Simple Merge Return Pass

A Compiler Construction Course Project

Konstantin Klima 476/2018

Faculty of Mathematics, University of Belgrade

31.10.2025.

Introduction

- *mergereturn* - standard LLVM optimization pass
- Merging multiple return points into one
- Goal is to improve readability of generated IR code as well as further analyses and optimizations
- We recognize three cases to handle:
  - Multiple unreachable instructions - they arise as a result of exceptions, instructions that interrupt the flow of execution
  - Multiple return instructions with void return type
  - Multiple return instructions with non-void return type

Algorithm

Algorithm Simple Merge Return Pass

unrBlocks ← array of blocks terminated by unreachable instruction
retBlocks ← array of blocks terminated by return instruction

for each *basicBlock* in function *F* **do**
  if terminating instruction of *basicBlock* is unreachable **then**
    Add *basicBlock* to *unrBlocks*
  if terminating instruction of *basicBlock* is ret **then**
    Add *basicBlock* to *retBlocks*

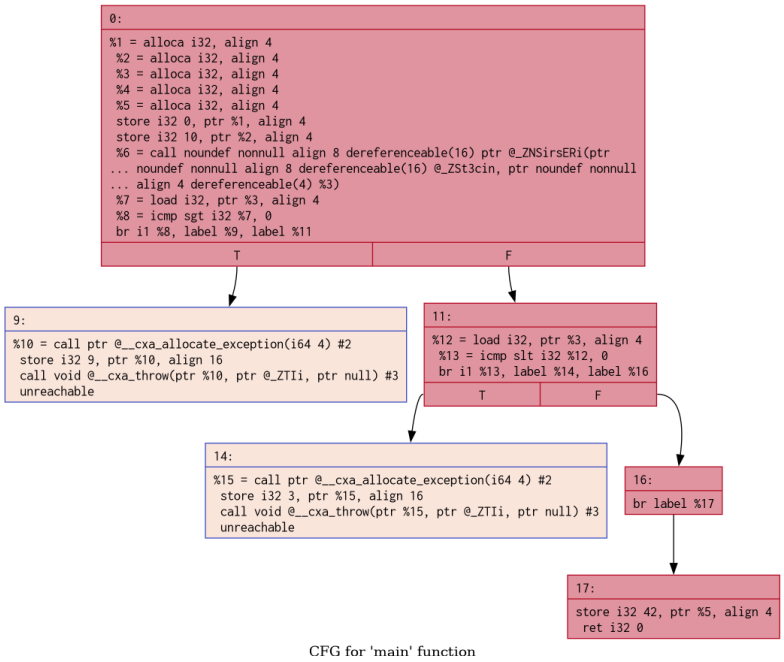
if $|unrBlocks| > 1$ **then**
  create new block *unrBlock* with Unreachable instruction
  for each *basicBlock* in *unrBlocks* **do**
    replace terminating instruction in *basicBlock* with branch to *unrBlock*

if $|retBlocks| > 1$ **then**
  create new block *retBlock* with return instruction
  if return type of *F* is not void **then**
    create new phi instruction
    modify return instruction in *retBlock* to load value from phi
  for each *basicBlock* in *retBlocks* **do**
    replace return instruction in *basicBlock* with branch to *retBlock*

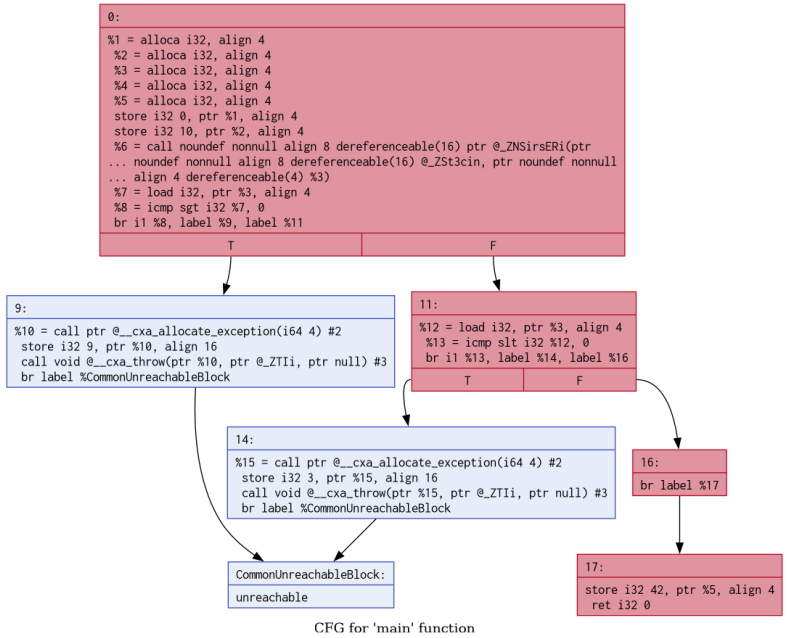
Notes

- The implemented optimization was tested using the LLVM New Pass Manager - a legacy version was also added but could not be tested due to limitations of the local environment.
- The IR for the example with multiple return instructions was manually modified because the frontend emitted already optimized IR with merged return instructions (this was not the case for unreachable examples)

Example 1: Multiple Unreachable Instructions

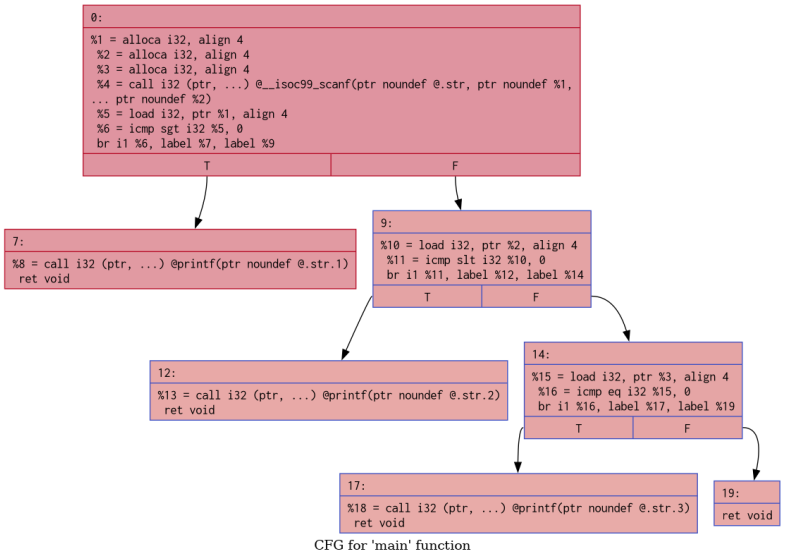


(a) Before optimization

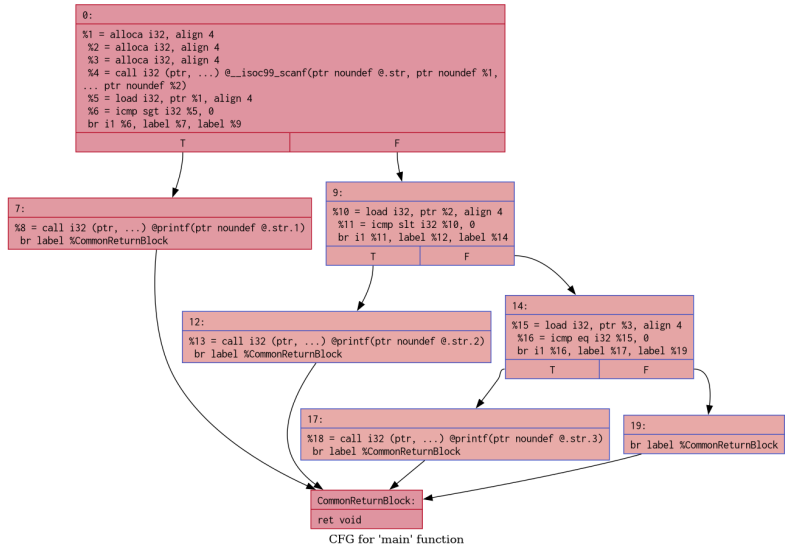


(b) After optimization

Example 2: Multiple Return Instructions

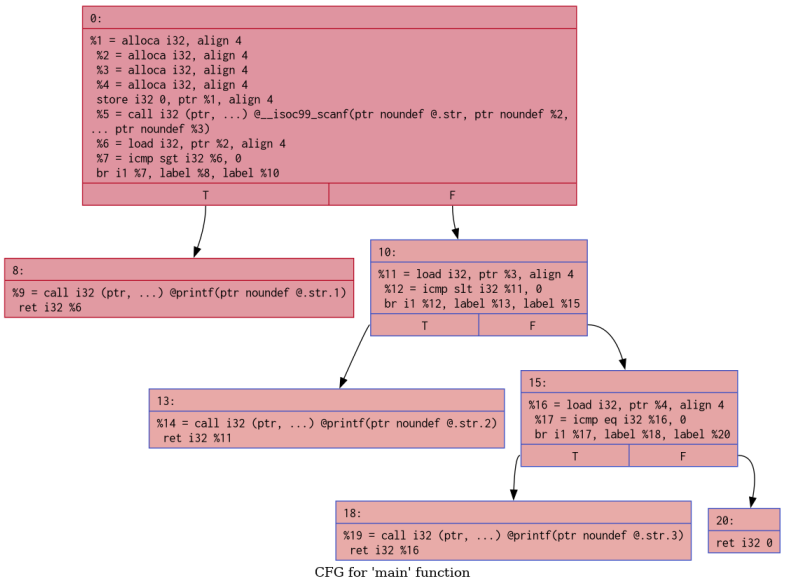


(c) Before optimization

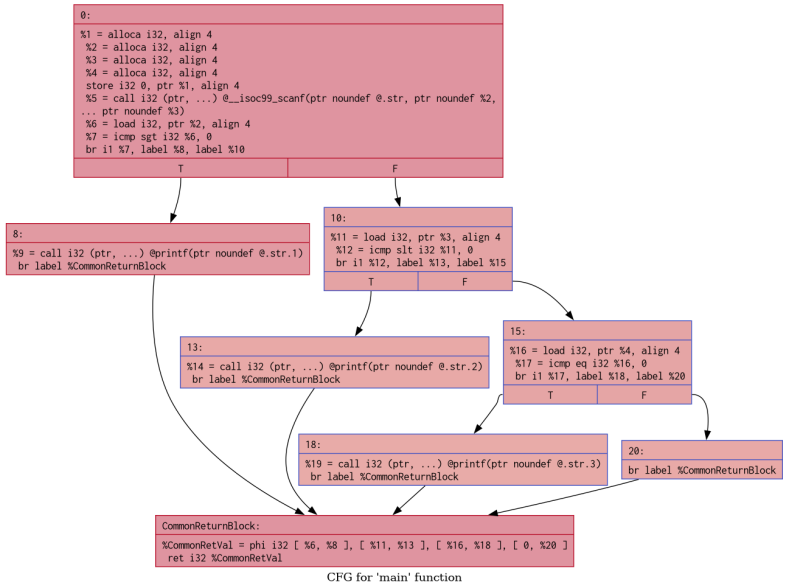


(d) After optimization

Example 3: Multiple Return Instructions with Non-Void Return Type



(e) Before optimization



(f) After optimization

Conclusion

- It is necessary to pay attention to different return types and ensure they are properly propagated when replacing return instructions
- The optimization should also handle unreachable instructions
- Since this optimization changes the CFG, it is necessary to repeat previously performed analyses that rely on the CFG