



217 lines (173 loc) · 8.72 KB

Alias Analysis Basic

Introduction

别名分析（又称指针分析）是程序分析中的一类技术，试图确定两个指针是否指向内存中的同一个对象。别名分析有很多中算法，也有很多种分类的方法：flow-sensitive(流敏感) vs. flow-insensitive(流不敏感), context-sensitive(上下文敏感) vs. context-insensitive(上下文不敏感), field-sensitive (域敏感) vs. field-insensitive(域不敏感), unification-based vs. subset-based, etc.

LLVM 中实现了很多别名分析算法：

- basicaa - Basic Alias Analysis (stateless AA impl)
- cfl-anders-aa - Inclusion-Based CFL Alias Analysis
- cfl-steens-aa - Unification-Based CFL Alias Analysis
- external-aa - External Alias Analysis
- globals-aa - Globals Alias Analysis
- scev-aa - ScalarEvolution-based Alias Analysis
- scoped-noalias - Scoped NoAlias Alias Analysis
- tbaa - Type-Based Alias Analysis
-

可以使用如下命令 `$ opt -cfl-anders-aa -aa-eval foo.bc -disable-output -stats` 来评估 LLVM 中已经实现的别名分析算法的分析效果。命令行参数 `cfl-anders-aa` 表示调用 Inclusion-Based CFL Alias Analysis 算法，命令行参数 `aa-eval` 调用的 `aa-eval` pass 用于输出一些统计信息，该 pass 的实现文件位于 `llvm-5.0.1.src/include/llvm/Analysis/AliasAnalysisEvaluator.h` 和 `llvm-5.0.1.src/lib/Analysis/AliasAnalysisEvaluator.cpp`，这些统计信息反映了别名分析的精确度（别名分析越精确，输出的统计信息中 `may alias` 所占的比例越小）。

上述命令的一个输出示例如下：

```
→ ~ opt -cfl-anders-aa -aa-eval ./test.bc -disable-output -stats
===== Alias Analysis Evaluator Report =====
 210 Total Alias Queries Performed
 161 no alias responses (76.6%)
  31 may alias responses (14.7%)
   0 partial alias responses (0.0%)
  18 must alias responses (8.5%)
Alias Analysis Evaluator Pointer Alias Summary: 76%/14%/0%/8%
  69 Total ModRef Queries Performed
  25 no mod/ref responses (36.2%)
   0 mod responses (0.0%)
   0 ref responses (0.0%)
  44 mod & ref responses (63.7%)
Alias Analysis Evaluator Mod/Ref Summary: 36%/0%/0%/63%
```



Must, May, Partial and No Alias Responses

正如上述命令的输出示例所示，在 LLVM 中别名分析的结果有四种：

- NoAlias，两个指针之间没有直接依赖的关系时就是 NoAlias。比如：两个指针指向非重叠的内存区域；两个指针只被用于读取内存（？）；有一段内存空间，存在一个指针用于访问该段内存，该段内存空间被 free（释放），然后被 realloc（重新分配），另外一个指针用于访问该段内存空间，这两个指针之间为 NoAlias。
- MayAlias，两个指针可能指向同一个对象，MayAlias 是最不精确（保守）的分析结果
- PartialAlias，两个内存对象以某种方式存在重叠的部分，注意：不管两个内存对象起始地址是否相同，只要有重叠的部分，它们之间就是 PartialAlias
- MustAlias，两个内存对象互为别名

AliasAttrs

类 AliasAttrs 用来描述一个指针是否具有某些对别名分析有用的特殊属性，包括：

- AttrNone, represent whether the said pointer comes from an unknown source
- AttrUnknown, represent whether the said pointer comes from a source not known to alias analyses
- AttrCaller, represent whether the said pointer comes from a source not known to the current function but known to the caller. Values pointed to by the arguments of the current function have this attribute set
- AttrEscaped, represent whether the said pointer comes from a known source but escapes to the unknown world (e.g. casted to an integer, or passed as an argument to opaque function). Unlike non-escaped pointers, escaped ones may alias pointers coming from unknown sources
- AttrGlobal, represent whether the said pointer is a global value
- AttrArg, represent whether the said pointer is an argument, and if so, what index the argument has
- ExternallyVisibleAttrs, return a new AliasAttrs that only contains attributes meaningful to the caller. This function is primarily used for interprocedural analysis. Currently, externally visible AliasAttrs include AttrUnknown, AttrGlobal, and AttrEscaped

Alias Analysis Precision Evaluator (aa-eval)

aa-eval 的实现是 AAEvalLegacyPass 类，其实现代码如下：

```
class AAEvalLegacyPass : public FunctionPass {  
    std::unique_ptr<AAEvaluator> P;  
  
public:  
    static char ID; // Pass identification, replacement for typeid  
    AAEvalLegacyPass() : FunctionPass(ID) {  
        initializeAAEvalLegacyPassPass(*PassRegistry::getPassRegistry());  
    }  
  
    void getAnalysisUsage(AnalysisUsage &AU) const override {  
        AU.addRequired<AAResultsWrapperPass>();  
        AU.setPreservesAll();  
    }  
  
    bool doInitialization(Module &M) override {  
        P.reset(new AAEvaluator());  
        return false;  
    }  
  
    bool runOnFunction(Function &F) override {  
        P->runInternal(F, getAnalysis<AAResultsWrapperPass>().getAAResults());  
        return false;  
    }  
};
```



```

    }
    bool doFinalization(Module &M) override {
        P.reset();
        return false;
    }
};

```

由 `getAnalysisUsage` 函数体的内容，可知 `AAEvalLegacyPass` 依赖于 `AAResultsWrapperPass` 的执行。而 `AAEvalLegacyPass` 在 `doInitialization` 中创建了一个 `AAEvaluator` 的对象，然后在 `runOnFunction` 函数中调用了 `AAEvaluator` 的 `runInternal` 函数，最后在 `doFinalization` 将指向之前创建的 `AAEvaluator` 对象的删除，可见 `aa-eval` 功能的实现是通过 `AAEvaluator` 这个类。

根据 `AAResultsWrapperPass` 这个 pass 的命名，猜测该 pass 用于收集 `AliasAnalysis` 的结果。找到 `AAResultsWrapperPass` 的实现：

```

/// A wrapper pass to provide the legacy pass manager access to a suitably
/// prepared AAResults object.

```



[LLVM-Clang-Study-Notes](#) / [source](#) / [analysis](#) / [alias-analysis](#) / [AliasAnalysis-Basic.rst](#)

[↑ Top](#)

Preview

Code

Blame

Raw



```

AAResultsWrapperPass();

AAResults &getAAResults() { return *AAR; }
const AAResults &getAAResults() const { return *AAR; }

bool runOnFunction(Function &F) override;

void getAnalysisUsage(AnalysisUsage &AU) const override;
};

```

注释的内容印证了我们的猜测，`AAResultsWrapperPass` 就是提供一个接口供 pass manager 来访问别名分析的结果。

如果看一下 `AAResultsWrapperPass::runOnFunction` 的实现，会看到很多类似于如下片段的代码。

```

if (auto *WrapperPass = getAnalysisIfAvailable<CFLAndersAAWrapperPass>())
    AAR->addAAResult(WrapperPass->getResult());

```



将各种别名分析算法（如果该别名分析算法 Available，即被指定执行）的结果加入到 `AAResults` 中。

我们回到 `AAEvalLegacyPass::runOnFunction` 的实现：

```
bool runOnFunction(Function &F) override {  
    P->runInternal(F, getAnalysis<AAResultsWrapperPass>().getAAResults());  
    return false;  
}
```



`getAnalysis<AAResultsWrapperPass>().getAAResults()` 的返回结果就是 `AAResultsWrapperPass` 的成员变量 `AAR` 所指向 `AAResults`。接下来，我们步入到 `void AAEvaluator::runInternal(Function &F, AAResults &AA)` 函数会看到，在该函数中是通过调用 `AAResults::alias` 来获取别名信息的。查看 `AAResults::alias` 的实现代码：

```
AliasResult AAResults::alias(const MemoryLocation &LocA,  
                             const MemoryLocation &LocB) {  
    for (const auto &AA : AAs) {  
        auto Result = AA->alias(LocA, LocB);  
        if (Result != MayAlias)  
            return Result;  
    }  
    return MayAlias;  
}
```



前面提到，`AAResults` 中会保存多种别名分析算法（如果该别名分析算法 Available）的结果，对于两个 `MemoryLocation` 来讲其别名关系就是前面提到的四种：`NoAlias`, `MayAlias`, `PartialAlias`, `MustAlias`；其中 `MayAlias` 是最不精确的结果，在 `AAResults::alias` 的实现中，在遍历不同的别名分析算法的结果给出的两个 `MemoryLocation` 之间的别名关系时，如果别名关系不是 `MayAlias`，就返回该别名分析结果。`AAResults::alias` 这样的实现方式，可以在指定别名分析算法时，组合多种别名分析算法，获取更精确的分析结果。

对于不同的别名分析算法在实现时，都要定义一个继承自 `AAResultBase` 的 `Result` 类，并重写 `AliasResult alias(const MemoryLocation &, const MemoryLocation &);` 函数。