Byte types, or how to get rid of i8 abuse for chars in LLVM IR

George Mitenkov, Juneyoung Lee, Nuno Lopes

LLVM Dev Meeting 2021

Memory, integers and pointers in LLVM IR

LLVM IR does not associate types with memory. The result type of a load merely indicates the size and alignment of the memory from which to load, as well as the interpretation of the value. The first operand type of a store similarly only indicates the size and alignment of the store.”

- LangRef

LLVM IR does not associate types with memory. The result type of a load merely indicates the size and alignment of the memory from which to load, as well as the interpretation of the value. The first operand type of a store similarly only indicates the size and alignment of the store.”

- LangRef

Memory is untyped?

Memory in LLVM

- Let’s suppose memory is untyped

Memory in LLVM

- ➔ Let’s suppose memory is untyped
- ➔ Every load/store of a type is equivalent of an integer load/store of some size

Memory in LLVM

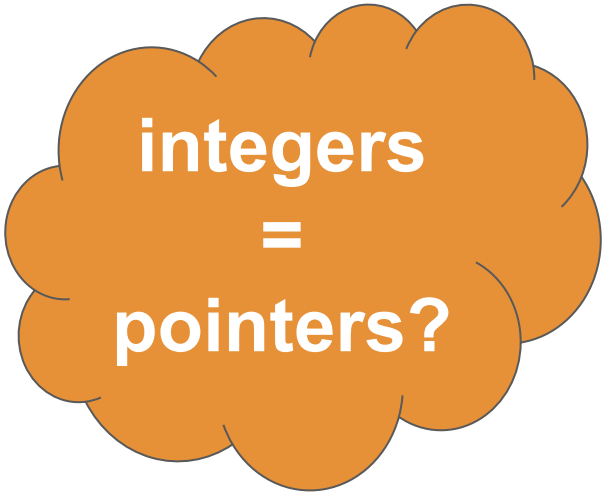
- Let's suppose memory is untyped
- Every load/store of a type is equivalent of an integer load/store of some size

```
define void @foo(i8** %memory, i8* %p) {
  store i8* %p, i8** %memory
  ret void
}

define void @bar(i8** %memory, i8* %p) {
  %m = bitcast i8* %memory to i64*
  %p_int = ptrtoint i8* %p to i64
  store i64 %p_int, i64* %m
  ret void
}
```

Memory in LLVM

- Let's suppose memory is untyped
- Every load/store of a type is equivalent of an integer load/store of some size

```
define void @foo(i8** %memory, i8* %p) {
  store i8* %p, i8** %memory
  ret void
}

define void @bar(i8** %memory, i8* %p) {
  %m = bitcast i8* %memory to i64*
  %p_int = ptrtoint i8* %p to i64
  store i64 %p_int, i64* %m
  ret void
}
```



Integers ≠ Pointers

Integers

- A collection of bits

Pointers

- An integer (address)
- A collection of metadata (e.g. *provenance* = what objects can be accessed)

Integers as pointers

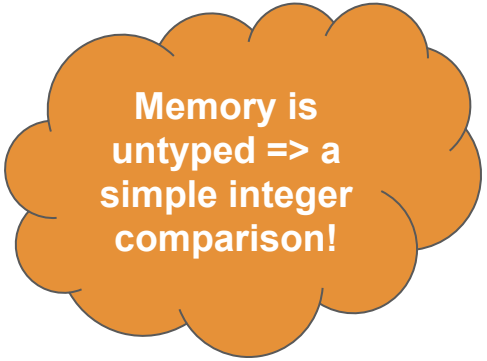
```
define i32 @foo(i32* %p) {
  %q = alloca i32*
  store 0 i32, i32* %p
  store i32* %p, i32** %q
  %q' = bitcast i32** to i64*
  %p_as_int = load i64, i64* %q'
  %cmp = cmp eq i64 %p_as_int, 0x42
  br i1 %cmp, label %true, label %false
%true
  call void @bar(i64 0x42)
  br label %false
%false
  %w = load i32, i32* %p
  ret i32 %w
}
```

Integers as pointers

```
define i32 @foo(i32* %p) {
  %q = alloca i32*
  store 0 i32, i32* %p
  store i32* %p, i32** %q
  %q' = bitcast i32** to i64*
  %p_as_int = load i64, i64* %q'
  %cmp = cmp eq i64 %p_as_int, 0x42
  br i1 %cmp, label %true, label %false
%true
  call void @bar(i64 0x42)
  br label %false
%false
  %w = load i32, i32* %p
  ret i32 %w
}
```

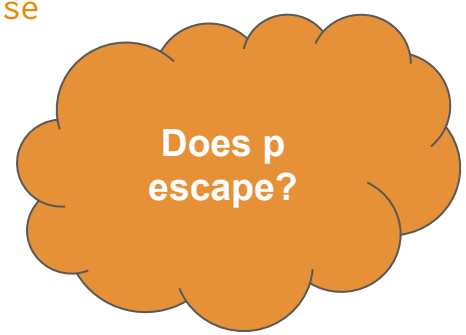
Integers as pointers

```
define i32 @foo(i32* %p) {
  %q = alloca i32*
  store 0 i32, i32* %p
  store i32* %p, i32** %q
  %q' = bitcast i32** to i64*
  %p_as_int = load i64, i64* %q'
  %cmp = cmp eq i64 %p_as_int, 0x42
  br i1 %cmp, label %true, label %false
%true
  call void @bar(i64 0x42)
  br label %false
%false
  %w = load i32, i32* %p
  ret i32 %w
}
```



Integers as pointers

```
define i32 @foo(i32* %p) {
  %q = alloca i32*
  store 0 i32, i32* %p
  store i32* %p, i32** %q
  %q' = bitcast i32** to i64*
  %p_as_int = load i64, i64* %q'
  %cmp = cmp eq i64 %p_as_int, 0x42
  br i1 %cmp, label %true, label %false
%true
  call void @bar(i64 0x42)
  br label %false
%false
  %w = load i32, i32* %p
  ret i32 %w
}
```



Contradiction?

- ➔ If memory is untyped, then loading an integer = loading a pointer
- ➔ But integers ≠ pointers.

Contradiction?

- ➔ If memory is untyped, then loading an integer = loading a pointer
- ➔ But integers ≠ pointers.

- ➔ C uses <u>unsigned char</u> to represent memory and for raw data access
- ➔ But LLVM uses integers and as it seems a typed memory model

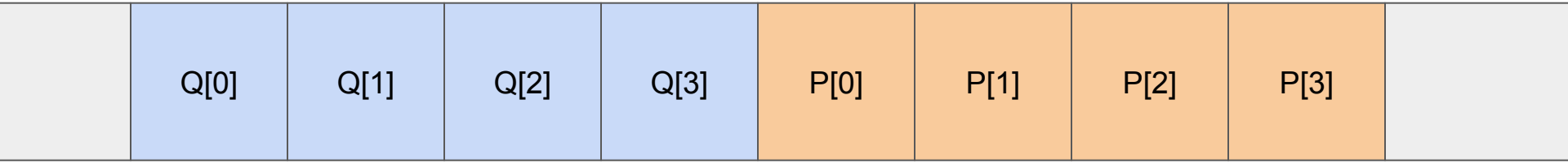
Contradiction?

- If memory is untyped, then loading a[illegible] loading a pointer
- But integers ≠ pointers
- C uses unsigned ch[illegible]d for raw data access
- But LLVM uses integers and as it [illegible]ns a typed memory model

Is this bad?

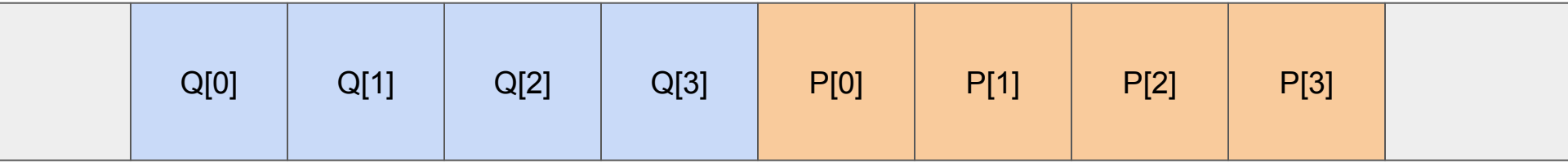
Motivational example

Bug report 37469



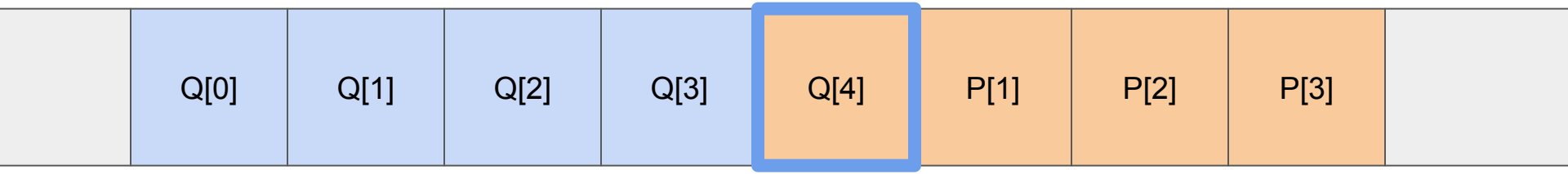

```
int P[4], Q[4];
```

Bug report 37469




```
int P[4], Q[4];

if ((uintptr_t)P == (uintptr_t)&Q[4]) {


}
```

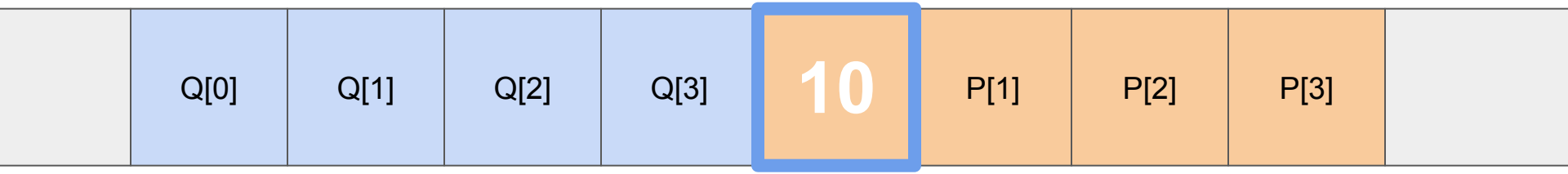
Bug report 37469




```
int P[4], Q[4];

if ((uintptr_t)P == (uintptr_t)&Q[4]) {


}
```

Taking the address of one-past-the-end element of an array is <u>not UB</u>

Bug report 37469



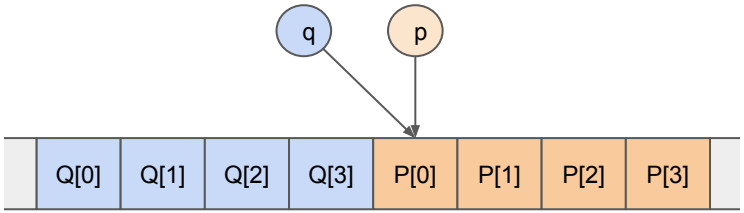

```
int P[4], Q[4];

if ((uintptr_t)P == (uintptr_t)&Q[4]) {
    store_10_to_p(P, &Q[4]);
    printf("%d\n", P[0]);
}
```

Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```

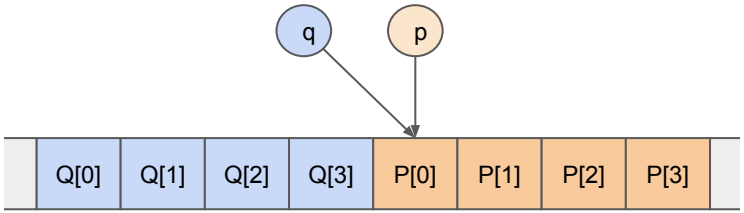
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



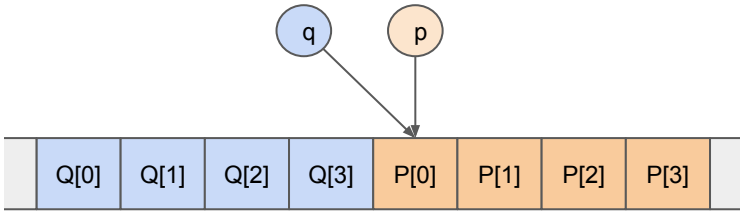
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



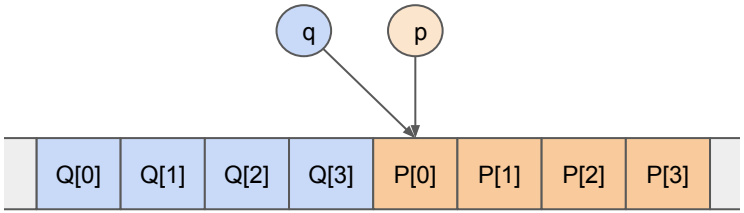
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



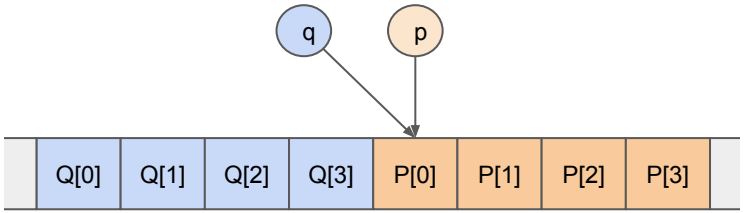
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



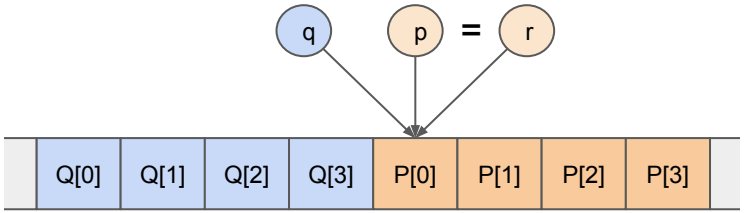
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



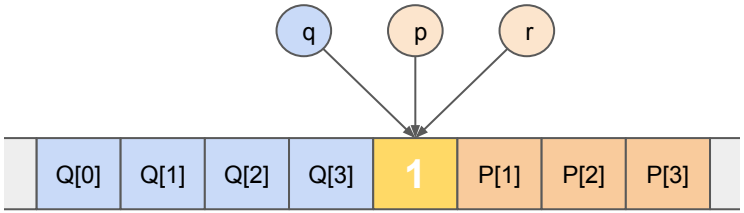
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



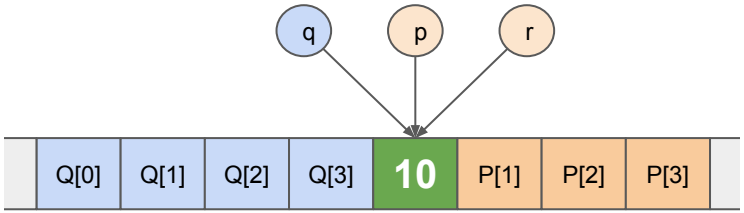
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



Bug report 37469

```
$ clang -O0 -o bug bug.c
$ ./bug

10
```

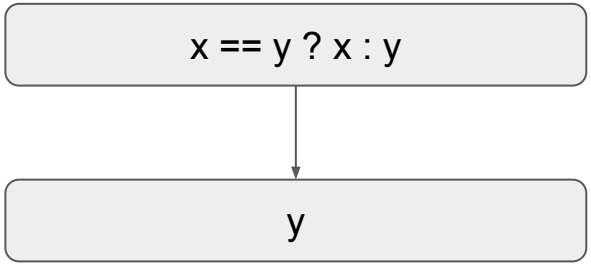
Bug report 37469

```
$ clang -O0 -o bug bug.c
$ ./bug

10
```

```
$ clang -O3 -o bug bug.c
$ ./bug

1
```

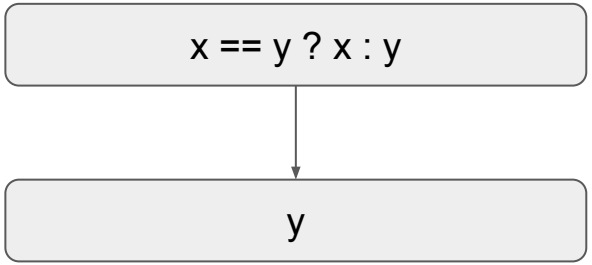
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {
        int c1 = buffer_p[i]; int c2 = buffer_q[i];
        buffer[i] = (c1 == c2) ? buffer_p[i] : buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



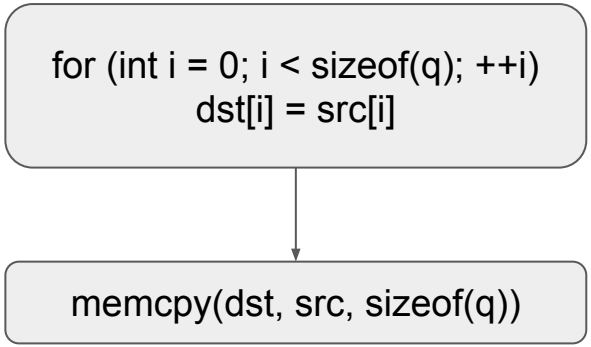
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {

        buffer[i] = buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



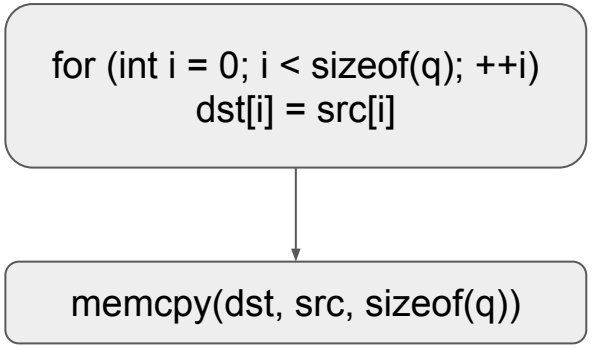
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];
    for (int i = 0; i < sizeof(q); ++i) {

        buffer[i] = buffer_q[i];
    }
    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



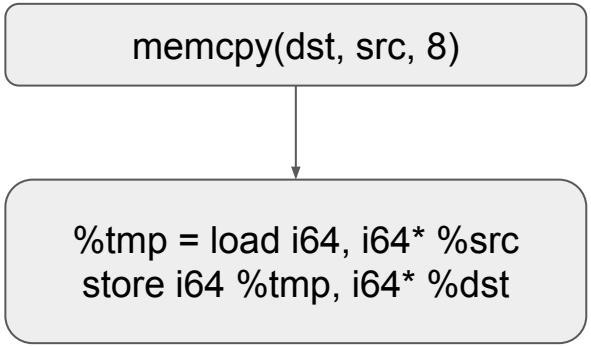
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];

    memcpy(buffer, buffer_q, sizeof(q));



    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



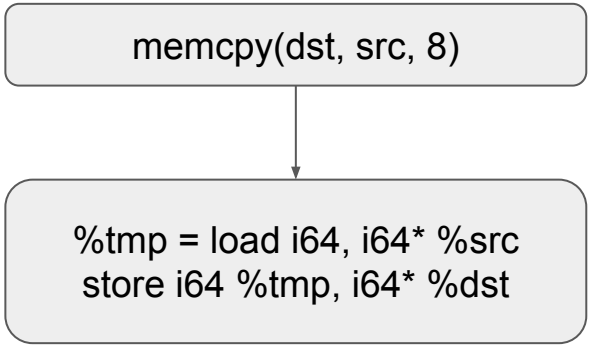
Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];

    memcpy(buffer, buffer_q, sizeof(q));



    int *r;
    memcpy(&r, buffer, sizeof(r));
    *p = 1;
    *r = 10;
}
```



Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];

    %a = load i64, i64* %buffer_q
    store i64 %a, i64* %buffer
    int *r;
    %b = load i64, i64* %buffer
    store i64 %b, i64* %r
    *p = 1;
    *r = 10;
}
```



Bug report 37469

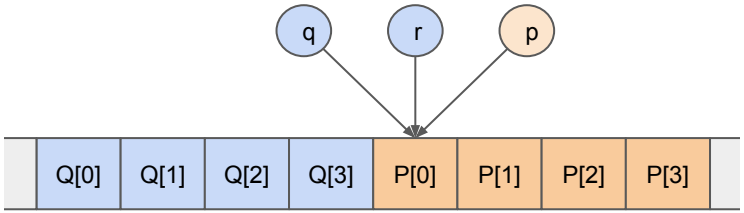
```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];

    %a = load i64, i64* %buffer_q
    store i64 %a, i64* %buffer
    int *r;
    %b = load i64, i64* %buffer
    store i64 %b, i64* %r
    *p = 1;
    *r = 10;
}
```

Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];


    %a = load i64, i64* %buffer_q
    store i64 %a, i64* %r


    *p = 1;
    *r = 10;
}
```

Bug report 37469

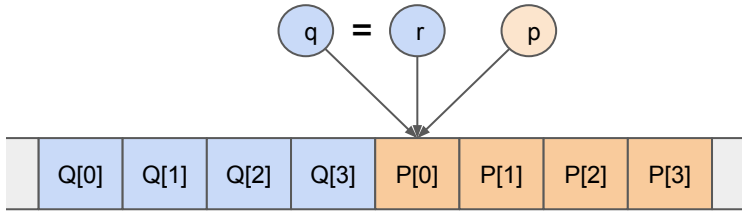
```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];


    %a = load i64, i64* %buffer_q
    store i64 %a, i64* %r


    *p = 1;
    *r = 10;
}
```



Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];


    %a = load i64, i64* %buffer_q
    store i64 %a, i64* %r


    *p = 1;
    *r = 10;
}
```

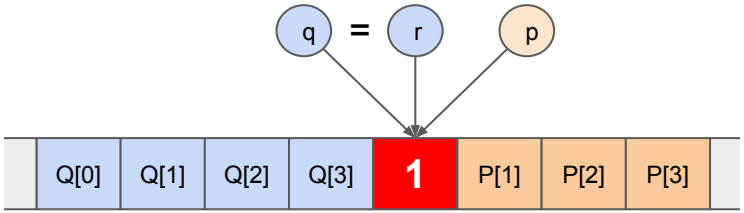
Optimizer thinks r is q!



Bug report 37469

```
void store_10_to_p(int *p, int *q) {
    unsigned char buffer_p[8], buffer_q[8];
    memcpy(buffer_p, &p, sizeof(p));
    memcpy(buffer_q, &q, sizeof(q));
    unsigned char buffer[8];


    %a = load i64, i64* %buffer_q
    store i64 %a, i64* %r


    *p = 1;
    *r = 10;
}
```

Optimizer thinks r is q!



What did go wrong?

Problem

- ➔ Semantics of unsigned char ≠ semantics of i8
- ➔ In LLVM, there is no universal type holder
- ➔ Integers can carry pointers
- ➔ Integers ≠ pointers

Can we solve this without changing LLVM?

Conservative solution

- Keep the memory untyped

Conservative solution

- Keep the memory untyped
- Disable optimizations that are unsound when integers carry pointers

Conservative solution

- Keep the memory untyped
- Disable optimizations that are unsound when integers carry pointers

```
(x == y) ? x : y    =>    y
```

Conservative solution

- Keep the memory untyped
- Disable optimizations that are unsound when integers carry pointers

~~(x == y) ? x : y => y~~

Conservative solution

Low engineering effort

Keep IR the same

- Keep the <u>memory untyped</u>
- <u>Disable optimizations</u> that are unsound when integers carry pointers

~~(x == y) ? x : y => y~~

Conservative solution

Low engineering effort

Keep IR the same

- Keep the memory untyped
- Disable optimizations that are unsound when integers carry pointers

~~(x == y) ? x : y => y~~

Technical debt

Performance regression

So how should we solve it?

Possible solution

- In C/C++ unsigned char and std::byte can access raw data.
- In LLVM IR no type can access raw data, and integers are used instead.
- In LLVM IR, memory is believed to be untyped, but it is not semantically consistent.

Change the lowering of unsigned char and std::byte so that optimizer is aware of “raw data” access properties!

Possible solution

- In C/C++ unsigned char and std::byte can access raw data.
- In LLVM IR no type can access raw data, and integers are used instead.
- In LLVM IR, memory is believed to be untyped, but it is not semantically consistent.

Change the lowering of unsigned char and std::byte so that optimizer is aware of “raw data” access properties!

Let the memory be typed.

Annotating types to keep “raw data” semantics

```
%src’ = bitcast i8** %src to i8*
%dst’ = bitcast i8** %dst to i8*
call void @llvm.memcpy(i8* %dst’, i8* %src’, i32 8, i1 false)
```



```
%src’ = bitcast i8** %src to i64*
%dst’ = bitcast i8** %dst to i64*
%val = load i64, i64* %src’
store i64 %val, i64* %dst’
```

Annotating types to keep “raw data” semantics

```
%src’ = bitcast i8** %src to i8*
%dst’ = bitcast i8** %dst to i8*
call void @llvm.memcpy(i8* %dst’, i8* %src’, i32 8, i1 false)
```



```
%src’ = bitcast i8** %src to i64*
%dst’ = bitcast i8** %dst to i64*
%val = load i64, i64* %src’, !raw_data
store i64 %val, i64* %dst’, !raw_data
```

Annotating types to keep “raw data” semantics

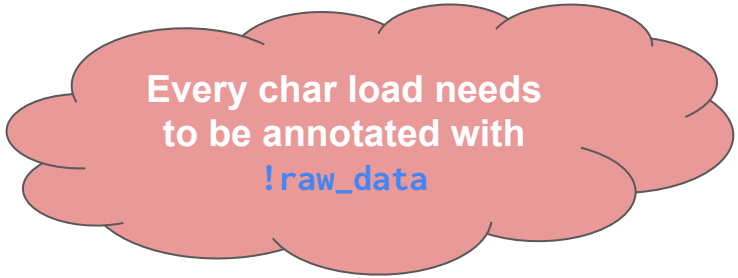
```
%src’ = bitcast i8** %src to i8*
%dst’ = bitcast i8** %dst to i8*
call void @llvm.memcpy(i8* %dst’, i8* %src’, i32 8, i1 false)
```



```
%src’ = bitcast i8** %src to i64*
%dst’ = bitcast i8** %dst to i64*
%val = load i64, i64* %src’, !raw_data
store i64 %val, i64* %dst’, !raw_data
```

Annotating types to keep “raw data” semantics

```
%val = load i8, i8* %p, !raw_data
```




```
store i8 %val, i8* %q, !raw_data
```

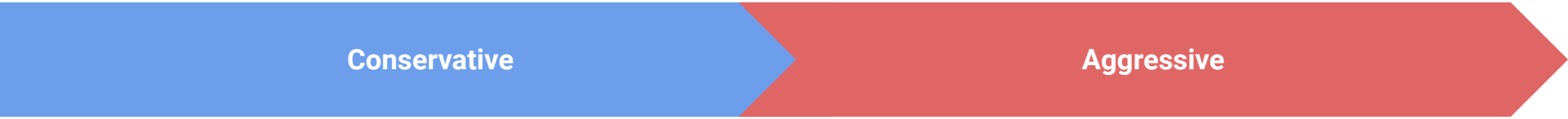
Annotating types to keep “raw data” semantics

```
%val = load i8, i8* %p, !raw_data

...
%a = add i8 %val, i8 %tmp, !raw_data
...

...
%b = select i8 %cnd, i8 %val, i8 %a, !raw_data
...

store i8 %val, i8* %q, !raw_data
```

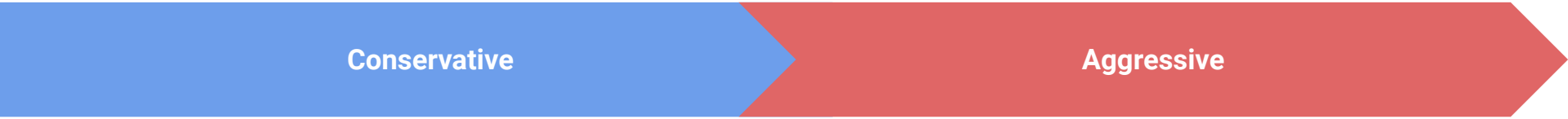
Every char load needs to be annotated with `!raw_data`

Need to annotate all instructions preceding the store

Annotating types to keep “raw data” semantics



Annotating types to keep “raw data” semantics



- **Add** attributes **everywhere**
- **Change optimizations** to take attributes into account
- **Make optimizations** that do not use the attributes **more aggressive**

Annotating types to keep “raw data” semantics

Conservative

- **Add** attributes **everywhere**
- **Change optimizations** to take attributes into account
- **Make optimizations** that do not use the attributes **more aggressive**

Aggressive

- **Add** attributes only **when needed**
- **Identify** which **optimizations** need **to be fixed**
- **Fix** optimizations

Annotating types to keep “raw data” semantics

Conservative

- **Add** attributes **everywhere**
- **Change optimizations** to take attributes into account
- **Make optimizations** that do not use the attributes **more aggressive**

Aggressive

- **Add** attributes only **when needed**
- **Identify** which **optimizations** need **to be fixed**
- **Fix** optimizations

Optimizations ignore attributes by default!
How do we find what we need to fix?

Solution: byte types!

What about a new type instead?

```
%src' = bitcast i8** %src to i8*
%dst' = bitcast i8** %dst to i8*
call void @llvm.memcpy(i8* %dst', i8* %src', i32 8, i1 false)
```



```
%src' = bitcast i8** %src to i64*
%dst' = bitcast i8** %dst to i64*
%val = load i64, i64* %src', !raw_data
store i64 %val, i64* %dst', !raw_data
```

What about a new type instead?

```
%src' = bitcast i8** %src to i8*
%dst' = bitcast i8** %dst to i8*
call void @llvm.memcpy(i8* %dst', i8* %src', i32 8, i1 false)
```



```
%src' = bitcast i8** %src to b64*
%dst' = bitcast i8** %dst to b64*
%val = load b64, b64* %src'
store b64 %val, b64* %dst'
```

Why a new type?

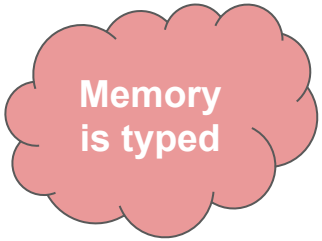
- We can follow an aggressive strategy as with annotations

Why a new type?

- We can follow an aggressive strategy as with annotations
- If we change the lowering of unsigned char to a new type, **optimizations will fail**!

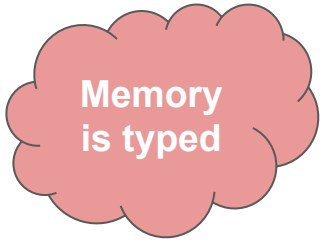
Why a new type?

- We can follow an aggressive strategy as with annotations
- If we change the lowering of unsigned char to a new type, **optimizations will fail**!



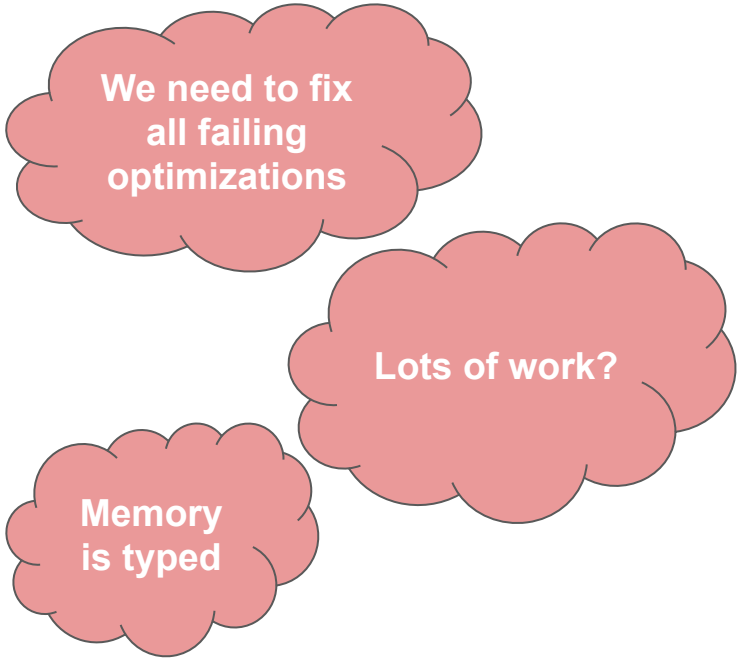
Why a new type?

- We can follow an aggressive strategy as with annotations
- If we change the lowering of unsigned char to a new type, **optimizations will fail**!



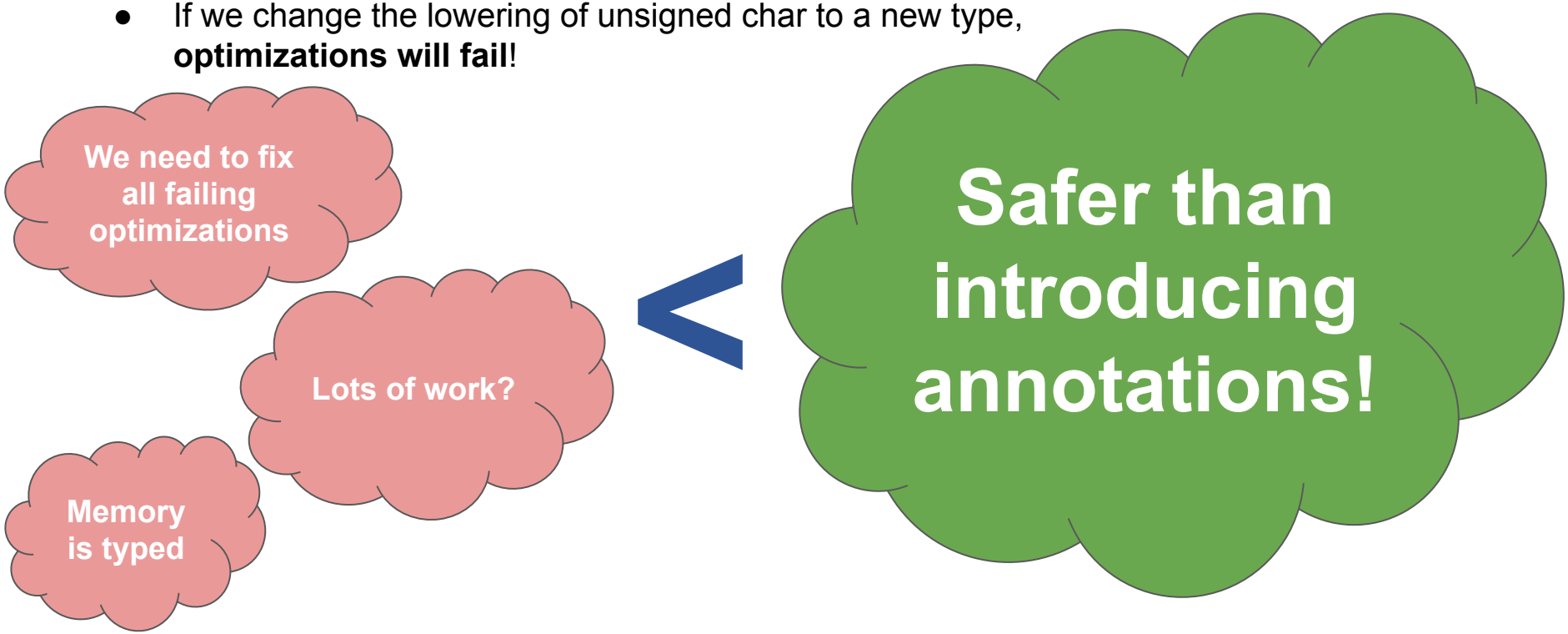
Why a new type?

- We can follow an aggressive strategy as with annotations
- If we change the lowering of unsigned char to a new type, **optimizations will fail**!



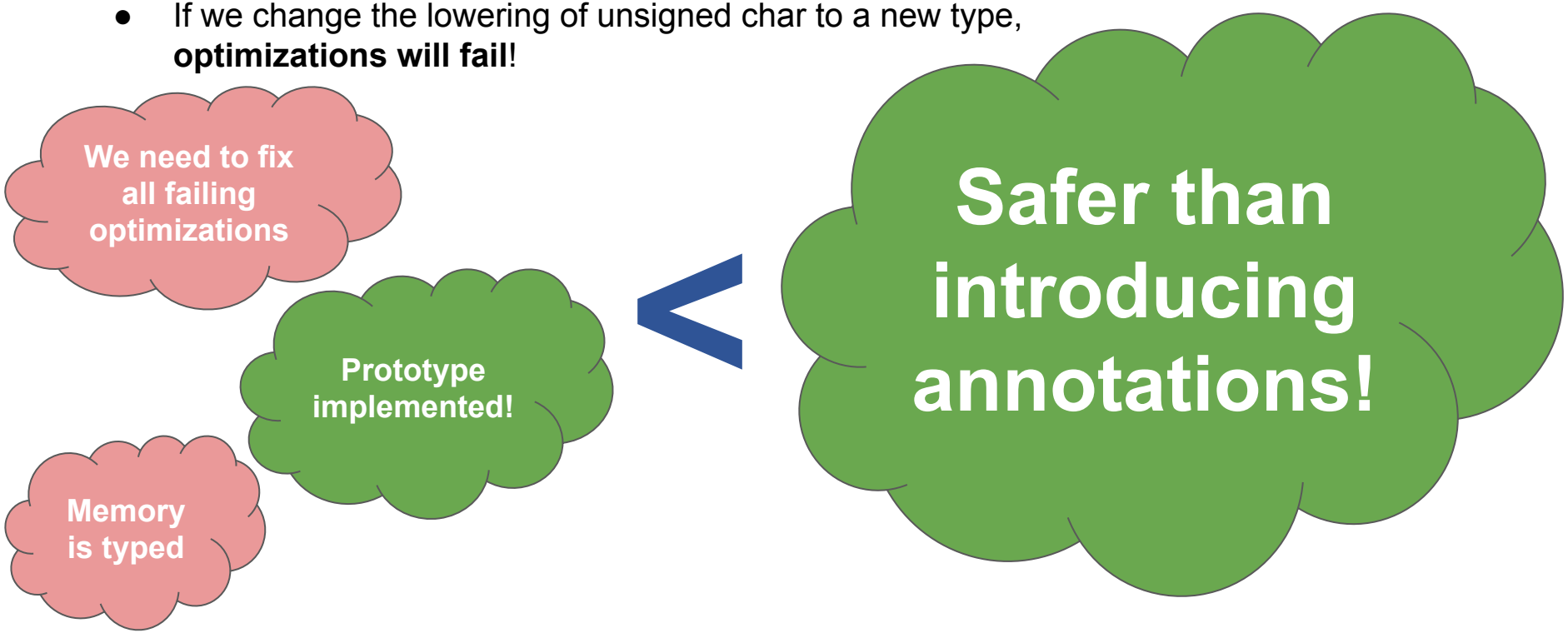
Why a new type?

- We can follow an aggressive strategy as with annotations
- If we change the lowering of unsigned char to a new type, **optimizations will fail**!



Why a new type?

- We can follow an aggressive strategy as with annotations
- If we change the lowering of unsigned char to a new type, **optimizations will fail**!



Preliminary results

Prototype implemented during GSoC 2021

- Added a new byte type family
- Added new cast instruction to handle byte conversions
- Changed Clang to emit byte types for chars
- Changed SelectionDAG to handle byte types as integers
- Fixed unsound optimizations (e.g. memcpy to integer load/store)
- Fixed failing optimizations (SLPVectorize, SROA, GVN, etc.)

Prototype implemented during GSoC 2021

- Added a new byte type family
- Added new cast instruction to handle byte conversions
- Changed Clang to emit byte types for chars
- Changed SelectionDAG to handle byte types as integers
- Fixed unsound optimizations (e.g. memcpy to integer load/store)
- Fixed failing optimizations (SLPVectorize, SROA, GVN, etc.)

Prototype implemented during GSoC 2021

- Added a new byte type family
- Added new cast instruction to handle byte conversions
- Changed Clang to emit byte types for chars
- Changed SelectionDAG to handle byte types as integers
- Fixed unsound optimizations (e.g. memcpy to integer load/store)
- Fixed failing optimizations (SLPVectorize, SROA, GVN, etc.)

Performance evaluation

Program	Compile-time speedup, %	Execution-time speedup, %	Binary size increase, %
500.perlbench_r.	0.38%	-0.88%	-0.98%
502.gcc_r.	0.37%	0.02%	-2.23%
505.mcf_r.	-5.64%	-0.17%	-0.19%
520.omnetpp_r.	-0.08%	-0.46%	-1.01%
523.xalancbmk_r.	0.10%	-4.83%	-0.17%
525.x264_r.	0.22%	-0.40%	-0.01%
531.deepsjeng_r.	0.56%	0.26%	-0.01%
541.leela_r.	0.02%	-0.01%	-0.01%
557.xz_r.	0.19%	-0.91%	1.84%

Performance evaluation

Program	Compile-time speedup, %	Execution-time speedup, %	Binary size increase, %
500.perlbench_r.	0.38%	-0.88%	-0.98%
502.gcc_r.	0.37%	0.02%	-2.23%
505.mcf_r.	-5.64%	-0.17%	-0.19%
520.omnetpp_r.	-0.08%	-0.46%	-1.01%
523.xalancbmk_r.	0.10%	-4.83%	-0.17%
525.x264_r.	0.22%	-0.40%	-0.01%
531.deepsjeng_r.	0.56%	0.26%	-0.01%
541.leela_r.	0.02%	-0.01%	-0.01%
557.xz_r.	0.19%	-0.91%	1.84%

Conclusions

What have we learned?

- LLVM IR semantics has a contradiction
- Untyped memory, integers as universal type holders and current optimizations cannot live together!

What have we learned?

- LLVM IR semantics has a contradiction
- Untyped memory, integers as universal type holders and current optimizations cannot live together!
- We proposed a solution: byte types
- Solution shows promising results

What have we learned?

- LLVM IR semantics has a contradiction
- Untyped memory, integers as universal type holders and current optimizations cannot live together!
- We proposed a solution: byte types
- Solution shows promising results

 # changes: ~2000 lines in LLVM, ~100 in Clang (tests excluded)

What have we learned?

- LLVM IR semantics has a contradiction
- Untyped memory, integers as universal type holders and current optimizations cannot live together!
- We proposed a solution: byte types
- Solution shows promising results

 # changes: ~2000 lines in LLVM, ~100 in Clang (tests excluded)

Missing: byte type optimizations, fixes to Clang tests

What have we learned?

- LLVM IR semantics has a contradiction
- Untyped memory, integers as universal type holders and current optimizations cannot live together!
- We proposed a solution: byte types
- Solution shows promising results

 # changes: ~2000 lines in LLVM, ~100 in Clang (tests excluded)

Missing: byte type optimizations, fixes to Clang tests

Contributions are welcome!