编译器优化那些事儿（6）：别名分析概述

简介

别名分析是编译器理论中的一种技术，用于确定存储位置是否可以以多种方式访问。如果两个指针指向相同的位置，则称这两个指针为别名。但是，它不能与指针分析混淆，指针分析解决的问题是一个指针可能指向哪些对象或者指向哪些地址，而别名分析解决的是两个指针指向的是否是同一个对象。指针分析和别名分析通常通过静态代码分析来实现。

别名分析在编译器理论中非常重要，在代码优化和安全方面有着非常广泛且重要的应用。编译器级优化需要指针别名信息来执行死代码消除(删除不影响程序结果的代码)、冗余加载/存储指令消除、指令调度(重排列指令)等。编译器级别的程序安全使用别名分析来检测内存泄漏和内存相关的安全漏洞。

别名分析分类

别名分析种类繁多，通常按如下属性进行分类：域敏感度(field-sensitivity)、过程内分析(Intra-Procedural)v.s.过程间分析(Inter-Procedural)、上下文敏感度(context-sensitivity)和流敏感度(flow-sensitivity)。

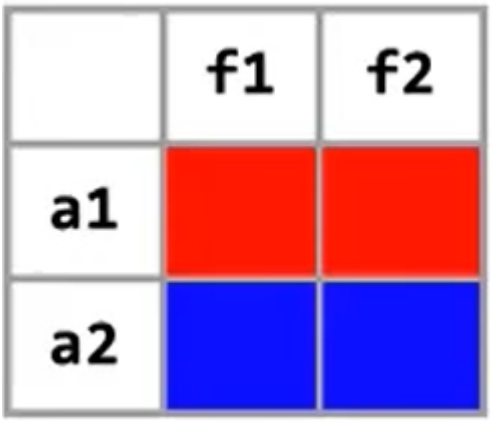
1. 域敏感(Field-Sensitivity)

域敏感度是对用户自定义类型进行分析的一种策略(亦可以处理数组)。在域敏感维度共有三种分析策略：域敏感(field-sensitive)、域非敏感(field-insensitive)、域基础分析(field-based)。以下面代码为例：

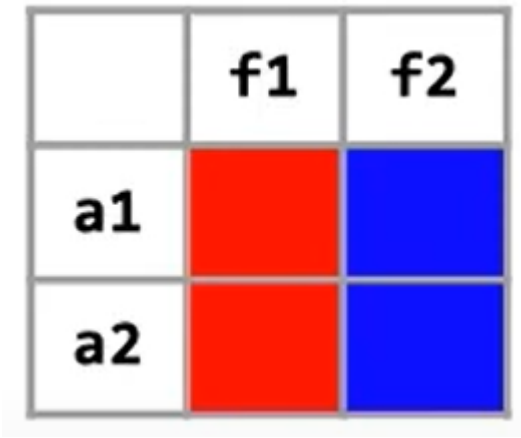
```
1. struct Test {
2. int field1;
3. int field2;
4. }
5.
6. Test a1;
7. Test a2;
```

Note：field这里为结构体或者类的数据成员。

域非敏感：对每个对象建模，而对对象中的成员不进行处理；其建模后的结果如下图，仅有a1.*和a2.*的区别：



域基础分析：仅对结构体中的成员进行建模，而不感知对象。其建模后的结果如下图，仅有*.field1和*.field2：



域敏感：既对对象建模，又对成员变量进行处理。其建模后的结果如下图，有a1.field1、a1.field2、a2.field1、a2.field2：

处理数组时，相同的原则亦适用。以C整数数组为例：int a[10]，域非敏感分析仅使用一个节点建模：a[*]，而域敏感分析创建10个节点：a[0]、a[1]、...、a[9]。

总结：域敏感别名分析准确性高，但是当存在嵌套结构或者大数组时，节点数量会迅速增加，分析成本也会陡然上升。

2. 过程内分析(Intra-Procedural)v.s.过程间分析(Inter-Procedural)

过程内分析仅分析函数体内部的指针，并没有考虑与其他函数之间的相互影响。需要特别指出的是，过程内分析当处理包含指针入参的函数或者返回指针的函数时，其分析可能不够准确。相反，过程间分析会在函数调用过程中处理指针的行为。

过程内分析不易于扩展，精度较低。相比过程间分析，过程内分析更容易实现，且过程内/间分析与上下文敏感度分析高度相关，因为一个上下文敏感分析必定是一个过程间分析。

3. 上下文敏感度(Context-Sensitivity)

上下文敏感度用来控制函数调用该如何分析。有两种分析方法：上下文敏感(context-sensitive)和上下文非敏感(context-insensitive)。上下文敏感在分析函数调用的目标(被调用者)时考虑调用上下文(调用者)。以如下代码为参考[1]:

```
1. 1  public static void main(String[] args) {
2. 2      String name1 = getName(3);  // Tainted
3. 3      String sql1 = "select * from user where name = " + name1;
4. 4      sqlExecute(sql1);  // Taint Sink
5. 5
6. 6      String name2 = getName(-1);  // Not Tainted
7. 7      String sql2 = "select * from user where name = " + name2;
8. 8      sqlExecute(sql2);
9. 9  }
10. 10
11. 11  private static String getName(int x) {
12. 12     if (x > 0) {
13. 13          return System.getProperty("name");
14. 14      } else {
15. 15          return "zhangsan";
16. 16      }
17. 17  }
```

如上所示，getName()方法基于入参的不同，会返回不同的结果，在第2行和第6行，获取到的name1和name2的污点信息不同，当入参为3时，返回的是一个从环境变量中获取的污染的数据，导致sql注入，而当入参为-1时，返回的是一个常量，不是污染数据，不会有问题。在上下文敏感的分析中，在第4行应该报一个sql注入问题，而在第8行则不应该报sql注入问题。而上下文非敏感的分析中，不考虑传入参数的不同，getName()方法则全部返回一个{System.getProperty("name")}∨{zhangsan}，从而导致第4行和第8行都会报一个sql注入的问题。

上下文敏感别名分析需要有一种方法，为函数getName创建抽象描述，以便每次调用它时，分析器都可以将调用上下文应用于抽象描述。

总结：上下文敏感分析比较准确，但是增加了复杂度。

4. 流敏感度(Flow-Sensitivity)

流敏感度是一种是否考虑代码顺序的原则。有两种方法：流敏感(flow-sensitive)和流非敏感(flow-insensitive)。

流非敏感不考虑代码顺序，并为整个程序生成一组别名分析结果，而流敏感考虑代码顺序，计算程序中每个指针出现的位置的别名信息。以如下代码为例：

```
1. 1   int a,b;
2. 2   int *p;
3. 3   p = &a;
4. 4   p = &b;
```

流非敏感的分析结果是针对整个代码块，其结果应该是：指针p可能指向变量a或者变量b。流敏感生成的别名信息是，在第3行，指针p指向变量a，在第4行以后指针p指向变量b。

Note：当程序具有许多条件语句、循环或递归函数时，流敏感分析的复杂性会大大增加。要执行流敏感分析，需要完整的控制流图。因此，流敏感分析非常精确，但对于大多数情况来说，它的分析成本过高，无法在整个程序上执行。

别名分析常见算法介绍

常见的别名算法共有三种：Andersen's指针分析算法、Steensgaard's指针分析算法和数据结构分析算法。

Andersen's指针分析是一种流非敏感和上下文非敏感的分析算法。Andersen's指针分析算法复杂度较高，实践应用性较差，其时间复杂度为$O(n^3)$，其中n为指针节点个数。

Steensgaard's指针分析算法也是一种流非敏感，上下文非敏感且域非敏感的别名分析算法。其时间复杂度较低，实现相对简单，实践应用广，其时间复杂度为$O(n\alpha(n))$，其中$\alpha(n)$无限接近于1，但是其别名分析的准确性较低。

数据结构分析算法是一种流非敏感，上下文敏感和域敏感的算法。其时间复杂度较低，为O(n * log(n))，应用性较好，但是由于不支持MustAlias(参考“AliasAnalysis Class概览”章节)，导致其应用有局限性。

别名分析在LLVM中的应用与实现

1. 应用

别名分析在代码优化和安全方面有着非常重要且广泛的应用，以下面C代码为例，来简单介绍别名分析在代码优化方面的应用[2]。

```
1. int foo (int __attribute__((address_space(0)))* a,
2.          int __attribute__((address_space(1)))* b) {
3.     *a = 42;
4.     *b = 20;
5.     return *a;
6. }
```

__attribute__属性指定了变量a指向地址0，变量b指向地址1。我们知道在ARM架构中，地址0和地址1是完全不同的，修改地址0中的内存永远不会修改地址1中的内存。以下为该函数可能生成的LLVM IR信息：

```
1. define i32 @foo(i32 addrspace(0)* %a, i32 addrspace(1)* %b) #0 {
2. entry:
3.   store i32 42, i32 addrspace(0)* %a, align 4
4.   store i32 20, i32 addrspace(1)* %b, align 4
5.   %0 = load i32, i32* %a, align 4
6.   ret i32 %0
7. }
```

第一个store将42存储到变量a指向的地址，第二个store指令将20存储到变量b指向的地址。%0 = ... 指向的行将变量a中的值加载到一个临时变量0中，并在最后一行返回该临时变量0。

上述代码是未对foo函数进行优化的情况，下面我们考虑对foo函数进行优化。

我们优化后的代码可能如下：删除了load指令对应的行，最后一行直接返回了常量42。

```
1. define i32 @foo(i32 addrspace(0)* %a, i32 addrspace(1)* %b) #0 {
2. entry:
3.   store i32 42, i32 addrspace(0)* %a, align 4
4.   store i32 20, i32 addrspace(1)* %b, align 4
5.   ret i32 42
6. }
```

然而，我们进行优化的时候需要仔细一些，因为上述优化仅在a和b指向的地址不会相互影响时有效。例如：当我们给foo函数传递的指针相互影响时：

```
1. int i = 0;
2. int result = foo(&i, &i);
```

在未开启优化的版本中，变量i将先被设置为42，然后被设置为20，最后返回20。然而，在优化版本中，虽然我们执行了两次store操作依次将42、20赋值给变量i，但是返回值是42，而不是20。因此优化版本破坏了foo函数本身的行为。

如果应用了别名分析，编译器能够合理地执行上述优化。在执行优化前判断入参a和b是否为别名，如果是别名，则不执行删除load指令对应行的操作，否则执行删除操作。

2. 实现

本文以LLVM16.0.0版本为参考，从代码接口入手，带领大家学习别名分析的代码实现。

LLVM AliasAnalysis类是LLVM系统中客户使用和别名分析实现的主要接口，或者说一个“基类”。除了简单的别名分析信息外，这个类还声明了Mod/Ref信息，从而使强大的分析和转换能够很好地协同工作。

源码参考链接：AliasAnalysis.h[3]、AliasAnalysis.cpp[4]。

(1)基础知识

MemoryLocation：LLVM中对内存地址的描述，主要应用在别名分析中，我们需要掌握该类中三个属性：

```
/// The address of the start of the location.
const Value *Ptr;

/// The maximum size of the location, in address-units, or
/// UnknownSize if the size is not known.
///
/// Note that an unknown size does not mean the pointer aliases the entire
/// virtual address space, because there are restrictions on stepping out of
/// one object and into another. See
/// http://llvm.org/docs/LangRef.html#pointeraliasing
LocationSize Size;

/// The metadata nodes which describes the aliasing of the location (each
/// member is null if that kind of information is unavailable).
AAMDNodes AATags;
```

其中，Ptr表示内存开始地址，Size表示内存大小，AATags是描述内存位置别名的metadata节点集合 。

(2)AliasAnalysis Class 概览

AliasAnalysis类定义了各种别名分析实现应该支持的接口。这个类导出两个重要的枚举:AliasResult和ModRefResult，它们分别表示别名查询或mod/ref查询的结果。

a. 关键代码如下，AliasAnalysis为AAResults类别名：

```
954  };
955
956  /// Temporary typedef for legacy code that uses a generic \c AliasAnalysis
957  /// pointer or reference.
958  using AliasAnalysis = AAResults;
```

b. AliasResult关键代码如下：

```
91
92  public:
93    enum Kind : uint8_t {
94      /// The two locations do not alias at all.
95      ///
96      /// This value is arranged to convert to false, while all other values
97      /// convert to true. This allows a boolean context to convert the result to
98      /// a binary flag indicating whether there is the possibility of aliasing.
99      NoAlias = 0,
100     /// The two locations may or may not alias. This is the least precise
101     /// result.
102     MayAlias,
103     /// The two locations alias, but only due to a partial overlap.
104     PartialAlias,
105     /// The two locations precisely alias each other.
106     MustAlias,
107   };
```

其中NoAlias表示两个内存对象没有任何重叠区域；MayAlias表示两个指针可能指向同一对象；PartialAlias表示两个内存对象对应的地址空间有重叠；MustAlias表示两个内存对象总是从同一位置开始。

c. ModRefResult关键代码

```
141
142  /// Flags indicating whether a memory access modifies or references memory.
143  ///
144  /// This is no access at all, a modification, a reference, or both
145  /// a modification and a reference.
146  enum class ModRefInfo : uint8_t {
147    /// The access neither references nor modifies the value stored in memory.
148    NoModRef = 0,
149    /// The access may reference the value stored in memory.
150    Ref = 1,
151    /// The access may modify the value stored in memory.
152    Mod = 2,
153    /// The access may reference and may modify the value stored in memory.
154    ModRef = Ref | Mod,
155    LLVM_MARK_AS_BITMASK_ENUM(ModRef),
156  };
```

其中NoModRef表示访问内存的操作既不会修改该内存也不会引用该内存；Ref表示访问内存的操作会可能引用该内存；Mod表示访问内存的操作可能会修改该内存；ModRef表示访问内存的操作既可能引用该内存也可能修改该内存。

alias接口

其接口定义如下：

```
500    /// \name Alias Queries
501    /// @{
502
503    /// The main low level interface to the alias analysis implementation.
504    /// Returns an AliasResult indicating whether the two pointers are aliased to
505    /// each other. This is the interface that must be implemented by specific
506    /// alias analysis implementations.
507    AliasResult alias(const MemoryLocation &LocA, const MemoryLocation &LocB);
508
509    /// A convenience wrapper around the primary \c alias interface.
510    AliasResult alias(const Value *V1, LocationSize V1Size, const Value *V2,
511                      LocationSize V2Size) {
512      return alias(MemoryLocation(V1, V1Size), MemoryLocation(V2, V2Size));
513    }
514
515    /// A convenience wrapper around the primary \c alias interface.
516    AliasResult alias(const Value *V1, const Value *V2) {
517      return alias(MemoryLocation::getBeforeOrAfter(V1),
518                   MemoryLocation::getBeforeOrAfter(V2));
519    }
```

别名方法是用于确定两个MemoryLocation对象是否相互别名的主要接口。它接受两个MemoryLocation对象作为输入，并根据需要返回MustAlias、PartialAlias、MayAlias或NoAlias。与所有AliasAnalysis接口一样，alias方法要求其入参的两个MemoryLocation对象定义在同一个函数中，或者至少有一个值是常量。

其接口实现如下：

```
119  AliasResult AAResults::alias(const MemoryLocation &LocA,
120                               const MemoryLocation &LocB) {
121    SimpleAAQueryInfo AAQIP;
122    return alias(LocA, LocB, AAQIP);
123  }
124
125  AliasResult AAResults::alias(const MemoryLocation &LocA,
126                               const MemoryLocation &LocB, AAQueryInfo &AAQI) {
127    AliasResult Result = AliasResult::MayAlias;
128
129    if (EnableAATrace) {
130      for (unsigned I = 0; I < AAQI.Depth; ++I)
131        dbgs() << "  ";
132      dbgs() << "Start " << *LocA.Ptr << " @ " << LocA.Size << ", "
133             << *LocB.Ptr << " @ " << LocB.Size << "\n";
134    }
135
136    AAQI.Depth++;
137    for (const auto &AA : AAs) {
138      Result = AA->alias(LocA, LocB, AAQI);
139      if (Result != AliasResult::MayAlias)
140        break;
141    }
142    AAQI.Depth--;
143
144    if (EnableAATrace) {
145      for (unsigned I = 0; I < AAQI.Depth; ++I)
146        dbgs() << "  ";
147      dbgs() << "End " << *LocA.Ptr << " @ " << LocA.Size << ", "
148             << *LocB.Ptr << " @ " << LocB.Size << " = " << Result << "\n";
149    }
150
151    if (AAQI.Depth == 0) {
152      if (Result == AliasResult::NoAlias)
153        ++NumNoAlias;
154      else if (Result == AliasResult::MustAlias)
155        ++NumMustAlias;
156      else
157        ++NumMayAlias;
158    }
159    return Result;
160  }
```

getModRefInfo 接口

getModReInfo方法返回关于给定的指令执行是否可以读取或修改给定内存位置的信息。Mod/Ref信息具有保守性：如果一条指令可能读或写一个位置，则返回ModRef。其接口定义众多，我们以如下接口为例来进行学习。

```
682
683    /// getModRefInfo (for call sites) - Return information about whether
684    /// a particular call site modifies or reads the specified memory location.
685    ModRefInfo getModRefInfo(const CallBase *Call, const MemoryLocation &Loc);
686
687    /// getModRefInfo (for call sites) - A convenience wrapper.
688    ModRefInfo getModRefInfo(const CallBase *Call, const Value *P,
689                             LocationSize Size) {
690      return getModRefInfo(Call, MemoryLocation(P, Size));
691    }
```

其接口实现如下：

```
217  ModRefInfo AAResults::getModRefInfo(const CallBase *Call,
218                                      const MemoryLocation &Loc) {
219    SimpleAAQueryInfo AAQIP;
220    return getModRefInfo(Call, Loc, AAQIP);
221  }
222
223  ModRefInfo AAResults::getModRefInfo(const CallBase *Call,
224                                      const MemoryLocation &Loc,
225                                      AAQueryInfo &AAQI) {
226    ModRefInfo Result = ModRefInfo::ModRef;
227
228    for (const auto &AA : AAs) {
229      Result &= AA->getModRefInfo(Call, Loc, AAQI);
230
231      // Early-exit the moment we reach the bottom of the lattice.
232      if (isNoModRef(Result))
233        return ModRefInfo::NoModRef;
234    }
235
236    // Try to refine the mod-ref info further using other API entry points to the
237    // aggregate set of AA results.
238    auto MRB = getModRefBehavior(Call);
239    if (onlyAccessesInaccessibleMem(MRB))
240      return ModRefInfo::NoModRef;
241
242    if (onlyReadsMemory(MRB))
243      Result &= ModRefInfo::Ref;
244    else if (onlyWritesMemory(MRB))
245      Result &= ModRefInfo::Mod;
246
247    if (onlyAccessesArgPointees(MRB) || onlyAccessesInaccessibleOrArgMem(MRB)) {
248      ModRefInfo AllArgsMask = ModRefInfo::NoModRef;
249      if (doesAccessArgPointees(MRB)) {
250        for (const auto &I : llvm::enumerate(Call->args())) {
251          const Value *Arg = I.value();
252          if (!Arg->getType()->isPointerTy())
253            continue;
254          unsigned ArgIdx = I.index();
255          MemoryLocation ArgLoc =
256              MemoryLocation::getForArgument(Call, ArgIdx, TLI);
257          AliasResult ArgAlias = alias(ArgLoc, Loc, AAQI);
258          if (ArgAlias != AliasResult::NoAlias)
259            AllArgsMask |= getArgModRefInfo(Call, ArgIdx);
260        }
261      }
262      // Return NoModRef if no alias found with any argument.
263      if (isNoModRef(AllArgsMask))
264        return ModRefInfo::NoModRef;
265      // Logical & between other AA analyses and argument analysis.
266      Result &= AllArgsMask;
267    }
268
269    // If Loc is a constant memory location, the call definitely could not
270    // modify the memory location.
271    if (isModSet(Result) && pointsToConstantMemory(Loc, AAQI, /*OrLocal*/ false))
272      Result &= ModRefInfo::Ref;
273
274    return Result;
275  }
```

从上述代码可知，处理共分为四步：

① 遍历AAs，如果发现其任一结果是NoModRef，则直接返回，对应代码行228-234；

② 调用节点(call)操作中是否访问了一个在LLVM IR中无法访问的地址，如果是的话，直接返回NoModRef，否则获取其调用节点的ModRefInfo信息，对应代码行239-240；

③ 处理调用节点中指针入参的ModRefInfo信息，如果发现是NoModRef，则直接返回NoModRef，否则将ModRefInfo信息和之前的结果合并，对应代码行247-266；

④ 如果getModRefInfo函数中的入参Loc指定的内存地址具有常量属性并且ModRefInfo信息包含Mod，则调用节点一定不会修改Loc内存，因此需要将Ref属于与之前的结果做逻辑与操作，对应代码行271-272。

(3)LLVM中已经实现的别名分析

-basic-aa pass

-basic-aa pass是一种激进的本地分析，它提供许多重要的事实信息[5]：

a. 不同的全局变量、堆栈分配和堆分配永远不能别名。

b. 全局变量、栈分配的变量和堆分配变量永远不会和空指针别名。

c. 结构体中的不同字段不能别名。

d. 同一数组，索引不同的两个对象不能别名。

e. 许多通用的标准C库函数从不访问内存或只读取内存。

-globals-aa pass

这个pass实现了一个简单的对内部全局变量(该变量的地址没有被获取过)进行上下文敏感的mod/ref分析和别名分析。如果某个全局变量的地址没有被获取，则该pass可以得出如下结论：没有指针作为该全局变量的别名。该pass还会识别从不访问内存或从不读取内存的函数。这允许某些指定的优化(例如GVN)完全消除调用指令。

这个pass的真正威力在于它为调用指令提供了上下文敏感的mod/ref信息。这使优化器清楚的了解到对于某些函数的调用不会破坏或读取全局变量的值，从而允许消除加载和存储指令。

Note：该pass在使用范围上有一定限制，仅支持没有被取过地址的全局变量，但是该pass分析速度非常快。

除了上述pass外，LLVM中还实现了cfl-steens-aa、cfl-anders-aa、tbaa、scev-aa。目前LLVM中O1，O2，O3优化默认开启的别名分析是basic-aa，globals-aa和tb-aa。

写在最后

编译器技术从20世纪50年代起，已经发展了近70年的历史，但是编译器技术发展到今天，依然是一个非常热门的技术，各大硬件厂商都在开发自己的编译器，包括因特尔推出的Inter C++、ARM公司推出的armclang以及华为推出的毕昇编译器等，且上述三款编译器都是基于LLVM开发。

编译器技术是一门庞大且繁杂的技术，对于初学者来说，这条学习之路道阻且长，盼那些热爱这门技术的赶路人能够行而不辍，未来可期。

参考

[1]https://bbs.huaweicloud.com/blogs/234041

[2]https://blog.tartanllama.xyz/llvm-alias-analysis/

[3]https://llvm.org/doxygen/AliasAnalysis_8h_source.html

[4]https://llvm.org/doxygen/AliasAnalysis_8cpp_source.html

[5]https://llvm.org/docs/AliasAnalysis.html