Alias Analysis

Simone Campanoni
simonec@eeecs.northwestern.edu
Memory alias analysis: the problem

• Does $j$ depend on $i$?

• Do $p$ and $q$ point to the same memory location?
  • Does $q$ alias $p$?

\[
\begin{align*}
i: \text{(*p)} &= \text{varA} + 1 \\
j: \text{varB} &= \text{(*q)} \times 2
\end{align*}
\quad \begin{align*}
i: \text{obj1.f} &= \text{varA} + 1 \\
j: \text{varB} &= \text{obj2.f} \times 2
\end{align*}
\]
Memory alias/data dependence analysis

Code → Memory alias analysis

Aliases: \{ (p, q, strength, location) \}

Data dependence analysis

Data dependences: \{ (i1, i2, type, strength) \}
Outline

• Enhance CAT with alias analysis

• Simple alias analysis

• Alias analysis in LLVM
Let’s start looking at the interaction between memory alias analysis and a code transformation you are familiar with: constant propagation
Escape variables

```
int x, y;
int *p;
p = &x;
myF(p);
...
```

```
void myF (int *q){
  ...
}
```
Constant propagation revisited

```c
int x, y;
int *p;
... = &x;
...
x = 5;
*p = 42;
y = x + 1;
```

We need to know which variables escape. (think about how to do it in LLVM)

Is x constant here?
- Yes, does not escape, reaches this last statement
- Definitely points to x, the statement
- If p might point to x, then we have two reaching definitions that reach this last statement, so x is not constant

Goal of memory alias analysis: understanding
To exploit **memory alias analysis** in a code transformation typically you extend the related code analyses to use the information about pointer aliases
Do you remember liveness analysis?

• A variable \( v \) is live at a given point of a program \( p \) if
  • Exist a directed path from \( p \) to an use of \( v \) and
  • that path does not contain any definition of \( v \)
• Liveness analysis is backwards
• What is the most conservative output of the analysis?

\[
\begin{align*}
\text{GEN}[i] &= \ ? \\
\text{KILL}[i] &= \ ? \\
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
Liveness analysis revisited

int x, y;
int *p;
... = &x;
x = 5;
... (no uses/definitions of x)
*p = 42;
y = x + 1;

How can we modify liveness analysis?

Is x alive here?

• Yes, does not point to x, so the value of x stored there will be used later; yes
• If p definitely points to x, then no
• If p might point to x, then yes
Liveness analysis revisited

mayAliasVar: variable -> set<variable>
mustAliasVar: variable -> set<variable>

How can we modify conventional liveness analysis?

\[
\text{GEN}[i] = \{v \mid \text{variable } v \text{ is used by } i\} \\
\text{KILL}[i] = \{v' \mid \text{variable } v' \text{ is defined by } i\} \\
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Liveness analysis revisited

mayAliasVar : variable -> set<variable>
mustAliasVar: variable -> set<variable>

\[ \text{GEN}[i] = \{ \text{mayAliasVar}(v) \cup \text{mustAliasVar}(v) | \text{variable } v \text{ is used by } i \} \]
\[ \text{KILL}[i] = \{ \text{mustAliasVar}(v) | \text{variable } v \text{ is defined by } i \} \]

\[ \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \]
\[ \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s] \]
Trivial analysis: no code analysis

```c
int x, y;
int *p;
... = &x;
x = 5;
...(no uses/definitions of x)
*p = 42;
y = x + 1;
```

Trivial memory alias analysis

Nothing must alias
Anything may alias everything else

$$\begin{align*}
\text{GEN}[i] &= \{ \text{mayAliasVar}(v) \cup \text{mustAliasVar}(v) \mid v \text{ is used by } i \} \\
\text{KILL}[i] &= \{ \text{mustAliasVar}(v) \mid v \text{ is defined by } i \} \\
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}$$
Great alias analysis impact

int x, y;
int *p;
... = &x;
x = 5;
...(no uses/definitions of x)
*p = 42;
y = x + 1;

Some compilers expose only data dependences. How can we compute aliases for them?

```
GEN[i] = {mayAliasVar(v) U mustAliasVar(v) | v is used by i}
KILL[i] = {mustAliasVar(v) | v is defined by i}
IN[i] = GEN[i] U (OUT[i] – KILL[i])
OUT[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
```
Data dependences and pointer aliases

int x, y;
int *p;
... = &x;
...
x = 5;
*p = 42;
y = x + 1;
Outline

• Enhance CAT with alias analysis

• Simple alias analysis

• Alias analysis in LLVM
Memory alias analysis

• **Assumption:**
  no dynamic memory, pointers can point only to variables

• **Goal:**
  at each program point, compute set of (p->x) pairs
  if p points to variable x

• **Approach:**
  • Based on data-flow analysis
  • May information

```
1: p = &x ;
2: q = &y;
3: if (...){
  4:   z = &v;
  }
5: x++;  
6: p = q;
7: print *p
```
May points-to analysis

• Data flow values:
  \{ (v, x) | v is a pointer variable and x is a variable \}

• Direction: forward

• i: p = &x
  - GEN[i] = \{ (p, x) \}
  - KILL[i] = \{ (p, v) | v “escapes” \}
  - OUT[i] = GEN[i] U (IN[i] – KILL[i])

• IN[i] = U \_{p is a predecessor of i} OUT[p]

• Different OUT[i] equation for different instructions

• i: p = q
  - GEN[i] = \{ \}
  - KILL[i] = \{ \}
  - OUT[i] = \{ (p, z) | (q, z) \in IN[i] \} U (IN[i] – \{ (p, x) for all x \})

Which variable does p point to? print *p
Code example

1: p = &x ;
2: q = &y;
3: if (...){
4:   z = &v;
   }
5: x++;
6: p = q;

```
GEN[1] = {(p, x)}
GEN[2] = {(q, y)}
GEN[3] = { }
GEN[4] = {(z, v)}
GEN[5] = { }
GEN[6] = { }

KILL[1] = {(p, x), (p, y), (p,v)}
KILL[2] = {(q, x), (q, y), (q,v)}
KILL[3] = { }
KILL[4] = {(z, x), (z, y), (z, v)}
KILL[5] = { }
KILL[6] = { }
```

```
IN[1] = { }
IN[2] = {(p,x)}
IN[3] = {(q,y),(p,x)}
IN[4] = {(q,y),(p,x)}
IN[5] = {(z,v),(q,y),(p,x)}
IN[6] = {(z,v),(q,y),(p,x)}

OUT[1] = {(p,x)}
OUT[2] = {(q,y),(p,x)}
OUT[3] = {(q,y),(p,x)}
OUT[4] = {(z,v),(q,y),(p,x)}
OUT[5] = {(z,v),(q,y),(p,x)}
OUT[6] = {(p,y),(z,v),(q,y)}
```
May points-to analysis

• $\text{IN}[i] = U_p \text{ is a predecessor of } i \quad \text{OUT}[p]$ 
• $i: p = &x$
  • $\text{GEN}[i] = \{(p,x)\}$
  • $\text{KILL}[i] = \{(p,v) \mid v \text{ “escapes”}\}$
  • $\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])$
• $i: p = q$
  • $\text{GEN}[i] = \{\}$
  • $\text{KILL}[i] = \{\}$
  • $\text{OUT}[i] = \{(p,z) \mid (q,z) \in \text{IN}[i]\} \cup (\text{IN}[i] - \{(p,x) \text{ for all } x\})$
• $i: p = *q$
  • $\text{GEN}[i] = \{\}$
  • $\text{KILL}[i] = \{\}$
  • $\text{OUT}[i] = \{(p,t) \mid (q,r) \in \text{IN}[i] \land (r,t) \in \text{IN}[i]\} \cup (\text{IN}[i] - \{(p,x) \text{ for all } x\})$
• $i: *q = p \quad ?? (1 \text{ point})$
Memory alias analysis: dealing with dynamically allocated memory

• Issue: each allocation creates a new piece of memory
  
  \[ p = \text{new } T(); \quad p = \text{malloc}(10); \]

• Simple solution: generate a new “variable” for every DFA iteration to stand for new memory

• Extending our data-flow analysis
  
  \[ \text{OUT}[i] = \{(p, \text{newVar})\} \cup \text{IN}[i] – \{(p, x) \text{ for all } x\} \]

• Problem:
  • Domain is unbounded (why)?
  • Iterative data-flow analysis may not converge
Memory alias analysis: dealing with dynamically allocated memory

**Simple solution**
- Create a summary “variable” for each allocation statement
  - Domain is now bounded
- Data-flow equation
  
  \[
  i: p = \text{new } T
  \]
  
  \[
  \text{OUT}[i] = \{(p, \text{inst}_i)\} \cup \text{IN}[i] - \{(p, x) \text{ for all } x}\]

**Alternatives**
- Summary variable for entire heap
- Summary node for each type

**Analysis time/precision tradeoff**
Representations of aliasing

**Alias pairs**
- Pairs that refer to the same memory
- High memory requirements

**Equivalence sets**
- All memory references in the same set are aliases

**Points-to pairs**
- Pairs where the first member points to the second
- Specialized solution
How hard is the memory alias analysis problem?

• Undecidable
  • Landi 1992
  • Ramalingan 1994

• All solutions are conservative approximations

• Is this problem solved?
  • Numerous papers in this area
  • Haven’t we solved this problem yet? [Hind 2001]
Limits of intra-procedural analysis

```c
foo() {
    int x, y, a;
    int *p;
    x = 5;
    p = foo(&x);
    ...
}
```

```c
foo(int *p){
    return p;
}
```

**Does the function call modify x? Where does p point to?**

- With our intra-procedural analysis, we don’t know
- Make worst case assumptions
  - Assume that any reachable pointer may be changed
  - Pointers can be “reached” via globals and parameters
  - Pointers can be passed through objects in the heap
  - p may point to anything that might escape foo
Quality of memory alias analysis

• Quality decreases
  • Across functions
  • When indirect access pointers are used
  • When dynamically allocated memory is used

• Partial solutions to mitigate them
  • Inter-procedural analysis
  • Shape analysis
Outline

• Enhance CAT with alias analysis

• Simple alias analysis

• Alias analysis in LLVM
Using dependence analysis in LLVM

int x, y;
int *p;
... = &x;
x = 5;
...(no uses/definitions of x)
*p = 42;
y = x + 1;

opt -no-aa -CAT bitcode.bc -o optimized_bitcode.bc
LLVM alias analysis: basicaa

• Distinct globals, stack allocations, and heap allocations can never alias
  • p = &g1; q = &g2;
  • p = alloca(...); q = alloca(...);
  • p = malloc(...); q = malloc(...);

• They also never alias nullptr

• Different fields of a structure do not alias

• Baked in information about common standard C library functions

• ... a few more ...
Using basicaa

```c
int x, y;
int *p;
... = &x;
x = 5;
...(no uses/definitions of x)
*p = 42;
y = x + 1;
```

```
- opt -no-aa -CAT bitcode.bc -o optimized_bitcode.bc
- opt -basicaa -CAT bitcode.bc -o optimized_bitcode.bc
```
LLVM alias analysis: globals-aa

• Specialized for understanding reads/writes of globals
  • Analyze only globals that don’t have their address taken

• Context-sensitive
• Mod/ref
• Provide information for call instructions
  • e.g., does call i read/write global g1?

```c
int g1;
int g2;
void f (void *p1){
  ...
  = &g2;
  ...
  g(p1);
  ...
}
```
Using globals-aa

```c
int x, y;
int *p;
... = &x;
x = 5;
...(no uses/definitions of x)
*p = 42;
y = x + 1;
```

```
opt -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```
• basicaa, globals-aa have their strengths and weaknesses

• We would like to use both of them!

• LLVM can chain alias analyses 😊
Using basicaa and globals-aa

int x, y;
int *p;
... = &x;
x = 5;
...(no uses/definitions of x)
*p = 42;
y = x + 1;

```
opt -basicaa -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```
Other LLVM alias analyses

• tbaa
• cfl-steens-aa
• scev-aa
• cfl-anders-aa

• + others not included in the official LLVM codebase
Alias analyses used

• How can we find out what AA is used in O0/O1/O2/O3?
  • opt -O3 -disable-output -debug-pass=Arguments bitcode.bc

• -O0:
• -O1: -basicaa -globals-aa -tbaa
• -O2: -basicaa -globals-aa -tbaa
• -O3: -basicaa -globals-aa -tbaa

• You can always extend O3 adding other AA
• We have seen how to invoke alias analyses

• How can we access alias information and/or dependences in a pass?

• How can we identify which variables might escape?
Identify escaped variables in LLVM

```c
int main (int argc, char *argv[])
{
    CATData d1 = CAT_create_signed_value(5);

    function_that_complicates_everything(&d1);

    int64_t value = CAT_get_signed_value(d1);

    printf("Values: %lld\n", value);

    return 0;
}
```
Identify escaped variables in LLVM

```cpp
; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
  %d1 = alloca i8*, align 8
  %1 = call i8* @__CT_create_signed_value(i64 5)
  store i8* %1, i8** %d1, align 8
  call void @function_that.complicates.everthing(i8** %d1)
  %2 = load i8*, i8** %d1, align 8
  %3 = call i64 @__CT_get_signed_value(i8* %2)
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
  ret i32 0
```
Identify escaped variables in LLVM

... and if variable references are passed to other functions ...
Asking LLVM to run an AA before our pass

Which AA will run?

```c
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired< AAResultsWrapperPass>();
    return;
}
```

```
opt -basicaa -CAT bitcode.bc -o optimized_bitcode.bc

opt -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc

opt -basicaa -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```
AliasAnalysis LLVM class

• Interface between passes that use the information about pointer aliases and passes that compute them (i.e., alias analyses)

• To access the result of alias analyses:

```cpp
bool runOnFunction (Function &F) override {
    AliasAnalysis &aliasAnalysis =
        getAnalysis<AAResultsWrapperPass>() . getAAResults();
}
```

• AliasAnalysis provides information about pointers used by F

```cpp
bool runOnModule (Module &M) override {
    for (auto &F : M) {
        if (F . empty()) {
            continue;
        }
        errs() << "Function " << F . getName() << "\n";
        AliasAnalysis &aliasAnalysis =
            getAnalysis<AAResultsWrapperPass>(F) . getAAResults();
        checkFunction(M, F, aliasAnalysis);
    }
}
```
AliasAnalysis LLVM class: queries

• You can ask to AliasAnalysis the following common queries:
  • *Do these two memory objects alias?*
    
    \[
    \begin{align*}
    (*p1) &= \ldots \\
    \ldots &= \ast p2
    \end{align*}
    \]
  
  • Can this function call read/write a given memory object?

• Memory object representation:
  • Starting address (Value *)
  • Static size (e.g., 10 bytes)

\[
p1 = \text{malloc}(\text{sizeof(T1)});
\]
Why size is used to represent memory objects?

```cpp
int i;
char C[2];
char A[10];
/* ... */
for (i = 0; i != 10; ++i) {
    ((short*)C)[0] = A[i];  /* Two byte store! */
}
```
AliasAnalysis LLVM class: the alias method

- Query: the alias method
  
  ```
  aliasAnalysis.alias(...)
  ```
  
- Input: 2 memory objects

- The size can be platform dependent: ...
  ```
  = malloc(sizeof(long int))
  ```
AliasAnalysis LLVM class: query results

• Constrain to use AliasAnalysis:
  • Value(s) used in the APIs that are not constant must have been defined in the same function
  • Make sure you are asking a valid question

• AliasAnalysis exports two enums used to answer alias queries:
  • AliasResult : NoAlias, MayAlias, PartialAlias, MustAlias
  • ModRefResult: MRI_NoModRef, MRI_Mod, MRI_Ref, MRI_ModRef
AliasResult

- MayAlias
  - Two pointers might refer to the same object
- NoAlias
  - Two pointers cannot refer to the same object
- MustAlias
  - Two pointers always refer to the same object
- PartialAlias
  - Two pointers always point to two objects that partially overlap
Alias query example

```c
switch (aliasAnalysis.alias(pointer, sizePointer, pointer2, sizePointer2)){
    case NoAlias:
        errs() << "No alias\n" ;
        break ;
    case MayAlias:
        errs() << "May alias\n" ;
        break ;
    case PartialAlias:
        errs() << "Partial alias\n" ;
        break ;
    case MustAlias:
        errs() << "Must alias\n" ;
        break ;
default:
    abort();
}
```
Memory instructions

• What if we want to use memory instructions directly?
  • e.g., can this load access the same memory object of this store?

```cpp
switch (aliasAnalysis.alias(MemoryLocation::get(memInst), MemoryLocation::get(memInst2))) {
  case NoAlias:
    errs() << " No alias\n";
    break;
  case MayAlias:
    errs() << " May alias\n";
    break;
  case PartialAlias:
    errs() << " Partial alias\n";
    break;
  case MustAlias:
    errs() << " Must alias\n";
    break;
  default:
    abort();
}
```
Mod/ref queries

• Information about whether the execution of an instruction can modify (mod) or read (ref) a memory location
• It is always conservative (like alias queries)
• API: getModRefInfo
• This API is often used to understand dependences between function calls
Mod/ref query example

```c
switch (aliasAnalysis.getModRefInfo(memInst, pointer, sizePointer)) {
    case MRI_NoModRef:
        break;
    case MRI_Mod:
        break;
    case MRI_Ref:
        break;
    case MRI_ModRef:
        break;
    default:
        abort();
}
```

... call inst, fence inst, ...

MemoryLocation
Other alias queries

The AliasAnalysis and ModRef API includes other functions
- pointsToConstantMemory
- doesNotAccessMemory
- onlyReadsMemory
- onlyAccessesArgPointees
- ...