DEALING WITH ALIASING USING CONTRACTS

BEATING FORTRAN'S PERFORMANCE

Gábor Horváth, PhD Student, Eötvös Loránd University
xazax.hun@gmail.com

CAVEAT

Everything I tell you, is a ~~lie~~ simplification!

ALIASING

```
int f(int &a, float &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, float*) {
    store i32 2, i32* %a
    store float 3, float* %b
    ret i32 2
}
```

ALIASING

```
int f(int &a, float &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, float*) {
    store i32 2, i32* %a
    store float 3, float* %b
    ret i32 2
}
```

```
int f(int &a, int &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, i32*) {
    store i32 2, i32* %a
    store i32 3, i32* %b
    %tmp = load i32, i32* %a
    ret i32 %tmp
}
```

ALIASING

```
int f(int &a, float &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, float*) {
    store i32 2, i32* %a
    store float 3, float* %b
    ret i32 2
}
```

```
int f(int &a, int &b) {
    a = 2;
    b = 3;
    return a;
}
```

```
define i32 f(i32*, i32*) {
    store i32 2, i32* %a
    store i32 3, i32* %b
    %tmp = load i32, i32* %a
    ret i32 %tmp
}
```

Some parameters might alias!

Type based alias analysis

WHY ALIASING MATTERS?

LATENCY NUMBERS

```
L1 cache reference      0.5 ns
Branch mispredict       5   ns
L2 cache reference      7   ns   14x L1 cache
Mutex lock/unlock       25  ns
Main memory reference   100 ns   20x L2 cache, 200x L1 cache
```

WHY ALIASING MATTERS?

LATENCY NUMBERS

```
L1 cache reference      0.5 ns
Branch mispredict       5   ns
L2 cache reference      7   ns   14x L1 cache
Mutex lock/unlock       25  ns
Main memory reference   100 ns   20x L2 cache, 200x L1 cache
```

OPTIMIZATIONS

FORTRAN

- Procedure arguments and variables may not alias
- Inception when CPU time was expensive
- To convince people not to write in assembly...
- ...you need to generate blazing fast code

FORTRAN

- Procedure arguments and variables may not alias
- Inception when CPU time was expensive
- To convince people not to write in assembly...
- ...you need to generate blazing fast code

C++

No standard way (other than types) to give aliasing related hints.

UNVECTORIZED (AS OF CLANG 6)

```
void g(int *result, int **matrix, int height, int width) {
    for(int i = 0; i < height; ++i)
        for(int j = 0; j < width; ++j)
            result[i] += matrix[i][j];
}
```

UNVECTORIZED (AS OF CLANG 6)

```
void g(int *result, int **matrix, int height, int width) {
    for(int i = 0; i < height; ++i)
        for(int j = 0; j < width; ++j)
            result[i] += matrix[i][j];
}
```

VECTORIZED

```
void g(int * restrict result,
       int * restrict * matrix,
       int height, int width) {
    for(int i = 0; i < height; ++i)
        for(int j = 0; j < width; ++j)
            result[i] += matrix[i][j];
}
```

LET'S JUST ADD RESTRICT TO C++?

How to annotate the code below?

```
void g(vector<int> &result, vector<vector<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

LET'S JUST ADD RESTRICT TO C++?

How to annotate the code below?

```
void g(vector<int> &result, vector<vector<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

What would

```
vector<int restrict>
```

or

```
vector<int> restrict
```

mean?

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

Restrict is a precondition!

WHAT DO YOU THINK ABOUT THIS CODE?

```
void f(int * restrict x, int * restrict y);
void g() {
    int x;
    f(&x, &x);
}
```

Adding `restrict` to `f` makes it harder to use. It is now the caller's responsibility to ensure no aliasing is happening.

Restrict is a precondition!

Only if we had a way to describe preconditions in C++...

CONTRACTS TO THE RESCUE?

EXPLORING THE DESIGN SPACE

SIMPLE PRECONDITIONS

```
int f(int &a, int &b) [[expects axiom: &a != &b]] {
    a = 2;
    b = 3;
    return a;
}
```

- `f(x, x);` is undefined
- The precondition is documented
- We have two mitigations:
  - Runtime checks (with axiom removed)
  - Static analysis

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

```
int *merge(int *a, int *b, int num)
  [[expects: __disjoint(a, b, num)]];
```

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

```
int *merge(int *a, int *b, int num)
  [[expects: __disjoint(a, b, num)]];
```

`__disjoint(a, b, c, ..., num)`?

ARRAYS

```
int *merge(int *a, int *b, int num) [[expects: ???]];
```

Extend the language?

```
int *merge(int *a, int *b, int num)
  [[expects: __disjoint(a, b, num)]];
```

__disjoint(a, b, c, ..., num)?

```
int *merge(int *a, int *b, int num)
  [[expects: __unique(a) && __unique(b)]];
```

Unique object vs disjoint memory region.

USER DEFINED TYPES

```
int f(S a, S b)
    [[expects: __disjoint(a.member, b.member)]];
```

USER DEFINED TYPES

```
int f(S a, S b)
    [[expects: __disjoint(a.member, b.member)]];
```

```
int f(S a, S b)
    [[expects: __disjoint(a.method(), b.method())]];
```

USER DEFINED TYPES

```
int f(S a, S b)
    [[expects: __disjoint(a.member, b.member)]];
```

```
int f(S a, S b)
    [[expects: __disjoint(a.method(), b.method())]];
```

What if we need arguments? Use dummy symbols? Existentially or universally quantified?

```
int f(S a, S b)
    [[expects: __disjoint(a.method(???), b.method(???))]];
```

VIEWS TO THE RESCUE?

NON-ALIASING VIEW EXAMPLE

```
tmeplate <typename ... >
class unique_span {
    reference operator[](index_type idx) const
      [[ensures x: __unique(x, this, idx)]];
};
```

BACK TO THE MATRIX EXAMPLE

```
void g(unique_span<int> result,
       vector<unique_span<int>> &matrix) {
    for(int i = 0; i < matrix.size(); ++i)
        for(int j = 0; j < matrix[0].size(); ++j)
            result[i] += matrix[i][j];
}
```

A NEW TYPE? ISN'T THAT HEAVY WEIGHT?

ARE THESE FUNCTIONS THE SAME?

```
double my_sqrt(double x) {
    return sqrt(x);
}
```

```
double my_sqrt(double x) {
    if (x < 0) return 0;
    return sqrt(x);
}
```

```
double my_sqrt(double x) {
    if (x < 0) throw ...;
    return sqrt(x);
}
```

ARE THESE FUNCTIONS THE SAME?

```
double my_sqrt(double x);
```

```
double my_sqrt(double x) [[expects: x >= 0]];
```

```
double my_sqrt(double x) [[expects: x >= 0]]
                         [[ensures ret: ret >= 0]];
```

ARE THESE TYPES THE SAME?

```
unique_span<int>
```

```
span<int>
```

Exercise: how different are these types?

Exercise: how different are these types?

Hint: How many methods need to be annotated?

Exercise: how different are these types?

Hint: How many methods need to be annotated?

Hint2: How many other things need to be annotated? Iterators?

Exercise: how different are these types?

Hint: How many methods need to be annotated?

Hint2: How many other things need to be annotated?

Iterators?

Is it feasible to do all that inline?

It might be a lot of work to create such types, but..

- These can be vocabulary types
- We should use such classes sparingly, as they impose burden on the caller
- Those methods/functions are now screaming that they are special and error prone
- We can do overloads!

WE ALREADY HAVE TO REASON ABOUT ALIASING

- `std::copy*`
- `memcpy` vs `memmove`
- We would get mitigations for existing UB!

THANKS FOR YOUR ATTENTION!

RELATED WORK

- **p0856r0**: Restrict as a library feature
- **n3635**: Annotating alias sets
- The `malloc` attribute of GCC, `noalias` attribute of Clang
- All major compilers has `restrict` like features as extensions

BONUS

NOT VECTORIZED

```
void f(int *a, int *b, const int& num) {
    for(int i = 0; i < num; ++i) {
        a[i] = b[i] * b[i] + 1;
    }
}
```

NOT VECTORIZED

```
void f(int *a, int *b, const int& num) {
    for(int i = 0; i < num; ++i) {
        a[i] = b[i] * b[i] + 1;
    }
}
```

VECTORIZED

```
void f(int *a, int *b, int num) {
    for(int i = 0; i < num; ++i) {
        a[i] = b[i] * b[i] + 1;
    }
}
```

WHO WRITES CODE LIKE THAT?

WHO WRITES CODE LIKE THAT?

```
template<typename T, ...>
void foo(..., const T&) { ... }
```

WHO WRITES CODE LIKE THAT?

```
template<typename T, ...>
void foo(..., const T&) { ... }
```

Rings some bells?

METAPROGRAM TO DECIDE BY REF OR VALUE?

- That might find local optimum
- But global optimum depends on many things:
  - Are the paramaters actually used?
  - What are the types of the formals?
  - How many parameters are there?
  - ...
- Why can't we let the compiler to decide?
- But we would need a good ABI