Гибкий подход к подъёму LLVM MIR кода в SSA форму LLVM IR

(https://github.com/ajlekcahdp4/llvm-bleach)

Романов Александр

Бинарные трансляторы

Обычные

- Специализированное низкоуровневое представление
- Сложная поддержка новых целевых архитектур
- Сопостовление регистров
- Сопоставление инструкций

LLVM-based

- Подъём в LLVM IR
- Развитая инфраструктура анализа и инструментации

Трансляторы в LLVM

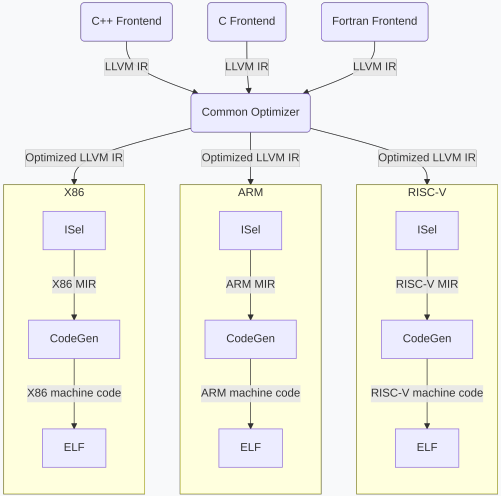
Существующие решение

- mcsema
- mctoll
- instrew

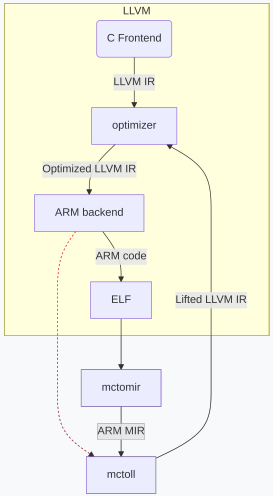
Недостатки

- Алгоритм трансляции специфичен для архитектуры
- Сложная поддержка новых исходных архитектур

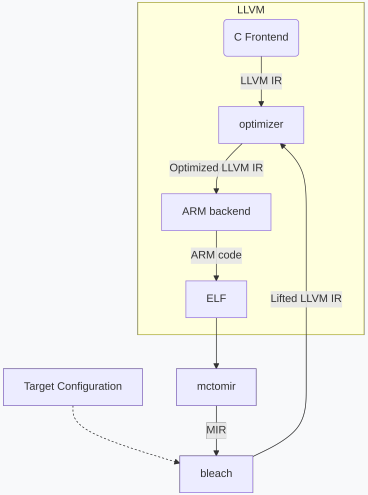
Компилятор LLVM



LLVM-based lifters



llvm-bleach


LLVM
C Frontend
LLVM IR
optimizer
Optimized LLVM IR
ARM backend
ARM code
ELF
Lifted LLVM IR
Target Configuration
mctomir
MIR
bleach

llvm-bleach

Нововведения

- Описание исходной архитектуры - конфигурация
- Не пытаемся восстановить сигнатуру функций
- Обобщённые алгоритмы трансляции
- Не используется LLVM backend

llvm-bleach

Преимущества

- Легко поддержкать новую исходную архитектуру
- Настраиваемый процесс подъёма
- Упрощённая сборка проекта

Общие принципы

Каждой инструкции сопоставляется функция

- Входной операнд -> аргумент
- Результат -> возвращаемое значение

Инструкция

```
$x1 = ADD $x2, $x3
```

Функция

```
define i64 @ADD(i64 %0, i64 %1) {
    %3 = add i64 %1, %0
    ret i64 %3
}
```

Преобразование кода

Блок MIR (RISC-V)

```
bb.1:
  $x19 = MUL $x7, $x2
  $x28 = ADD $x21, $x19
  $x13 = ADD $x13, $x1
  BNE $x13, $x3, %bb.1
```

Получившийся LLVM IR

```
bb1:
%19 = call i64 @MUL(i64 %x7, i64 %x2)
%x28 = call i64 @ADD(i64 %x21, i64 %19)
%13 = call i64 @ADD(i64 %x13, i64 %x1)
%cmp = icmp ne i64 %13, %x3
br i1 %cmp, label %bb1, label %bb2
```

LLVM IR после подстановки

```
bb1:
  %19 = mul i64 %x7, %x2
  %x28 = add i64 %x21, %19
  %13 = add i64 %x13, %x1
  %cmp = icmp ne i64 %13, %x3
  br i1 %cmp, label %bb1, label %bb2
```

Преобразование функций

- Своё соглашение о вызовах

Структура состояния - единственный аргумент

```
define void @foo(ptr %0) {
  %GPR = getelementptr %register_state, ptr %0, i32 0, i32 0
  ... # loading registers from state
  %x2_upd = add i64 %x2, -1
  ... # save updated registers to state
  call void @bar(ptr %0)
  ... # reloading registers from state
  %x23_upd = mul i64 %x16, %x14
  ... # save updated registers to state
  ret void
}
```

Алгоритм

```
def lift(F: function):
  loadRegsFromState(F.StateArgument)
  for MBB in F:
    for I in MBB:
      if I.isCall():
        saveRegsToState(F.StateArgument)
        BB.insertCall(I.Callee, F.StateArgument)
        reloadRegs(F.StateArgument)
      else if I.isCondBranch():
        Cond = BB.insertCall(I.function, I.Src1, I.Src2)
        BB.insertBranch(Cond, I.ifTrue, I.ifFalse);
      else:
        Regs[I.Dst] = BB.insertCall(I.func, I.Src1, I.Src2)
  saveRegsToState(F.StateArgument)
```

Что дальше

- Генерация конфигурации из формальной спецификации
- Трансляция системных вызовов
- Динамическая и статическая бинарная трансляция
- Тестирование компилятора clang
- Использование в бинарной инструментации

Спасибо за внимание!