明顯的改善。

### **Pointer variable process**

對於大部分的腳本語言而言，一般的變數是只能以 `By Value` 方式傳遞，無法以 `By Reference` 方式傳遞。換句話說，腳本語言基本上是沒有指標變數的種概念的。所以面臨 LLVM 中間表示式的指標變數時，我也必須對其進行特殊處理。

### **PHI instruction process**

由於 LLVM 中間表示式有使用 `SSA Form` 的指令「`phi`」，而這種指令在腳本語言中是沒有的，因此要對 `phi` 指令做特殊的處理。

### **Boolean to Integer translation**

由於 Lua 腳本語言具有一個特性，即布林值與數值無法通用。這與一般「零即為 `True`，非零即為 `False`」的程式語言不同，因此當 LLVM 中間表示式使用到布林值與數值之間的轉換時，也要特別加以處理。

## **3.3 實作細節**

在前一節，我對於整個 LLVM 後端的架構進行了大略的介紹，同時也說明進行轉譯時的各個步驟。而在這一節，我會對 LLVM 後端中各個主要部分進行詳細的說明。

### 3.3.1 Function Generalize

如同之前所提的，LLVM 前端在產生中間表示式的函式時，只要函式的引數的型別不同，即使函式內部的工作是相同的，LLVM 前端依然會各自建立獨立的函式。

舉例來說，當前端在轉譯 C++ 的 `template` 函式時，就很可能會產生類似的情形。LLVM 前端會將一個 `template` 函式轉譯成許多相同功能，但是引數的型別不同的函式。

而這些函式在某種程度上，卻可利用腳本語言 `Weak Type` 特性將它們合而為一，就如同圖 3-2 所示。

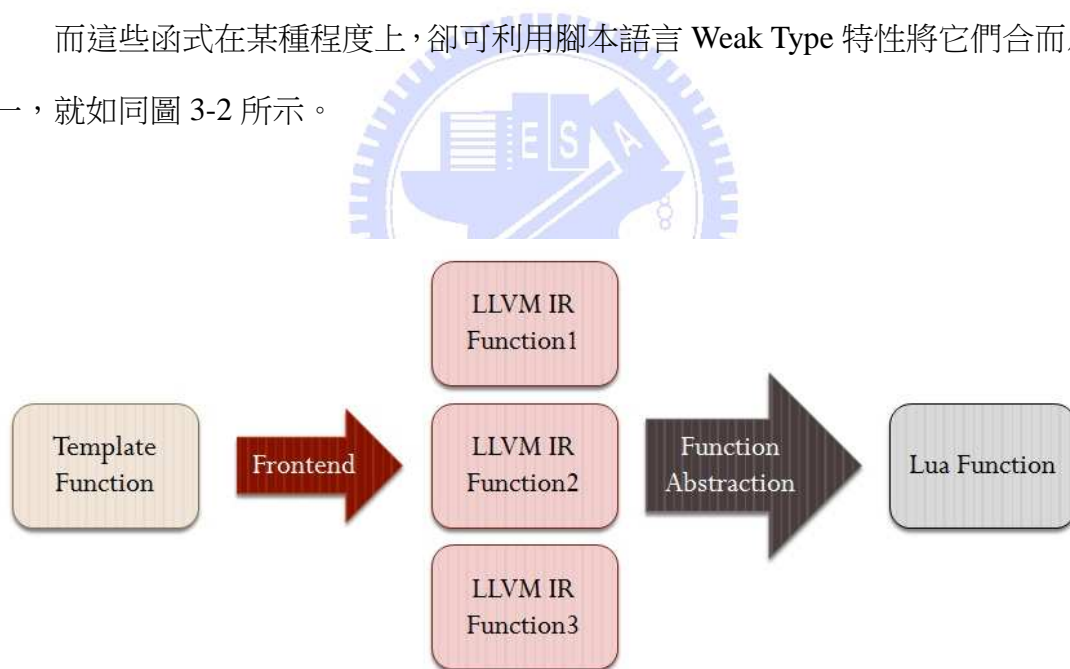


圖 3-2 LLVM 前後端對 `Template` 函式的處理

`Function Generalize` 就是設計來處理此一工作，其下又可分為兩部分：

#### Preliminary Function Match

比對兩個函式中所有指令，確認其功能是否相同。如果所有指令的功能都相同的話，將此函式組合加入「類似函式」的清單(`Match List`)中；如果除了函式呼

叫以外指令的功能都相同的話，但是函式呼叫指令所呼叫函式並不相同的話，將此函式組合加入「可能類似函式」的清單(Similar List)中，如圖 3-3 所示。

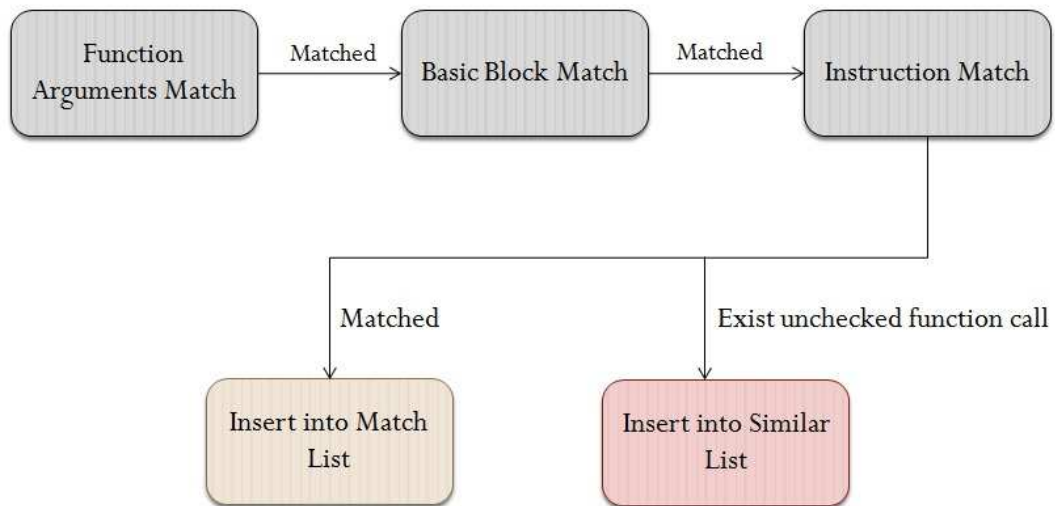


圖 3-3 Preliminary Function Match

### Nested Function Analyze

根據已經完成的「可能類似函式」清單，對裡面的每組函式進行更為嚴謹的比對，確認函式中的 call 指令的確是呼叫相同功能的函式。

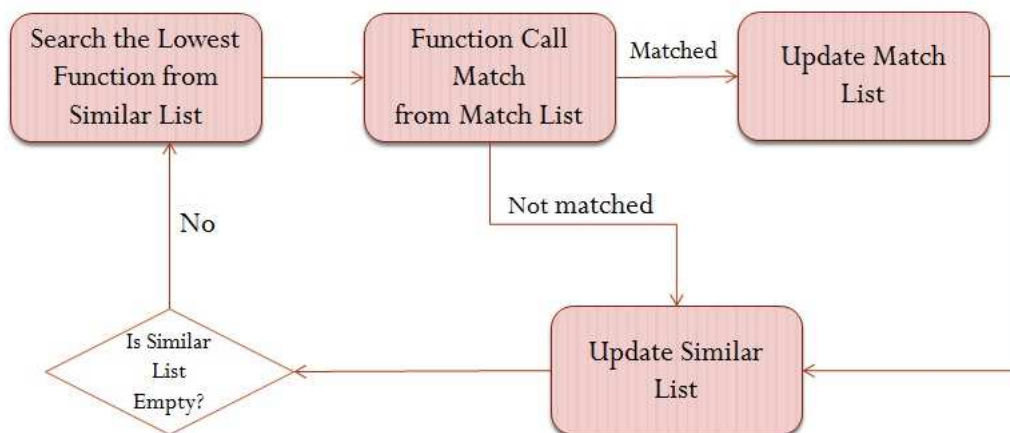


圖 3-4 Nested Function Analyze

如圖 3-4 所述，所有的通過 **Preliminary Function Match** 檢測的函式組都會從最底層的函式開始進行比對，直到所有能比對的函式組皆確認完畢，詳細的演算法如下：

### **Preliminary Function Match**

```
for(every function F1 in module){  
    for(every function F2 in functionMap){  
        if(the number of argument in F1 == the number of argument in F2){  
            if(isFunBodySimilar(F1, F2) == STRONG)  
                templateMap.insert(F1, F2)  
            else if(isFunBodySimilar(F1, F2) == WEAK)  
                weakTemplateMap.insert(F1, F2)  
        }  
    }  
    functionMap.insert(F1, F1's name)  
}
```

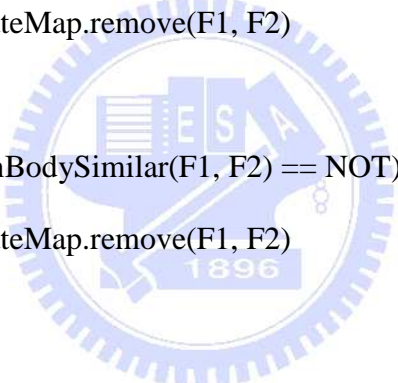
其中 **module** 即是前一節所提的包含所有原始碼的相關資訊的物件；**functionMap** 則是記錄著所有確定沒有相似函式的函式資訊；至於 **templateMap** 是記錄已經確定功能相同的函式組，**weakTemplateMap** 則是記錄可能功能相同的函式組(還未確認 **call** 指令是否是呼叫相同功能的函式)，此三個 **Map** 初始皆為空。

**isFunBodySimilar** 函式則是負責檢查函式組內部指令的功能是否相同，其傳

回值分為三種：STRONG 代表此一函式組內部指令的功能完全相同；WEAK 則代表函式組內部指令的功能大致相同，但卻有 call 指令呼叫不同的函式，所以還需進一步確認；NOT 則代表函式組的功能並不相同。

### Nested Function Analyze(Base)

```
while(weakTemplateMap.size dosen't decrease){
    for(every function pair<F1, F2> in weakTemplateMap){
        if(isWeakFunBodySimilar(F1, F2) == STRONG){
            templateMap.insert(F1, F2)
            weakTemplateMap.remove(F1, F2)
        }
        else if(isWeakFunBodySimilar(F1, F2) == NOT)
            weakTemplateMap.remove(F1, F2)
    }
}
```



在 Nested Function Analyze 演算法中的 isWeakFunBodySimilar 函式的功用是利用 templateMap 來檢查所輸入的函式組中的 call 指令所呼叫的函式功能是否相同，其傳回值格式與 isFunBodySimilar 函式相同，演算法如下：

```
isFunBodySimilarWeak(F1, F2){
    for(every basic block B1 in F1 and every basic block B2 in F2 ){
        for(every instruction I1 in B1 and every instruction I2 in B2 ){
            if(I1 and I2 are call instruction){
```

```

        CF1 = called function of I1
        CF2 = called function of I2
        if(CF1 != CF2 and templateMap.find(CF1) != CF2)
            return WEAK
    }
}
}
return STRONG
}
.....

```

整體而言，如同演算法所顯示的，在 Preliminary Function Match 時我會先建立 `templateMap` 以及 `weakTemplateMap` 兩組資料。之後在 Nested Function Analyze 時我則使用 `templateMap` 中的資訊一步一步檢驗 `weakTemplateMap` 中的函式組功能是否相同，檢驗後的資訊則又能回饋到 `templateMap`，藉此達到由低層往高層逐步確定 `weakTemplateMap` 中的函式組是否一致的目的。

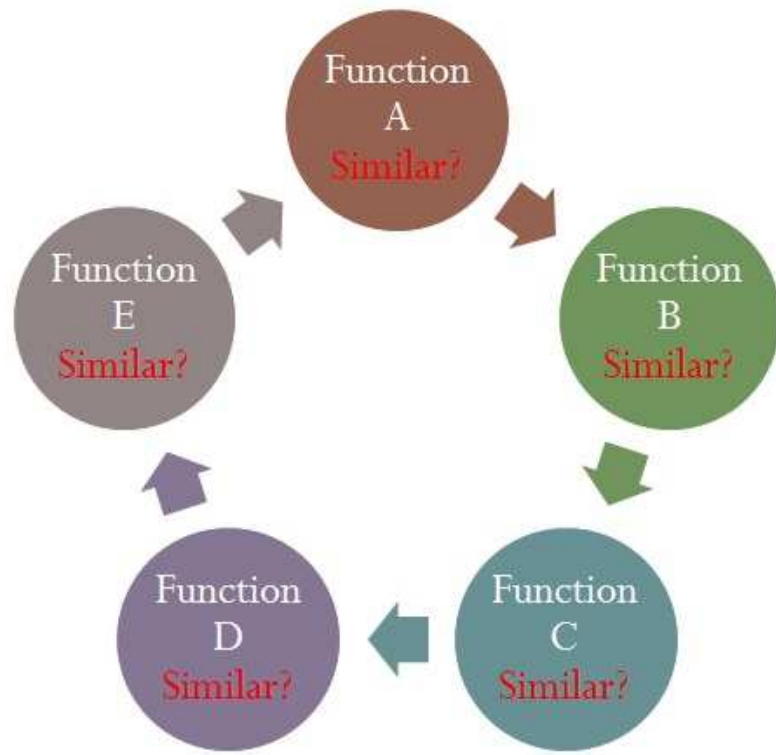


圖 3-5 遞迴函式範例 1

然而，只有這樣子的處理還會有一個缺陷：就是當函式之間的呼叫關係並不是單純的樹狀，而有迴圈產生時，例如遞迴函式，光憑上述的演算法就無法確認這些函式是否擁有相同的功能。

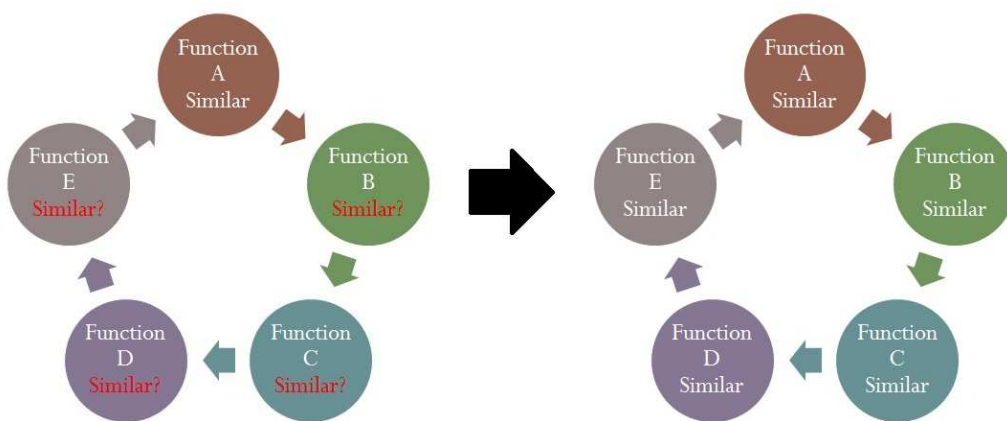


圖 3-6 遞迴函式範例 2

在這種狀況下，我採取的方法是先假定迴圈中某一個函式是相同功能的，然



後依以假定繼續推演。假如最後能推演回原本所假定的函式，則代表整個迴圈中的函式皆是具有相同功能，演算法如下：

### **Nested Function Analyze(Recursive)**

```
int I = 0
while(I < weakTemplateMap.size()){
    orgWeakTemplateMap = weakTemplateMap
    orgTemplateMap = templateMap
    function pair<SF1,SF2> = weakTemplateMap[I]
    templateMap.insert(SF1, SF2)
    while(weakTemplateMap.size dosen't decrease){
        for(every function pair<F1, F2> in weakTemplateMap){
            if(isWeakFunBodySimilar(F1, F2) == STRONG){
                templateMap.insert(F1, F2)
                weakTemplateMap.remove(F1, F2)
            }
        }
    }
    if(weakTemplateMap.find(FF1) != NULL){
        weakTemplateMap = orgWeakTemplateMap
        templateMap = orgTemplateMap
        I++
    }
    else
```

I = 0

}

`weakTemplateMap` 以及 `TemplateMap` 即是之前演算法所使用之物件，而 `orgWeakTemplateMap` 以及 `orgTemplateMap` 則是為了還原所建立的備份資料。一開始，我先在 `weakTemplateMap` 中隨機挑一組函式組並將它加入 `templateMap` 做為假定用的初始函式組，之後再進行推演。

在推演的過程中，被認定是相同功能的函式組會從 `weakTemplateMap` 中移除。所以當推演完成後，如果 `weakTemplateMap` 中沒有原本所假定的初始函式組，那代表這個推演能證明回自身，那麼這個假定是成功的，可以根據更新後的 `weakTemplateMap` 以及 `TemplateMap` 進行下一步的假定。相對的，如果 `weakTemplateMap` 中依然有原本所假定的初始函式組，那就意味著原本的假定失敗，必須從原本的 `weakTemplateMap` 中重新找一組來進行新的推演。

### 3.3.2 Loop Recovery

如同 2-2 節對 Lua 的介紹，Lua 最為特殊的一點，就是他非常容易嵌入其它程式語言中使用。但是，就算是再容易嵌入的腳本語言，Lua 仍然需要一些協定才能達成與宿主語言的溝通。而這些協定的設定，基本上還是要由程式設計師手動進行處理。

因此，與一般的轉譯程式不同，我的後端對於轉譯出來程式碼的可讀性具有一定的要求，而 `Loop Recovery` 就是我為了增加可讀性而建立的機制之一。主要可分為 `Branch Analyze`、`CFG Rebuild` 以及 `Reloop` 三個部分，而其中 `Reloop` 是最

重要的部分。

基本上，**Reloop** 並不是試圖回復原本已經被摧毀的迴圈結構，而是直接根據現有的 **CFG** 重新建立新的迴圈結構。這樣不僅在實作上比較容易，還能夠使用優化技術產生出更加優秀的迴圈結構。

如同 3-2 節所述，由於重新建立迴圈結構後很難檢驗程式的 **Control Flow** 絕對沒有變動。為了保障重建機制的正確性，我使用修改過的 **Emscripten Relooper** 演算法為核心來實作。**Relooper** 演算法是使用深度優先搜尋來分析 **CFG**，並在原本的 **Basic Block** 結構上再加上一層名為 **Shape** 的結構來表示迴圈結構，如圖 3-7 所示。

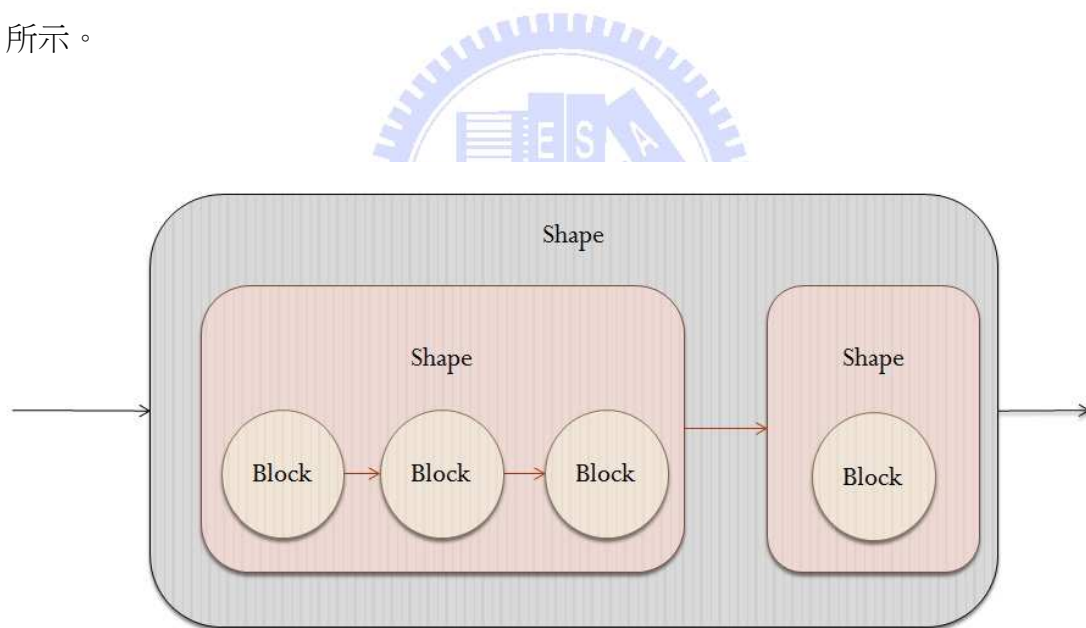


圖 3-7 Shape 結構範例

而它所使用的 **Shape** 結構可以分為三種：

### **Simple Shape**

**Simple Shape** 是最基礎的 **Shape**，同時也是最底層的 **Shape**，它只含有一個 **Basic Block**。基本上，**Simple Shape** 就是單純在單個 **Basic Block** 上加上一層 **Shape**

的殼而已。當 Relooper 演算法所遇到的是無法轉換成迴圈結構的 Basic Block 時，它就會將 Basic Block 設定成 Simple Shape。

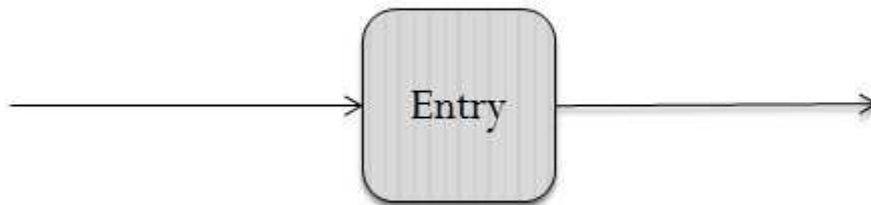


圖 3-8 Simple Shape 範例

### Loop Shape

Loop Shape 底下包含著一連串的 Basic Block。與 Simple Shape 最為關鍵的不同在於，Loop Shape 的 Entry Block 會擁有一個以上的 in-edge，也就是會有多個 Basic Block branch 到 Entry Block。在這種狀況下，Relooper 演算法就會判定這串 Basic Block 可以組成一個 Loop Shape。而每個 Loop Shape 在轉譯後皆會變成一個 while 迴圈。

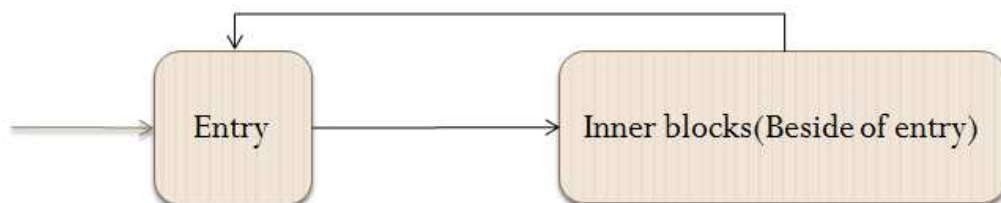


圖 3-9 Loop Shape 範例

## Multiple Shape

與前面兩種 Shape 不同，Multiple Shape 最大的特點在於它擁有多個 Entry Block。每個 Entry Block 各擁有一個名為 Independent Group 的子 Block 串。而在轉譯後，Multiple Shape 會變成一連串の if 以及 else if 結構。

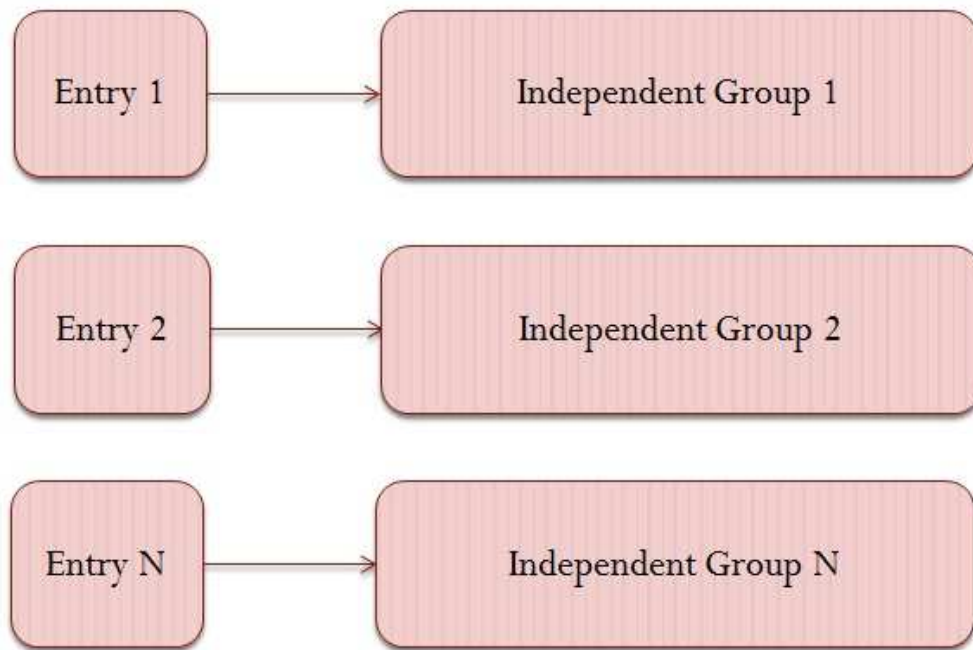


圖 3-10 Multiple Shape 範例

於要如何從 CFG 中建立各個 Shape，可參考以下的演算法：

### Analyze and build Simple and Loop Shape

```
BlockSet NextEntries = NULL
```

```
Shape Ret = NULL
```

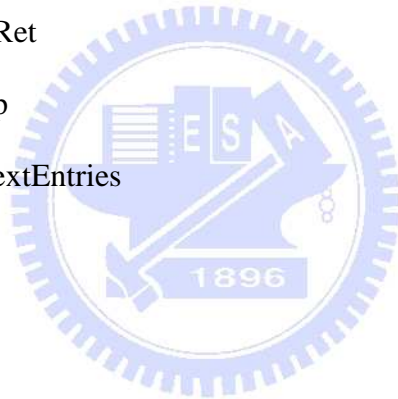
```
while(1){
```

```
    if (The number of blocks in Entries is 1){
```

```

Block Curr = Entries
if (The other in-edges of Curr == 0)
    Shape Temp = MakeSimple(Blocks, Curr, NextEntries)
else
    Shape Temp = MakeLoop(Blocks, Entries, NextEntries)
if (Prev != NULL)
    Prev.Next = Temp
if (Ret == NULL)
    Ret = Temp
if (NextEntries.size() == 0)
    return Ret
Prev = Temp
Entries = NextEntries
continue
}
.....

```



演算法的前半段是在分析該串 **Basic Block** 串是否為 **Simple** 或 **Loop Shape**，如果結果為是，則呼叫對應的 **Shape** 建立函式。其中，**Blocks** 代表著分析範圍中所有的 **Block** 集合(**BlocksSet**)，初始值為函式中所有 **Basic Block**；**Entries** 代表著分析範圍中的 **Entry Block** 集合，初始值是函式的 **Entry Block**；**Prev** 則是代表分析範圍的前一個 **Shape**，初始值為空。而所呼叫的函式 **MakeSimple** 以及 **MakeLoop** 演算法如下：

```
Shape MakeSimple(Blocks, Inner, NextEntries){
```

```

SimpleShape Simple
Simple.Inner = Inner
if (Blocks.size() > 1){
    Blocks.erase(Inner)
    Add Inner's next block to NextEntries if it is in Blocks.
}
return Simple
}

```

```

Shape MakeLoop(Blocks, Entries, NextEntries){
    BlockSet InnerBlocks
    Search all blocks's previous block B1 of Blocks from Entries, then insert B1
to InnerBlocks and erase from Blocks.
    Search all blocks's next block B2 of InnerBlocks
    if(InnerBlocks.find(B2) == NULL)
        NextEntries.insert(B2)
    LoopShape Loop
    Shape Inner = Process(InnerBlocks, Entries, NULL)
    Loop.Inner = Inner
    return Loop
}

```

當 Entries 中只有一個 Basic Block 時，代表 Blocks 應該建立成 Simple 或 Loop Shape，此時再對 Entries 中唯一的 Basic Block 做檢驗。根據此 Basic Block 是否有多餘的 in-edge，而呼叫 MakeSimple 函式建立 Simple Shape 或 MakeLoop 函式

建立 Loop Shape，並進一步分析新建立 Shape 內部的 Basic Block 是否能夠建立子 Shape。最後，再取得新的 Entries。

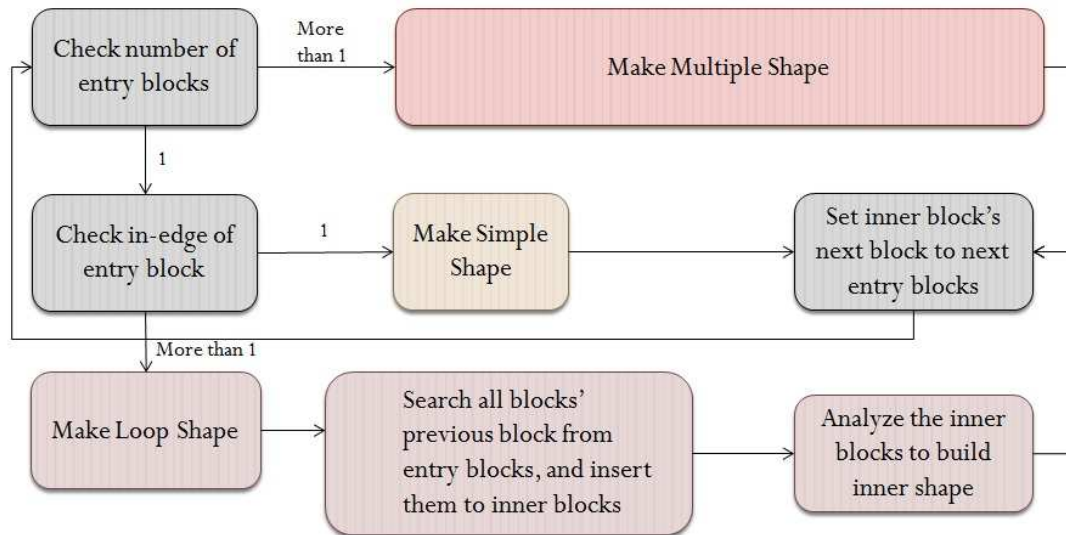


圖 3-11 Reloop 運行步驟 1

### Analyze and build Multiple Shape

```

IG = FindIndependentGroups(Blocks, Entries)
if (IG has at least 1 child block set){
    for(every child blockset G1 of IG){
        Block Entry = the entry block of G1
        Search all in-edges of Entry
        if(Some in-edge of Entry is reached from outside of G1)
            IG.erase(G1)
    }
}
if (IG still has at least 1 child block set)
  
```



```

        Shape Temp = MakeMultiple(Blocks, Entries, IG, Prev, NextEntries)
    else
        Shape Temp = MakeLoop(Blocks, Entries, NextEntries)
    if (Prev != NULL)
        Prev.Next = Temp
    if (Ret == NULL)
        Ret = Temp
    if (NextEntries.size() == 0)
        return Ret

    Prev = Temp
    Entries = NextEntries
    continue
}

```



至於後半段是嘗試將 Basic Block 串建立為 Multiple Shape。其中，IG 代表著各個 Entry Block 底下的子 Block 集的總集合。而 FindIndependentGroups 函式則是根據各個 Entry Block 來搜尋底下的子 Block 集。至於 MakeMultiple 函式演算法如下：

```

Shape MakeMultiple(Blocks, Entries, IG, Prev, NextEntries){
    MultipleShape Multiple
    BlockSet CurrEntries
    for(every child block set G1 of IG){
        Block CurrEntry = the entry block of G1
        BlockSet CurrBlocks = the total blocks of G1
    }
}

```

```

BlockSet CurrEntries
CurrEntries.insert(CurrEntry)
for(every block B1 of CurrBlocks ){
    Blocks.erase(B1)
    Search all blocks's next block B2 from B1
    if(CurrBlocks .find(B2) == NULL)
        NextEntries.insert(B2)
    }
    Multiple.InnerMap[CurrEntry] = Process(CurrBlocks, CurrEntries,
NULL)
}
for (every block B3 of Entries){
    if (B3 isn't the entry block of IG)
        NextEntries.insert(B3)
    }
return Multiple
}

```

當 Entries 中有複數個 Basic Block 時，代表 Blocks 可能可以建立成 Multiple Shape。在這種狀況下，首先要根據每個 Entry Block 找到其所屬的子 Block 集，並確認每個 Entry Block 是否的確真的是 Entry Block，而不是別的 Entry Block 底下子 Block 集中的一份子。

如果的確有一個以上的 Entry Block 符合條件，那代表這串 Basic Block 的確可以建立成 Multiple Shape，則呼叫 MakeMultiple 函式建立 Multiple Shape。並進

一步分析新建立 Shape 內部的 Basic Block 是否能夠建立子 Shape。最後，再取得新的 Entries。相反的，如果的確沒有任何一個 Entry Block 符合條件，那就只能將這串 Basic Block 建立成 Loop Shape。

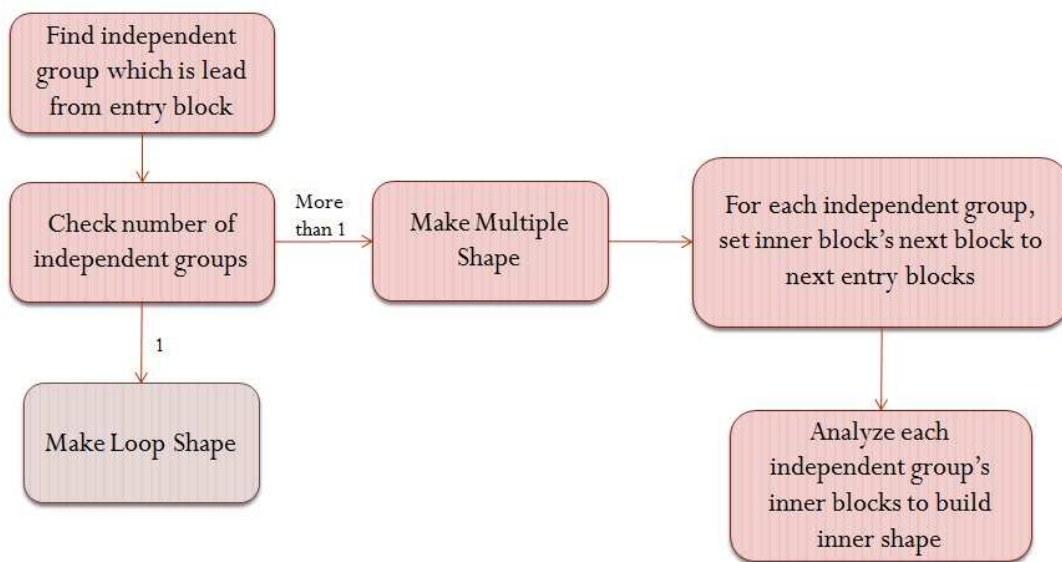
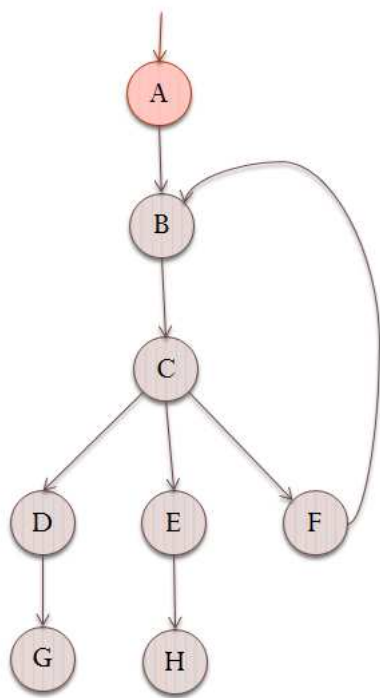


圖 3-12 Reloop 運行步驟 2

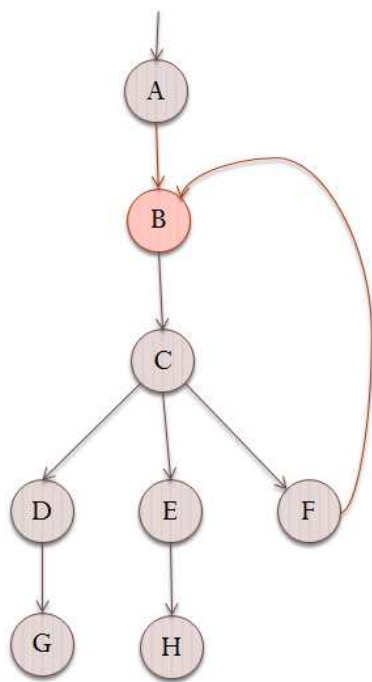
圖 3-13、3-14、3-15 即為一簡單的範例。在圖 3-13 中，一開始 A、B、C、D、E、F、G、H 皆在應分析的 Basic Block 串中，其中 A 為 Entry Block。由於 A 的 in-Edge 只有一條，故建立一 Simple Shape，其內部 Block 只有 A。而內部 Block 中的 Next Block 只有 B，所以新的 Entry Block 為 B，應分析的 Basic Block 串變為 B、C、D、E、F、G、H。



- Block Range : A, B, C, D, E, F, G, H
- Entry Block : A
- In-Edge : 1 → **Simple**
- Next Entry blocks : B
- **Simple Shade 1 : {A}**

圖 3-13 Reloop 運行範例 1

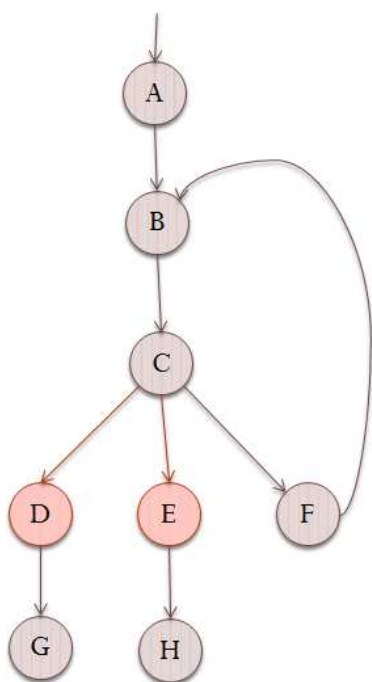
在圖 3-14 中，此時應分析的 Basic Block 串為 B、C、D、E、F、G、H，而 B 為 Entry Block。由於 B 除了由 A branch 過來的 Edge 之外，還有另外一條 in-Edge，故須建立成 Loop Shape。從 B 往回搜尋後，可找到的內部 Block 為 F、C、B(A 不在應分析的 Basic Block 串中)。而內部 Block 中的 Next Block 為 D、E，所以新的 Entry Block 為 D、E，應分析的 Basic Block 串變為 D、E、G、H。



- Block Range : B, C, D, E, F, G, H
- Entry Block : B
- In-Edge : 2 → Loop
- Next Entry blocks : D, E
- Simple Shade 1 : {A}
- Loop Shade 2 : {B, C, F}

圖 3-14 Re loop 運行範例 2

在圖 3-15 中，應分析的 Basic Block 串為 D、E、G、H，而其中 D、E 也同時為 Entry Block。由於 Entry Block 不只一個，故可能可以建立成 Multiple Shape。分別從 D、E 往下搜尋後，可找到兩組子 Block 集 {D、G} 以及 {E、H}。由於 D、E 都不是對方底下子 Block 集中的一份子，故 Multiple Shape 成立。而 Next Block 為空，於是分析完成，可根據 Shape 結構重建迴圈。



- Block Range : D, E, G, H
- Entry Block : D, E → **Multiple**
- Next Entry blocks : NULL
- Simple Shade 1 : {A}
- Loop Shade 2 : {B, C, F}
- **Multiple Shade 3 : {{D, G}, {E, H}}**

圖 3-15 Reloop 運行範例 3

### 3.3.3 Implementation of Instruction translation

在我的後端中，由於必須考慮到 Lua 腳本語言的語法以及發揮其特性，所以在實作 Instruction translation 時添加了許多機制來進行特處理以及優化，以下是詳細的介紹：

#### 3.3.3.1 Pointer variable process and Instruction optimization

由於這兩部分一起構成 Instruction translation 的主體，故放在一起說明，其整體步驟如圖 3-16 所示：

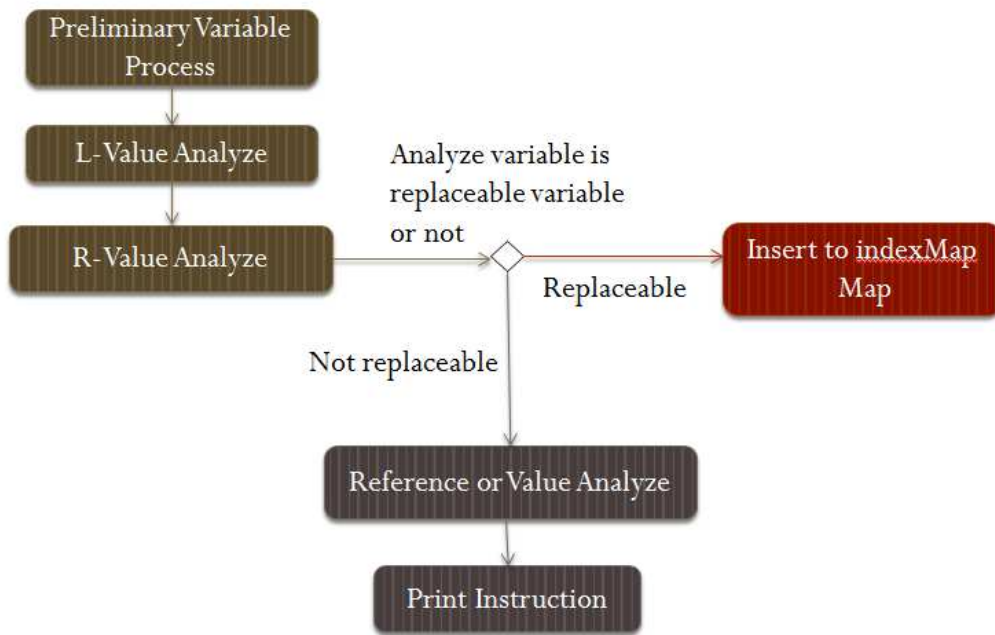
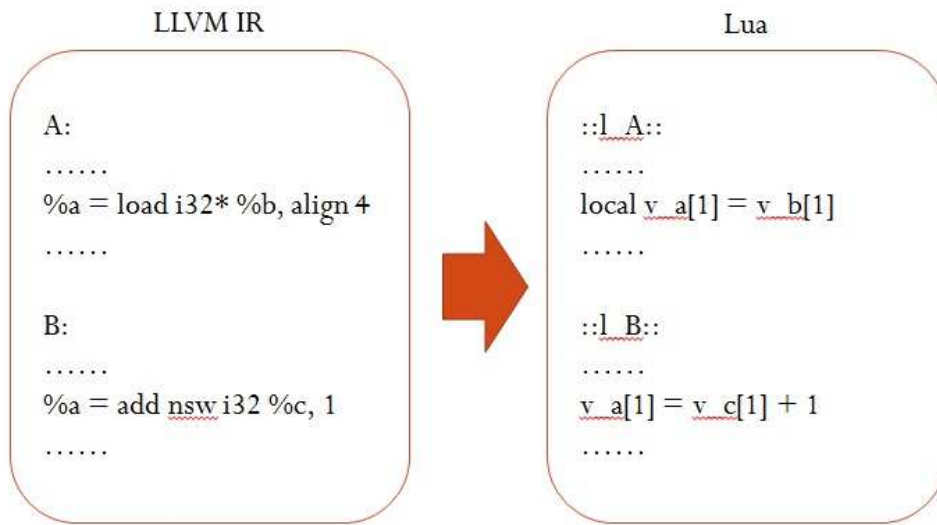


圖 3-16 Instruction translation 執行步驟

一開始的 Preliminary variable process 步驟是為了預先確認變數宣告。由於程式在執行時的指令先後順序，往往與後端的轉譯順序是不相同的。如果到轉譯到此變數才宣告的話，有可能發生如圖 3-17 的問題。

- Translation Order



- Runtime Order

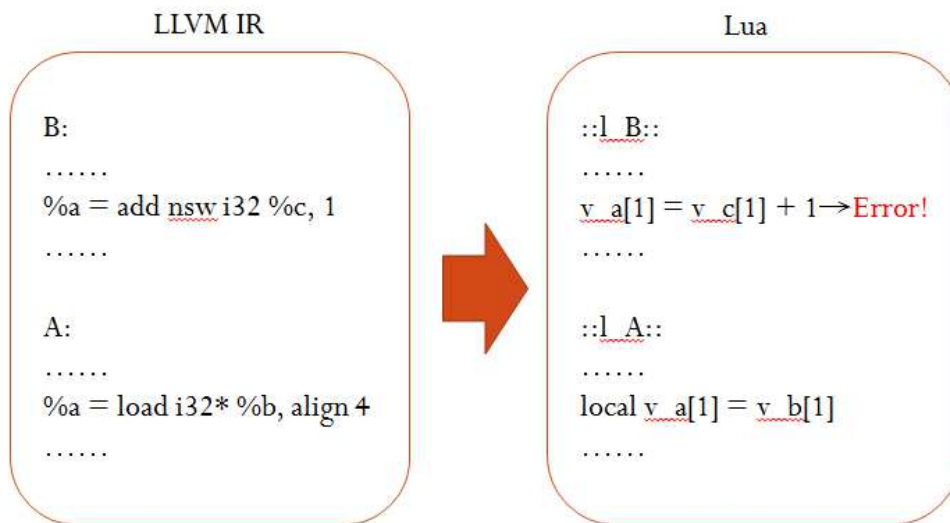


圖 3-17 轉譯與執行順序不同之問題

而在啟用 Preliminary variable process，就能在轉譯指令之前就先確認所需要的變數。這樣之後的轉譯以及 Instruction optimization 步驟都能避免此一問題，如圖 3-18 所示。



## • Runtime Order

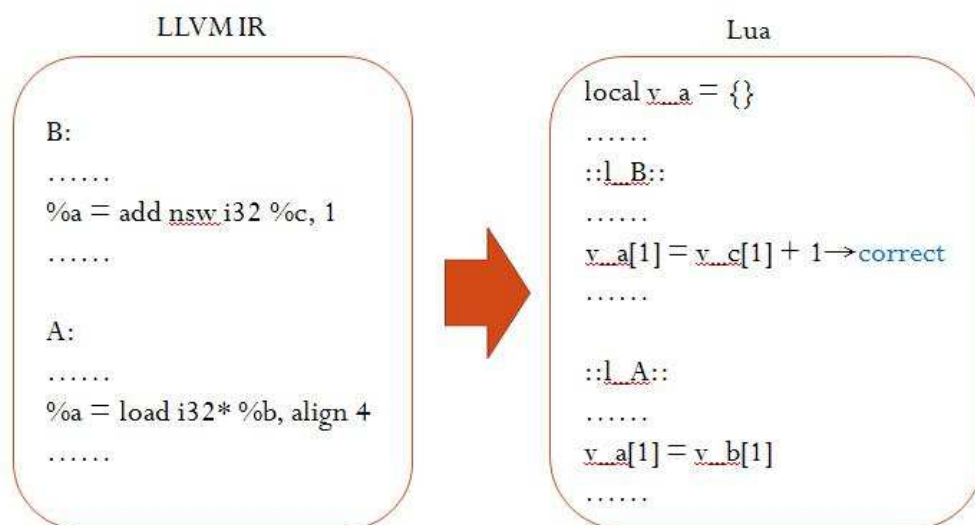


圖 3-18 啟用 Preliminary variable process

在 Preliminary variable process 步驟之後，接著則正式開始進行指令的分析、優化以及轉譯。首先是 L-Value 與 R-Value 的分析，對於大部分的數學運算以及邏輯運算指令，LLVM 中間表示式都會使用新建立的變數作為 L-Value 來儲存運算後的結果，之後再將結果存進真正想要放置的變數裡。會這麼做的原因，應該是考慮到低階語言的暫存器設計。然而對於腳本語言而言，這樣分拆純粹只是造成執行速度的變慢以及 statement 數增加而已。因此當遇到這類的指令時，我並不會直接進行轉譯，反而會建立 indexMap 這個 Map 結構，來記錄指令的 L-Value 以及 R-Value。相對的，在轉譯其他指令時，我也必須先根據 indexMap 檢查其 L-Value 以及 R-Value 是否須要被取代。

在 Replaceable 的檢測過後，則要進行最後的 By Reference or Value 的分析。因為 Lua 腳本語言的變數一般只能以 By Value 方式傳遞，只有宣告為 Table(類似 Array)型態的變數才能以 By Reference 方式傳遞。為了解決此一問題，我對此的處理方法，就是將變數宣告為大小為 1 的 Table，然後根據指令來決定是用 By

Reference 還是 By Value 方式傳值。這樣子固然可以解決此問題，但相對的，也會造成不易區分一般變數以及 Table 變數的問題。因此，我配合之前的 Preliminary variable process 步驟，建立 varMap 這個 Map 結構，來記錄指令的型態。

圖 3-19、3-20 以及 3-21 即為一簡單的例子，而表 3-1 則是對應產生的 indexMap 以及 varMap 內容。其中，紅色的指令皆被判定為可省略的指令，不轉譯而存進 indexMap 中。由此可見，此步驟的確可以減少無謂的指令數，進而達到改善執行效率的目的。

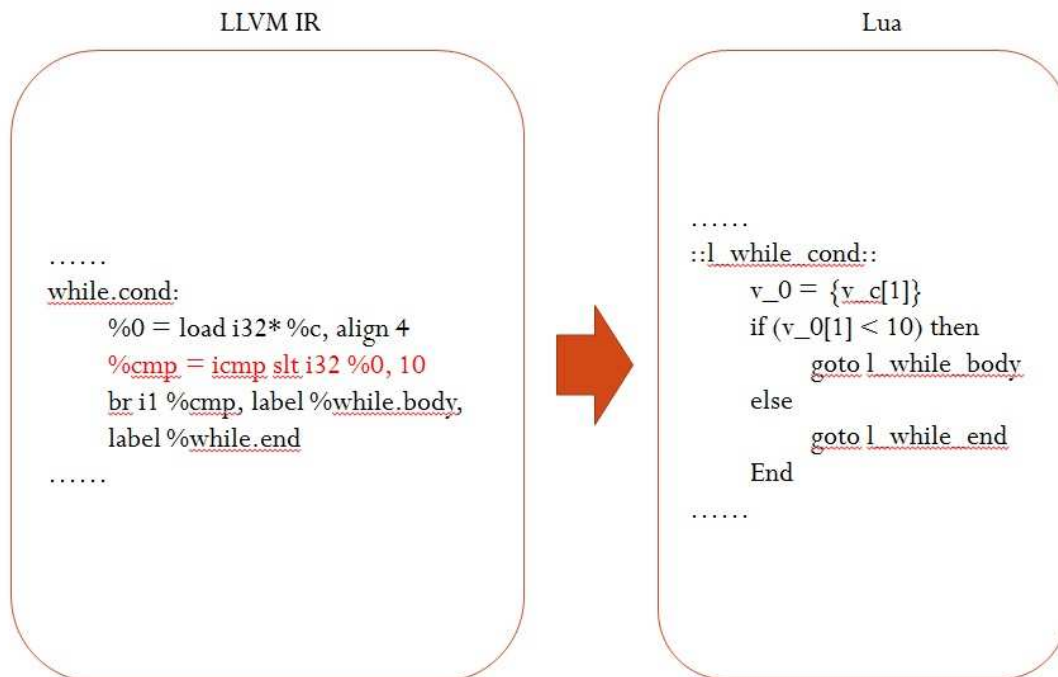


圖 3-1 Instruction translation 範例 1

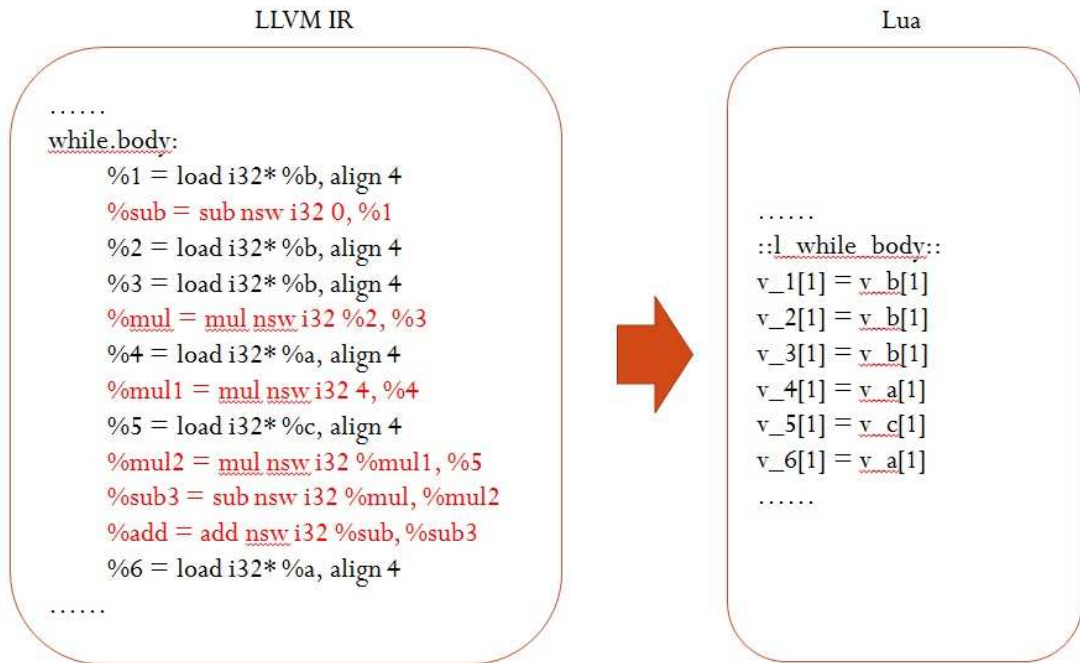


圖 3-2 Instruction translation 範例 2

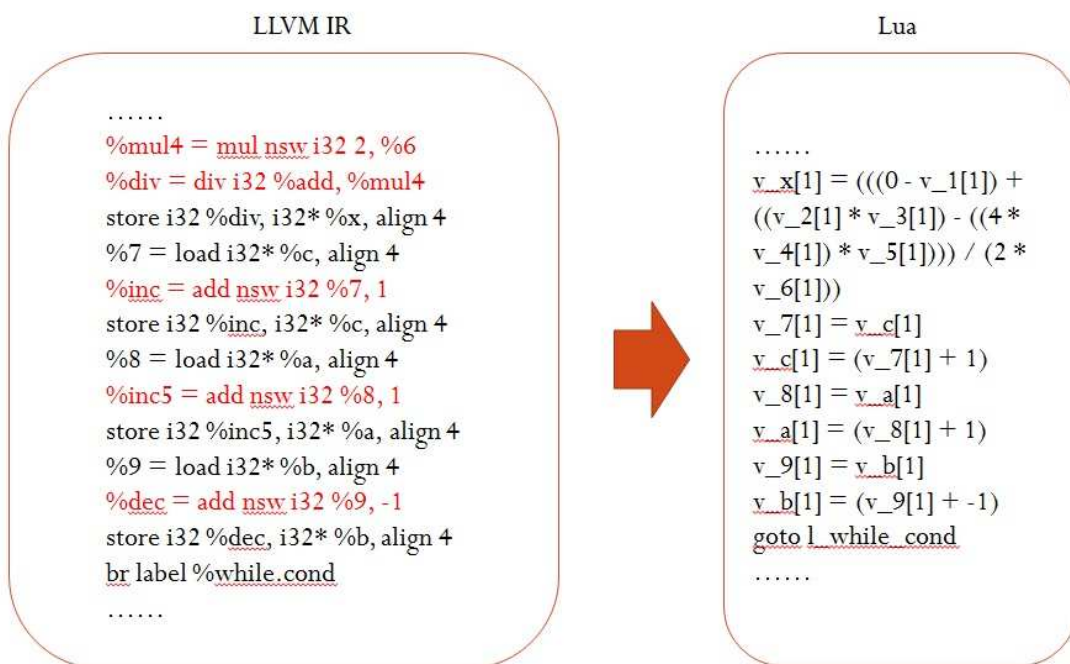


圖 3-23 Instruction translation 範例 3

v_cmp	$(v\_0[1] < 10) \text{ and } 1 \text{ or } 0$	v_0	VAR
v_sub	$(0 - v\_1[1])$	v_1	VAR
v_mul	$(v\_2[1] * v\_3[1])$	v_2	VAR
v_mul1	$(4 * v\_4[1])$	v_3	VAR
v_mul2	$((4 * v\_4[1]) * v\_5[1])$	v_4	VAR
v_sub3	$((v\_2[1] * v\_3[1]) - ((4 * v\_4[1]) * v\_5[1]))$	v_5	VAR
v_add	$((0 - v\_1[1]) + ((v\_2[1] * v\_3[1]) - ((4 * v\_4[1]) * v\_5[1])))$	v_6	VAR
v_mul4	$(2 * v\_6[1])$	v_7	VAR
v_div	$((((0 - v\_1[1]) + ((v\_2[1] * v\_3[1]) - ((4 * v\_4[1]) * v\_5[1]))) / (2 * v\_6[1])))$	v_c	VAR
		v_8	VAR
v_inc	$(v\_7[1] + 1)$	v_a	VAR
v_inc5	$(v\_8[1] + 1)$	v_9	VAR
v_dec	$(v\_9[1] + -1)$	v_b	VAR

表 3-1 indexMap 以及 varMap 結構

### 3.3.3.2 Special instruction process

#### PHI instruction process

在 LLVM 中間表示式的眾多指令中，PHI 指令算是十分特別的一種指令。它的所賦予 L-Value 的值，會根據 Control Flow 而變動。而這種指令，在一般腳本語言中是沒有的。

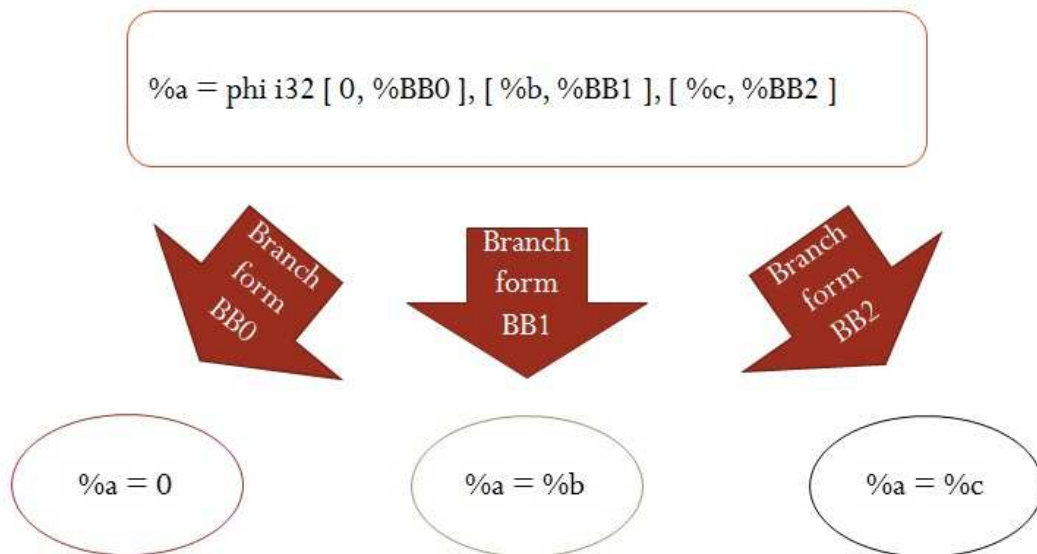


圖 3-22 PHI 指令

因此，在轉譯此指令時，我採用 SSA Form Destruction 方式來處理此問題。首先，我先分析 PHI 指令的 R-Value，確認它可能由哪幾個 Basic Block Branch 過來，之後再將 PHI 指令分拆成數個指令，並搬移到來源 Basic Block 的尾端，如圖 3-23 與圖 3-24。

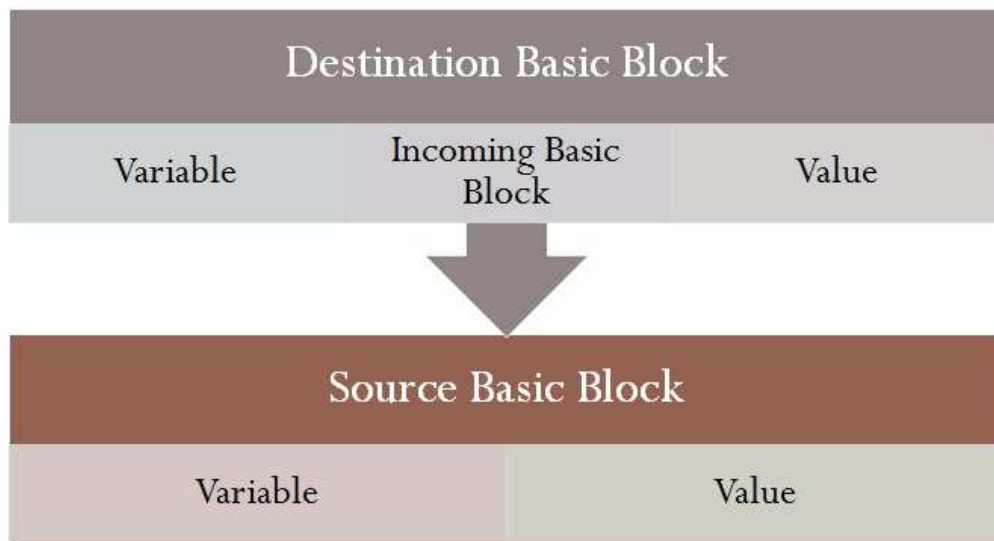


圖 3-23 分解 PHI 指令

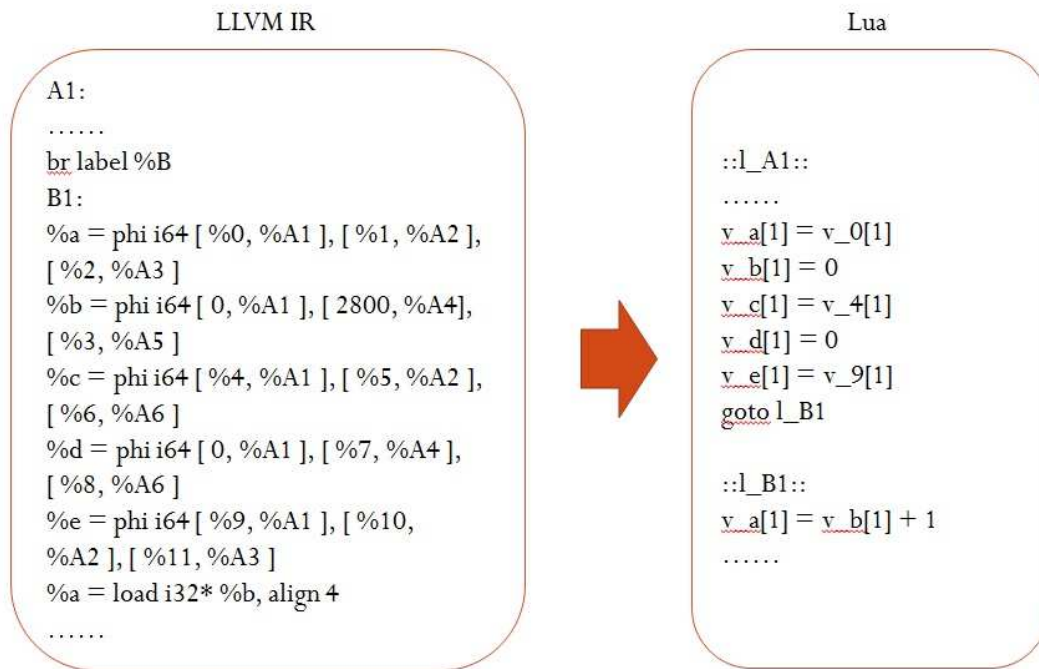


圖 3-24 轉譯 PHI 指令範例

採用這種方法，最大的優點在於可以減少轉譯後所需的 if statement 數量，而執行時的需要執行的 statement 數也不會變多。雖然整體的 statement 數會有略微的增加，但以執行效率來看，會比單純直接用 if statement 進行轉譯要好的多。

### Boolean to Integer translation

對於很多的程式語言而言，布林值跟數值在一定程度上是可以互相轉變的。以 C 語言為例子，當用數值來進行布林判斷時，非零值會被判斷成 True，而零會被判斷成 False。相對的，當布林值轉換成數值時，一般而言，True 會轉換成 1，而 False 會轉換成 0。

然而，Lua 腳本語言是不允許布林值跟數值進行直接轉換的，這使得當我遇到 LLVM 中間表示式進行布林值跟數值轉換，特別是布林值轉換成數值的時候，就必須進行特殊處理。

幸好，Lua 腳本語言遵守以下的邏輯判斷規則：

- 對 And 指令而言，如果指令的第一個變數其值為 False，則 And 指令直接傳回 False；否則，And 指令直接傳回第二個變數的值，不論其為何。
- 對 Or 指令而言，如果指令的第一個變數不為 False，則 Or 指令直接傳回第一個變數的值；否則，Or 指令直接傳回第二個變數的值，不論其為何。

因此，我可以在原本的布林值後面加上「and 1 or 0」來進行轉換。

- 當原本的布林值為 True 時： $\text{True and 1 or 0} \rightarrow 1 \text{ or 0} \rightarrow 1$
- 當原本的布林值為 False 時： $\text{False and 1 or 0} \rightarrow \text{False or 0} \rightarrow 0$




## 第四章 成果與分析

### 4.1 環境設置

因為本篇論文主要在探討 LLVM 後端之相關轉譯技術，所以在這個章節，我將使用 Benchmark 進行測試，並提出實驗數據來顯示我所提出的轉譯技術之效果。

由於我的轉譯器基本上是使用 Linux 來進行開發，而且考慮到 Lua 腳本語言主要搭配的程式語言是 C 與 C++。因此我選擇 Clang 以及 Linux 做為測試的環境，詳細的資料如表 4-1 所示：



Title	Description
Benchmark	EEMBC benchmarks
Test Machine	48 core x86_64 machine
Test OS	Linux Debian 3.2.35-2
Frontend for benchmark	Clang-3.0
LLVM	LLVM-3.0
CPU	AMD Opteron™ Processor 6172 , 2.1 GHz
Memory Size	48GB

表 4 -1 測試環境



## 4.2 實驗方法

對於我的轉譯器而言，所需要的測試程式並不是已經編譯完成的 Binary Code，而是只編譯成中間表示式的半成品，因此原本 EEMBC 的 makefile 並不適用。除了使用身為前端的 Clang 之外，還必須使用 llvm-ld 以及 llvm-dis 進行相關的連結以及解碼工作，最後再使用 llc 啟動我的後端進行轉譯。

## 4.3 成果展現

### 4.3.1 啟用 Instruction Optimization 造成之影響

圖 4-1 以及圖 4-2 分別是啟用 Instruction Optimization 機制後對 Statement 數以及執行速度之影響。其中橫軸為 Benchmark，圖 4-1 之縱軸為 Statement 數之比例，圖 4-2 之縱軸為執行時間之比例。

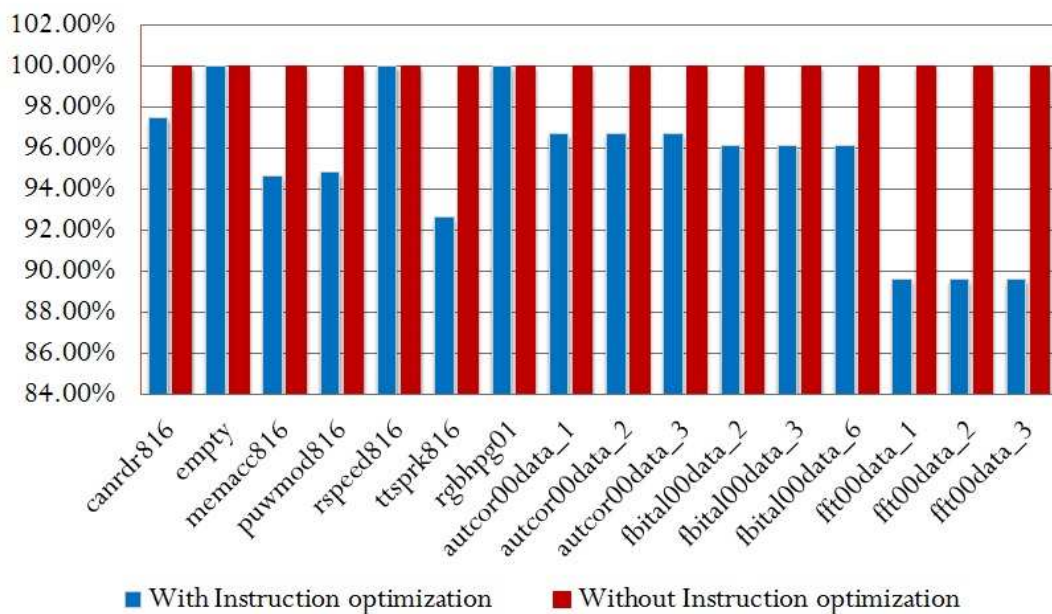


圖 4-1 有無 Instruction Optimization 機制對 Statement 數之影響

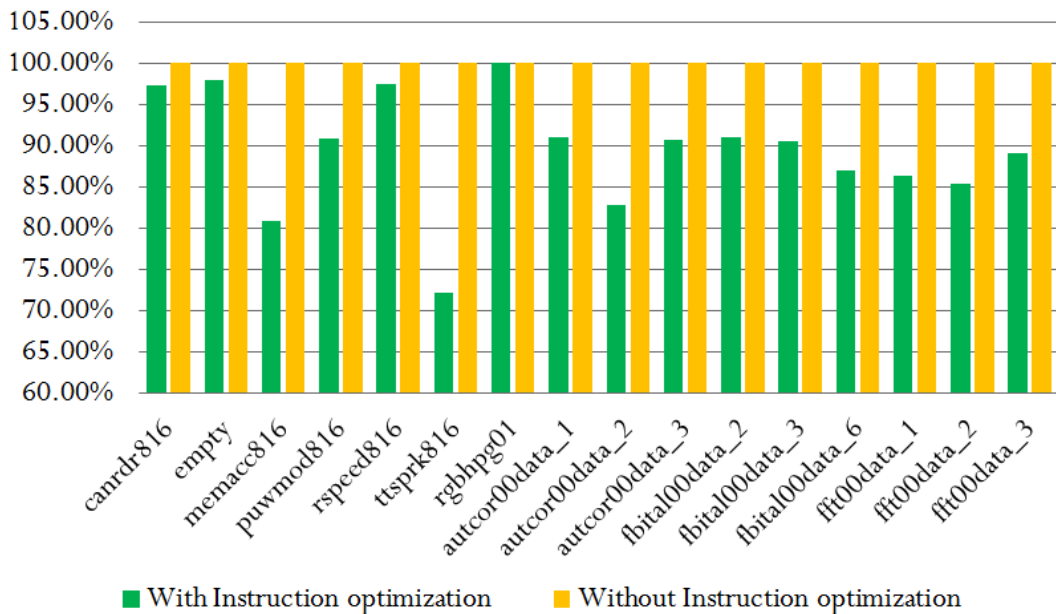


圖 4-2 有無 Instruction Optimization 機制對執行時間之影響

由圖 4-1 可以得知，啟用 Instruction Optimization 機制後，隨著每個 Benchmark 之間數學以及邏輯運算所佔的比例不同，平均能夠減少 5% 的 Statement 數，最高甚至能減少 10% 以上的 Statement。而由圖 4-2 更顯示出在執行時間方面，Instruction Optimization 可以造成平均 11% 的優化。

### 4.3.2 啟用 Loop Recovery 造成之影響

圖 4-3 是 Loop Recovery 機制所能影響的 Statement 數比例。平均而言，有 44% 的 Statement 會重建為迴圈結構的一部分。同時根據圖 4-5 可得知，Loop Recovery 機制所能影響比例較少的測試程式，對執行時間的影響也相對較少。

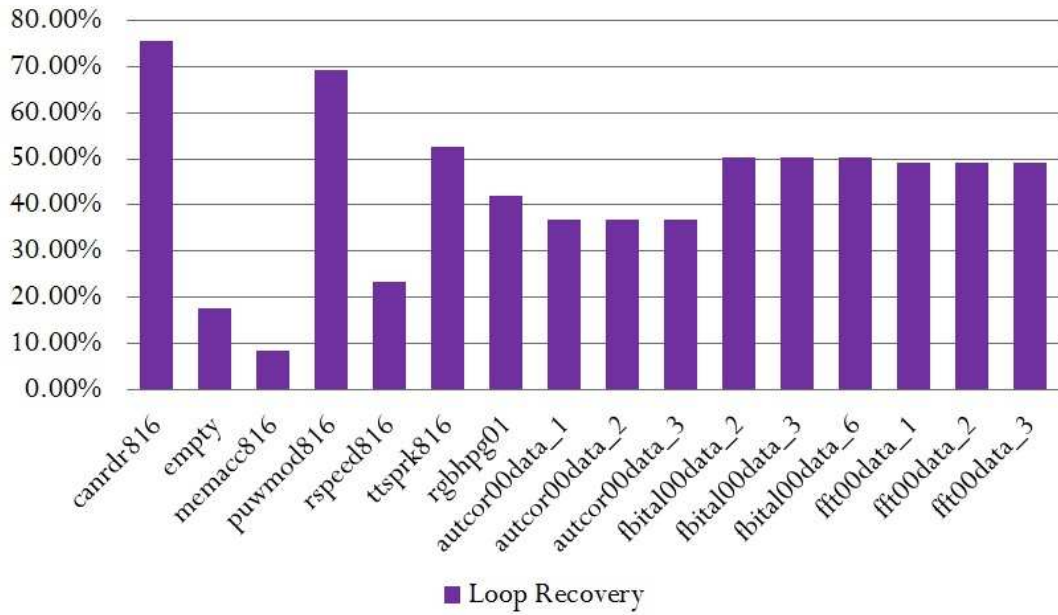


圖 4-3 Loop Recovery 機制影響範圍

圖 4-4 以及圖 4-5 則是啟用 Loop Recovery 機制後對 Statement 數以及執行速度之影響。其中橫軸為 Benchmark，圖 4-4 之縱軸為 Statement 數之比例，圖 4-5 之縱軸為執行時間之比例。

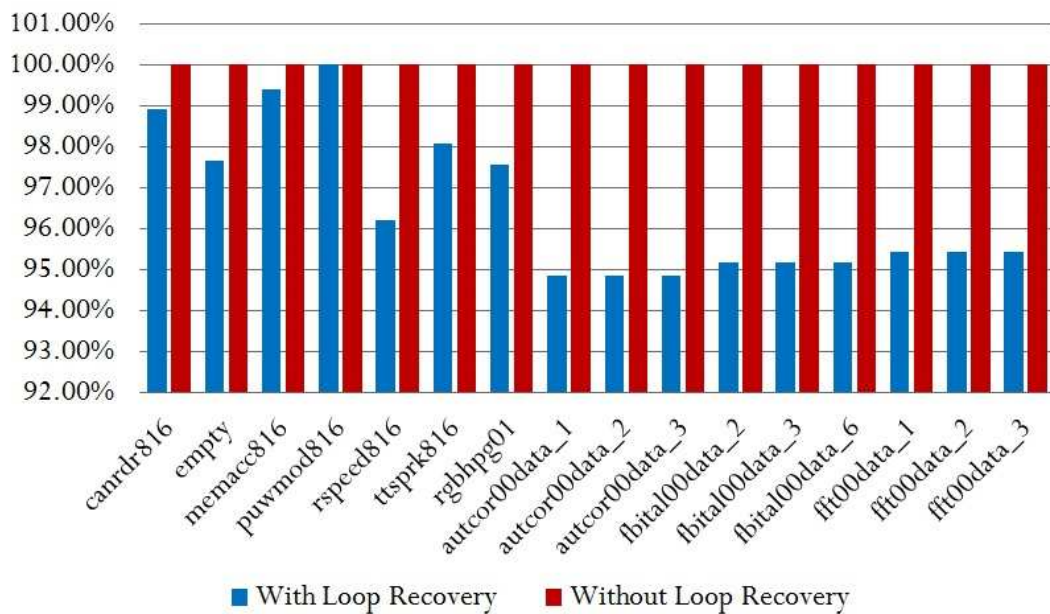


圖 4-4 有無 Loop Recover 機制對 Statement 數之影響

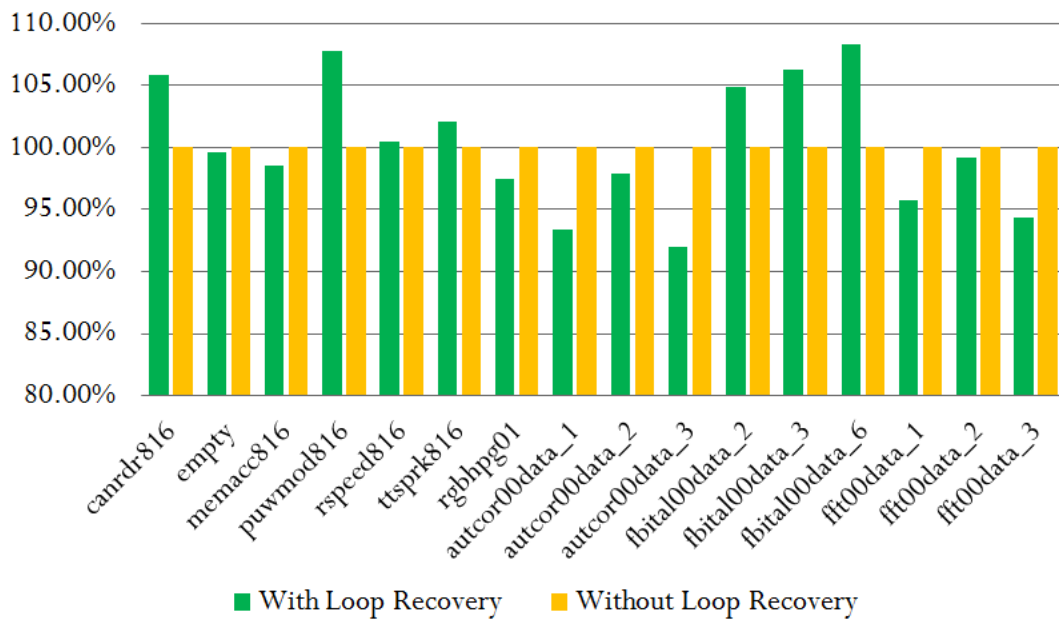


圖 4-5 有無 Loop Recover 機制對執行時間之影響

由於 Loop Recover 機制主要是著重在針對可讀性的優化，所以對於執行速度的影響並不是很正面。不過整體而言，Loop Recover 機制仍然可以為 Statement 數帶來平均 4%的優化。同時也會增加 0.25%的執行時間，不過相較於可讀性的改善，這可以算是可以接受的損失。

#### 4.4 整體成果

綜合前述的轉譯優化技術，我不僅提昇了轉譯後程式碼的可讀性，對於 Statement 數以及執行速度方面也造成了不少的改善。整體而言，在啟用所有的優化機制之後，正如同圖4-6以及圖4-7所示，平均上我能夠減少8%的 Statement，並且提昇 10%的執行速度。

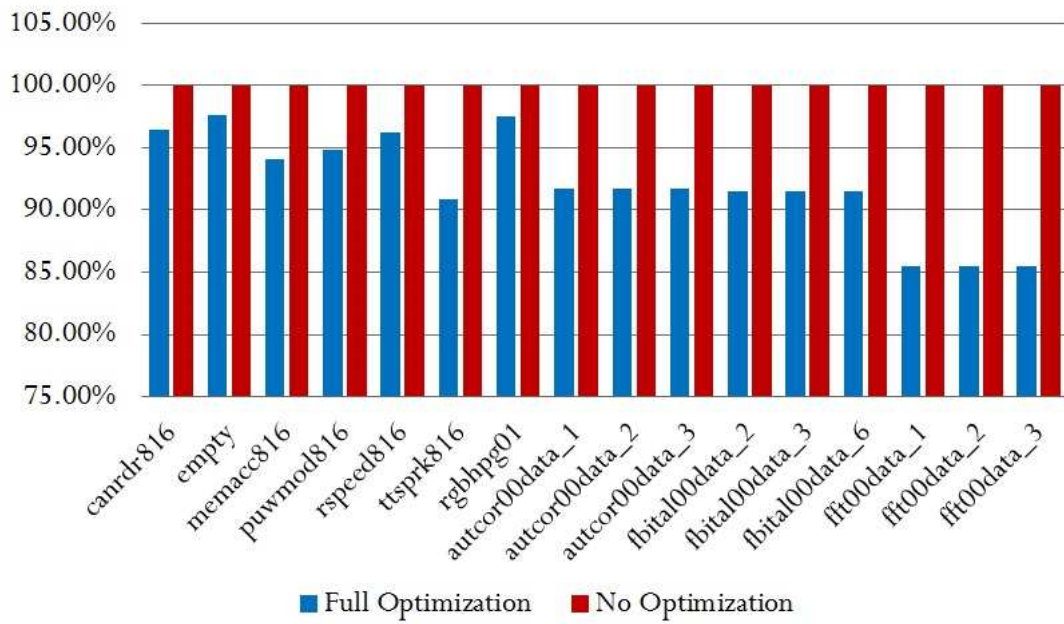


圖 4-6 整體優化機制對 Statement 數之影響

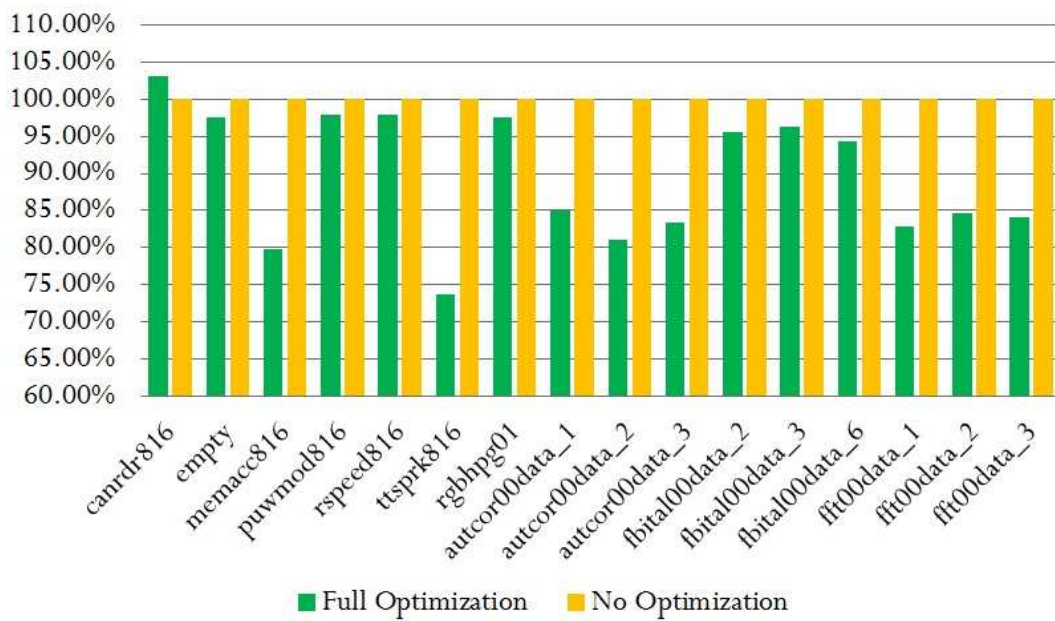


圖 4-7 整體優化機制對執行時間之影響

## 4.5 Lua 後端與 Emscripten 的比較

最後則是與 Emscripten 之間的比較，圖 4-8 是啟用 Loop Recovery 機制後對 Statement 數之影響。其中橫軸為 Benchmark，縱軸為 Statement 數。

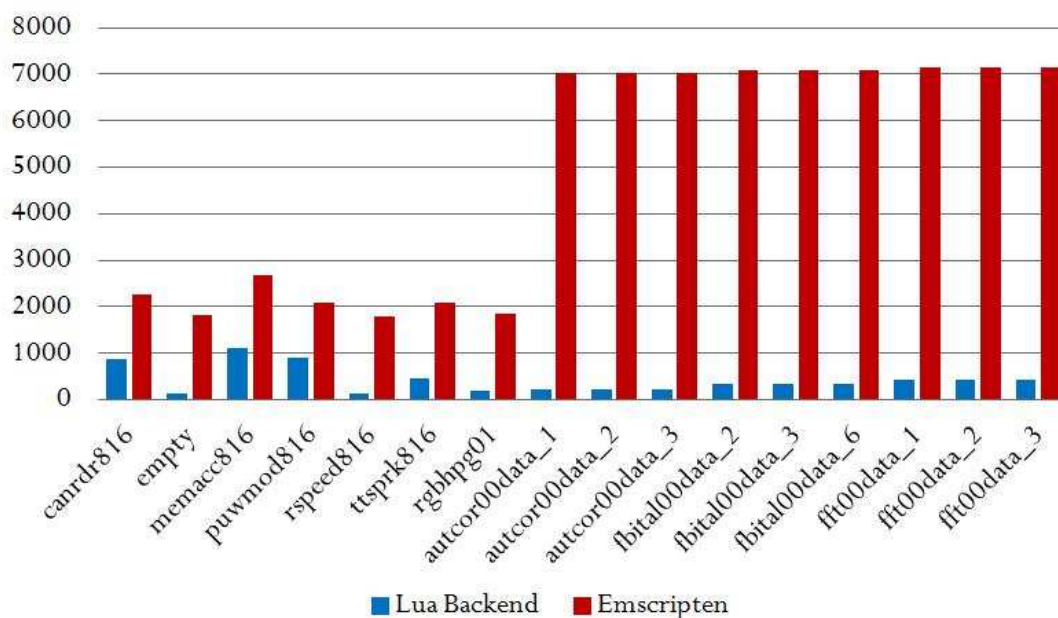


圖 4-8 Lua 後端與 Emscripten 對 Statement 數之比較

由於 Emscripten 並未針對指令進行優化，而且具有完整的 System function 支援，因此 Statement 數遠較我的後端要來的多，特別是有較多運算指令的 Benchmark。

至於執行速度的部份，在實驗中，Emscripten 要比我的後端要來得快，但是 Emscripten 在轉譯時會產生一個問題，就是有許多 Benchmark 並不能正常執行。Emscripten 轉譯出來的 JavaScript 程式碼會有錯誤，再加上執行速度會受到虛擬機器方面很大的影響，因此無法取得兩者有效的執行速度比較資訊。

# 第五章 結論與未來展望

## 5.1 結論

在這篇論文中，我利用 LLVM 系統設計了一個新型的後端，來實現 C 語言轉譯成 Lua 腳本語言的機制。與一般的後端不同，我的後端成功的實現了低階語言轉譯成腳本語言的行為，並且增添了許多優化機制讓轉譯後的程式更加的符合腳本語言程式應有的形態，以及節省了許多低階語言轉譯成高階語言時會產生的不必要的浪費。

同時，藉由此後端與 LLVM 系統，我能夠達成 Lua 腳本語言與 C、C++，甚至更多的語言之間自由轉譯。如此一來，要將原本由其他語言所撰寫的程式嵌入 C 語言中，將會變得更為容易。特別是對於使用 C 語言搭配 Lua 腳本語言的軟體而言，更可以大幅的減少變更架構時所需付出的成本。這對於常常使用 C 語言搭配 Lua 腳本語言的商業軟體來說，可說是一大福音。

## 5.2 未來發展與改進方向

儘管我成功的實作了由低階語言轉譯成高階語言的 LLVM 後端，然而以目前而言，我的後端仍然有所限制。

首先就是針對 LLVM 的 `getelementptr` 指令的支援有限。受制於 Lua 腳本語言的語法限制，有些 `getelementptr` 指令並不能成功的轉譯，其中最明顯的例子即是 `getelementptr` 指令允許其索引值為負數，然而這在 Lua 腳本語言中，是不允許的。因此我希望在未來，能夠處理此問題。

其次則是對 **System function** 的支援不足，由於時間有限，我尚未對 **LLVM** 所有能使用的 **C** 函式庫進行完整的轉譯，只有對目前有需要的進行處理。故在以後會對此部分進行補完。

最後則是可讀性進一步的強化。在實作了 **Loop Recovery** 以及 **Function Generalize** 之後，程式的可讀性固然上昇不少，但是轉譯後的程式碼依然有不少改善的空間。因此如果能夠將轉譯後的程式碼進一步優化，甚至達到接近物件導向程式的程度，那麼對於程式的可讀性而言，無疑會有更好的改善。





## 參考文獻

- [1] Wikipedia,  
<http://zh.wikipedia.org/wiki/%E8%84%9A%E6%9C%AC%E8%AF%AD%E8%A8%80>
- [2] John K. Ousterhout. “Scripting: Higher Level Programming for the 21st Century”,  
IEEE Computer, March 1998.
- [3] “About Lua” , <http://www.lua.org/about.html>
- [4] Wikipedia, <http://en.wikipedia.org/wiki/LLVM>
- [5] “Writing an LLVM Backend” , <http://www.lua.org/about.html>
- [6] Alon Zakai , "Emscripten: An LLVM-to-JavaScript Compiler", ACM, October 2011
- [7] kripken/Emscripten Wiki , <https://github.com/kripken/emscripten/wiki>
- [8] dmlap/llvm-js-backend , <https://github.com/dmlap/llvm-js-backend>
- [9] David A Roberts , LLJVM , <http://da.vidr.cc/projects/lljvm/>
- [10] davidar/lljvm , <https://github.com/davidar/lljvm>
- [11] LLVM API Documentation , <http://llvm.org/docs/doxygen/html/index.html>
- [12] llvm-lua , <http://code.google.com/p/llvm-lua/>
- [13] The LLVM Target-Independent Code Generator , <http://llvm.org/docs/CodeGen-rator.html#abstract-target-description>
- [14] ViewVC llvm-project , <http://llvm.org/viewvc/llvm-project/>
- [15] Scala LLVM Emitter Module , <http://code.google.com/p/slem/>
- [16] LLVM Language Reference Manual , <http://llvm.org/docs/LangRef.html>
- [17] Lua 5.2 Reference Manual , <http://www.lua.org/manual/5.2/>
- [18] Programming in Lua , <http://www.lua.org/pil/contents.html#P1>
- [19] lua-users.org , <http://lua-users.org/>

