Safely Optimizing Casts between Pointers and Integers

Seoul National Univ. — <u>Juneyoung Lee</u>, Chung-Kil Hur

MPI-SWS — Ralf Jung

University of Utah — Zhengyang Liu, John Regehr

Microsoft Research — Nuno P. Lopes

Overview

	Assembly (x86-64, ARM, ..)	LLVM IR
Pointer	$[0, 2^{64})$	$[0, 2^{64})$ + *provenance*
Integer	$[0, 2^{64})$	$[0, 2^{64})$ + **?**

Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code
for readability

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code
for readability

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

Memory:

0x0

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

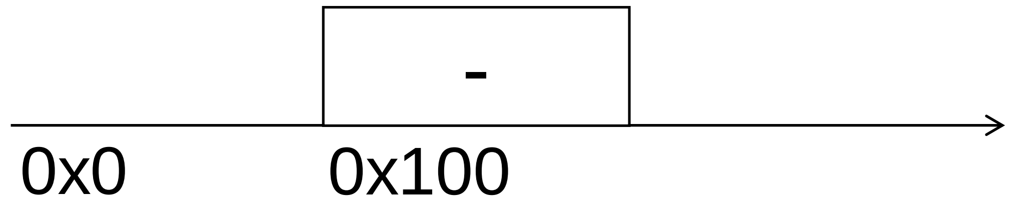
constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

Memory:



0x100

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

Memory:

	-	0
0x0	0x100	0x101

0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

Memory:

	-	0
0x0	0x100	0x101

0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

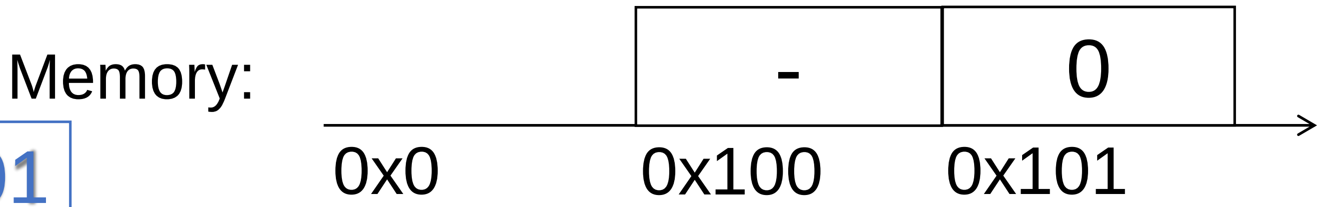
true

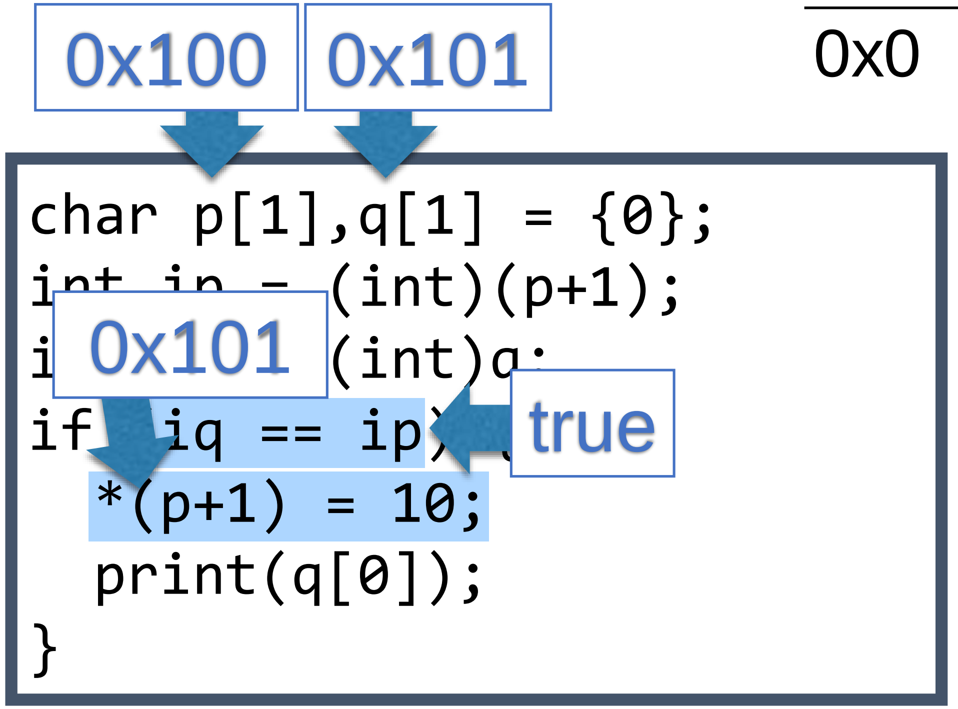
constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

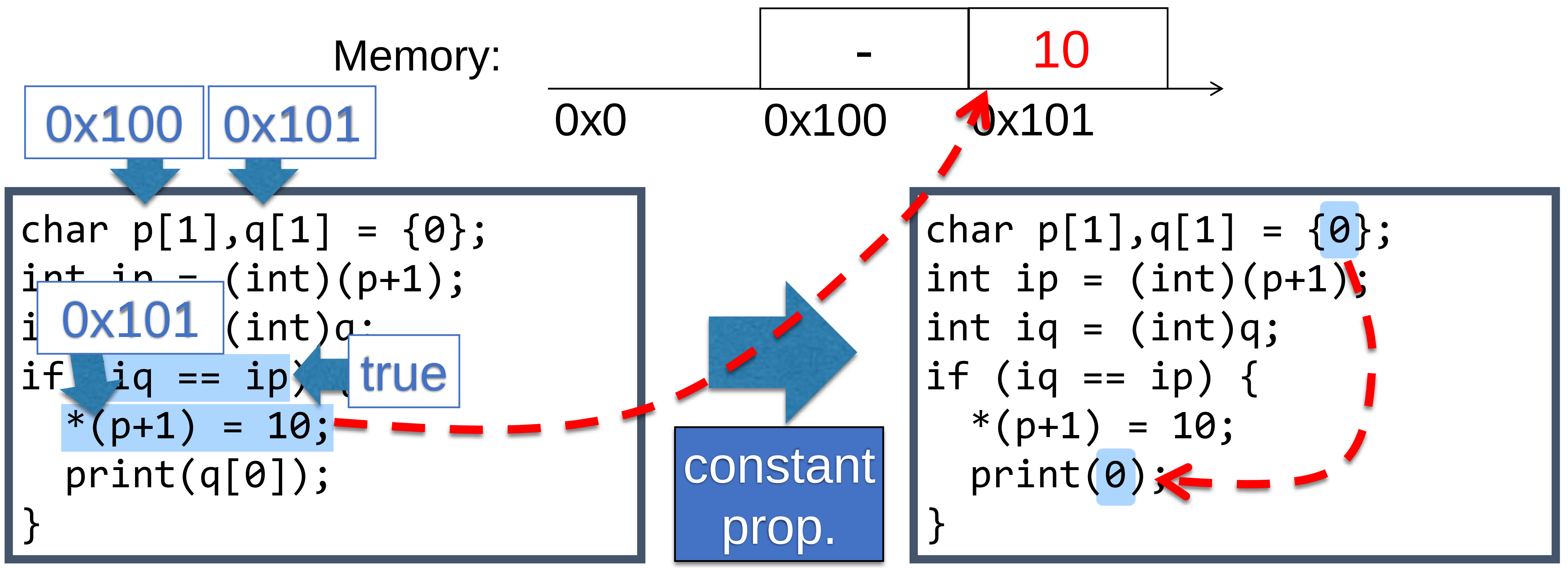





```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

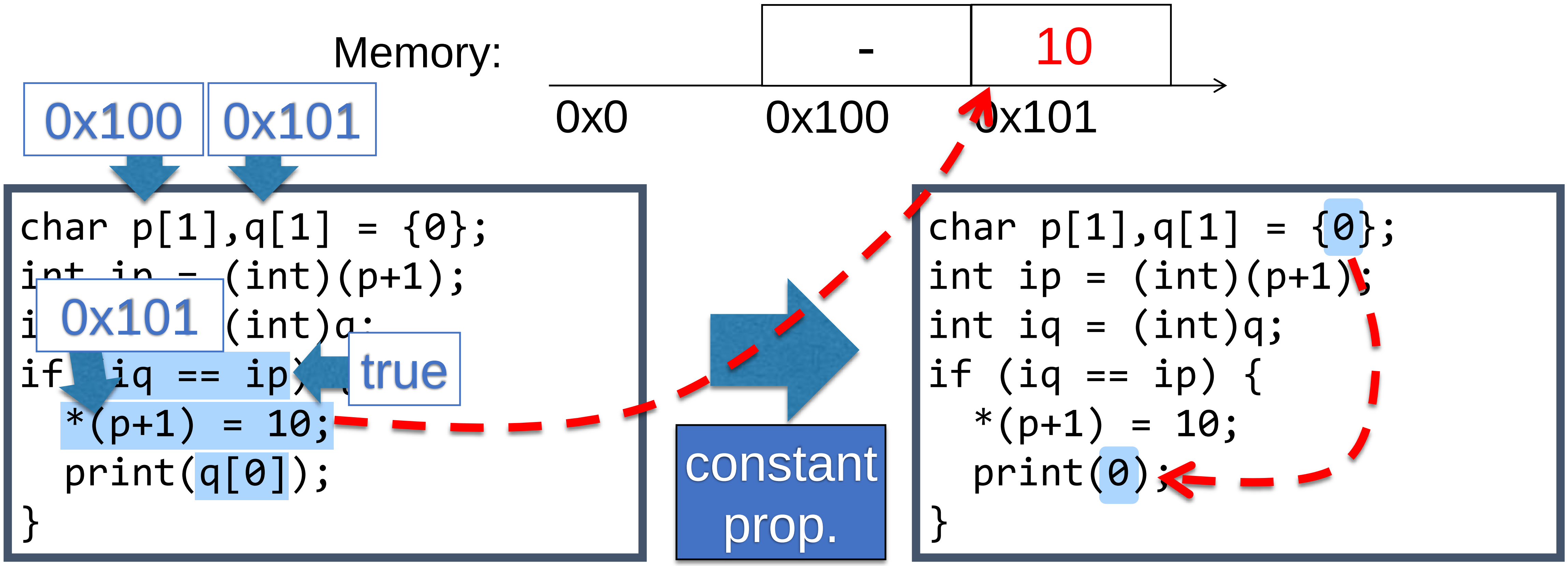
* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer



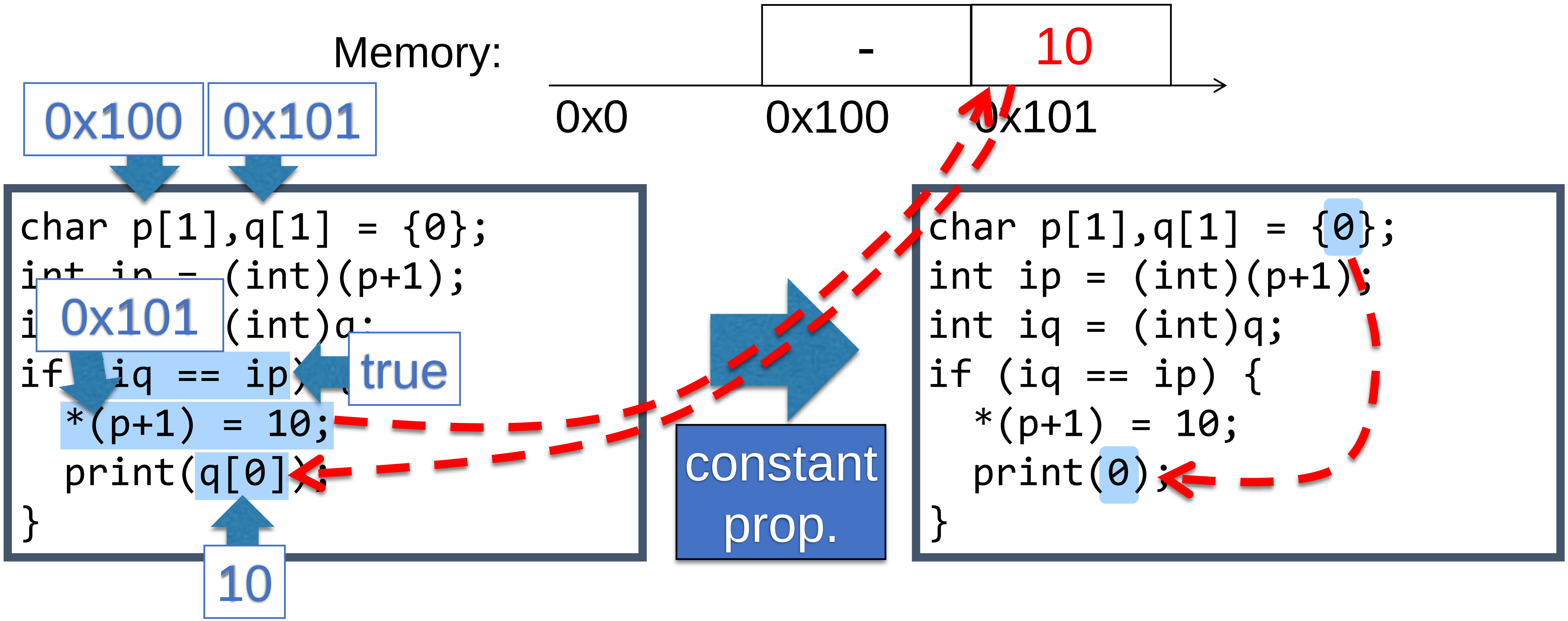
* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer



* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer



* https://godbolt.org/z/9eNt6w

Problem: Pointer as a Pure Integer

Memory: | - | 10 |

Problem with “pointer as a pure integer”

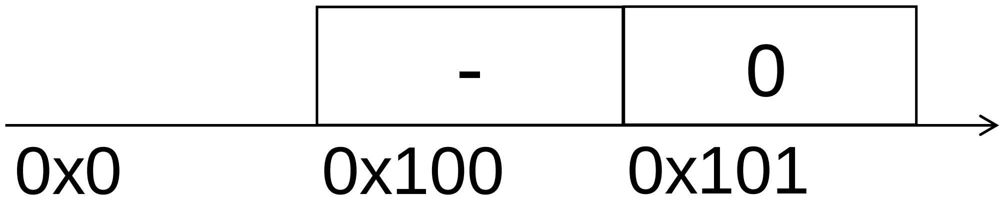
Cannot protect accesses from different blocks

10

* https://godbolt.org/z/9eNt6w

LLVM's Solution:
Pointers have Provenance

Memory:



0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

LLVM's Solution: Pointers have Provenance

Provenance

(p,0x100) (q,0x101)

p: -	q: 0
0x100	0x101

0x0

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

LLVM's Solution: Pointers have Provenance

Provenance

(p,0x100) (q,0x101)

0x0	0x100	0x101
	p: -	q: 0

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

true

constant prop.

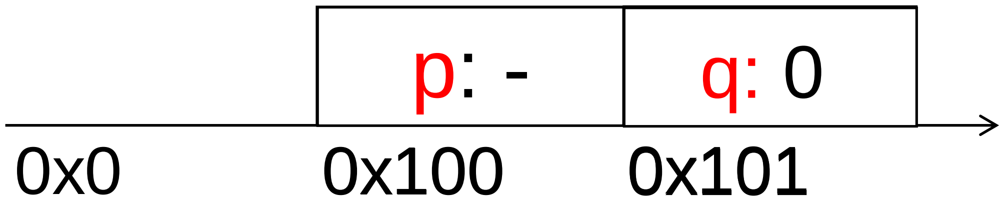
```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

LLVM’s Solution: Pointers have Provenance

Provenance

(p,0x100) (q,0x101)




```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

true

(p,0x101)

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

LLVM's Solution: Pointers have Provenance

Provenance

(p,0x100) (q,0x101)

p: -	q: 0

0x0 0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

true

(p,0x101)

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

* https://godbolt.org/z/9eNt6w

LLVM's Solution: Pointers have Provenance

Provenance

Memory:

(p,0x100) (q,0x101)

p: -	q: 0

0x0 0x100 0x101

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

true

(p,0x101)

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
```

Undefined Behavior
because p ≠ q

* https://godbolt.org/z/9eNt6w

What about Integers?

	Assembly (x86-64, ARM, ..)	LLVM IR ✓
Pointer	$[0, 2^{64})$	$[0, 2^{64})$ + *provenance*
Integer	$[0, 2^{64})$	$[0, 2^{64})$ + **?**

Casting

Miscompilation with PtrIntCast

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

10

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

10

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

10

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Miscompilation with PtrIntCast

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
```

We found this miscompilation bug in both LLVM & GCC

```
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Which pass is responsible for it?

Problem depends on the model

Integer with provenance cannot explain

int. eq. prop.

```
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
```

cast elim.

Integer without provenance cannot explain

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

```
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

constant prop.

Integer-With-Provenance Model

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Integer-With-Provenance Model

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Integer-With-Provenance Model

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

Has provenance **q**

cast elim.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1]
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Integer-With-Provenance Model

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

Has provenance **p**

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

Has provenance **q**

cast elim.

```
... = {0};
... (p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1]...
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Integer-Without-Provenance Model

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

cast elim

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Integer-Without-Provenance Model

```
char p[1]
int ip =
int iq =
if (iq ==  {
  *(char*)(int)(p+1)=10;
  print(q[0]);
}
```

Provenance p **removed**

int. eq. prop.

```
char p[1],q[1]={0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

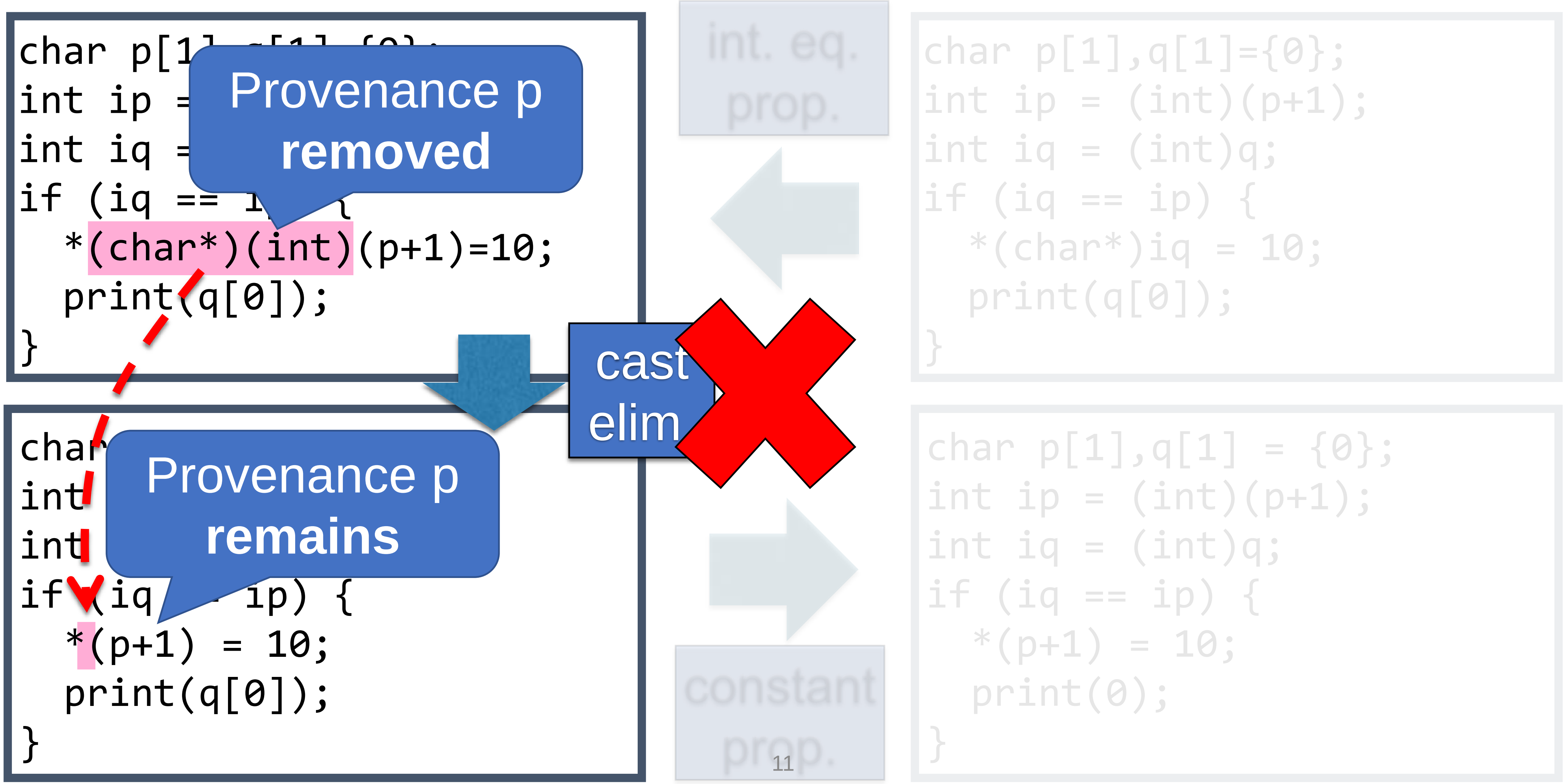
cast elim

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(0);
}
```

Integer-Without-Provenance Model



Integer-With-Provenance is Unnatural

- Hard to explain integer equality propagation
- Hard to explain many other transformations as well

```
r = (i + j) - k        ⇔        r = i + (j – k)

r = (int)(float)j      ⇒        r = j
```

Integer-With-Provenance is Unnatural

- Hard to explain integer equality propagation
- Hard to explain many other transformations as well

prov. + prov.? prov. – prov.?

```
r = (i + j) - k        ⇔    r = i + (j – k)

r = (int)(float)j      ⇒    r = j
```

Integer-With-Provenance is Unnatural

- Hard to explain integer equality propagation
- Hard to explain many other transformations as well

```
r = (i + j) - k        ⇔    r = i + (j – k)
```

prov. + prov.?

prov. – prov.?

```
r = (int)(float)j      ⇒    r = j
```

provenance in float types?

Our Suggestion [OOPSLA'18]: Integer-Without-Provenance Model

	Assembly (x86-64, ARM, ..)	**LLVM IR**
Pointer	$[0, 2^{64})$	$[0, 2^{64})$ + *provenance*
Integer	$[0, 2^{64})$	$\mathbf{[0, 2^{64})}$

Integer-Without-Provenance Model

- Semantics of Casts
- Problematic Optimizations
- How to Recover Performance?

Semantics of Casts [OOPSLA'18]

1. Pointer-to-integer casts remove provenance
2. Integer-to-pointer casts gain full provenance

How to regain protection from unknown accesses?

By exploiting <u>nondeterministic allocation</u>

How to perform in-bounds checking on full-provenance pointers?

By recording in-bounds offsets at the pointer & checking <u>when dereferenced</u>

Optimizations Unsound in Our Model

1. Cast Elimation

```
p2 = (char*)(int)p
```
⇨
```
p2 = p
```

2. Integer Comparison to Pointer Comparison

```
c = icmp eq (int)p, (int)q
```
⇨
```
c = icmp eq p, q
```

Optimizations Unsound in Our Model

1. Cast Elimation

```
p2 = (char*)(int)p
```

⇨

```
p2 = p
```

Full provenance

Provenance p

2. Integer Comparison to Pointer Comparison

```
c = icmp eq (int)p, (int)q
```

⇨

```
c = icmp eq p, q
```

Optimizations Unsound in Our Model

1. Cast Elimation

```
p2 = (char*)(int)p
```
Full provenance

⇨

```
p2 = p
```
Provenance p

2. Integer Comparison to Pointer Comparison

```
c = icmp eq (int)p, (int)q
```
Comparison of integers

⇨

```
c = icmp eq p, q
```
Comparison of pointers

Performance Issue

- **Cast elimination removes significant portion of casts**
  - 13% of ptrtoints, 40% of inttoptrs from C/C++ benchmarks *
- **Disabling cast elimination hinders other optimizations**
  - ptrtoint makes variables escaped
  - inttoptr is regarded as pointing to an unknown object
- **Disabling cast elimination causes slowdown**
  - 1% slowdown in perlbench_r, blender_r

* SPEC2017rate + LLVM test-suite, -O3

Our Solution

1. **Do not generate Ptr↔Int casts in the first place**
   - 86% of Ptr↔Int casts are introduced by LLVM, not by programmers
     - Ptr → Int casts are generated from pointer subtractions
     - Int → Ptr casts are from canonicalizing loads/stores as int types
   - **How:** by introducing new features
2. **Allow the previous optimizations conditionally**
   - **How:** by developing an analyzer to check such conditions

To reduce Ptr→Int Casts: Introduce Pointer Subtraction Operation

Before Fix (Uses ptrtoint)

```
ip = ptrtoint p
iq = ptrtoint q
i = ip - iq
```

After Fix (Uses psub)

```
i = psub p, q
```

$$\texttt{psub}\ p,\ q \overset{\text{def}}{\equiv} \begin{cases} p - q & \text{If } prov(p) = prov(q) \lor prov(p) = \texttt{full} \lor prov(q) = \texttt{full} \\ \texttt{poison} & \text{Otherwise} \end{cases}$$

To reduce Int→Ptr Casts: Stop Canonicalizing Loads/Stores as Ints

```
v = load i64* p
v2= load i8** p
```

* https://godbolt.org/z/y48Mkt

To reduce Int→Ptr Casts: Stop Canonicalizing Loads/Stores as Ints

```
v = load i64* p
v2= load i8** p
```

⇩

```
v = load i64* p
v2= inttoptr v
```

* https://godbolt.org/z/y48Mkt

To reduce Int→Ptr Casts:
Stop Canonicalizing Loads/Stores as Ints

```
v = load i8** p
v2= load i8** p
```

⇩

```
v = load i64* p
v2= load i8** p
```

⇩

```
v = load i64* p
v2= inttoptr v
```

* https://godbolt.org/z/y48Mkt

To reduce Int→Ptr Casts: Stop Canonicalizing Loads/Stores as Ints

```
v = load i8** p
v2= load i8** p
```

```
v = load i64* p
v2= load i8** p
```

```
v = load i64* p
v2= inttoptr v
```

Use 'd64' (data type) instead

	Has Provenance	Supports Integer operations
d64	Yes	No
i64	No	Yes

Unlike cast between int⇔ptr, d64⇔ptr preserves provenance.

* https://godbolt.org/z/y48Mkt

To reduce Int→Ptr Casts: Stop Canonicalizing Loads/Stores as Ints

```
v = load i8** p
v2= load i8** p
```

Use ‘d64’ (data type) instead

```
v = load i64* p
v2= load i8** p
```

```
v = load i64* p
v2= inttoptr v
```

	Has Provenance	Supports Integer operations
d64	Yes	No
i64	No	Yes

Unlike cast between int⇔ptr, d64⇔ptr preserves provenance.

* https://godbolt.org/z/y48Mkt

Conditionally Allowing Cast Elimination

```
// p and q have same underlying object
```

```
p2 = inttoptr(ptrtoint p)
c  = icmp eq/ne p2, q
```

⇨

```

c  = icmp eq/ne p, q
```

```
p2 = inttoptr(ptrtoint p)
c  = psub p2, q
```

⇨

```

c  = psub p, q
```

- More examples & descriptions are listed at https://github.com/aqjune/eurollvm19

Evaluation: the # of Casts

Disable unsound opts.

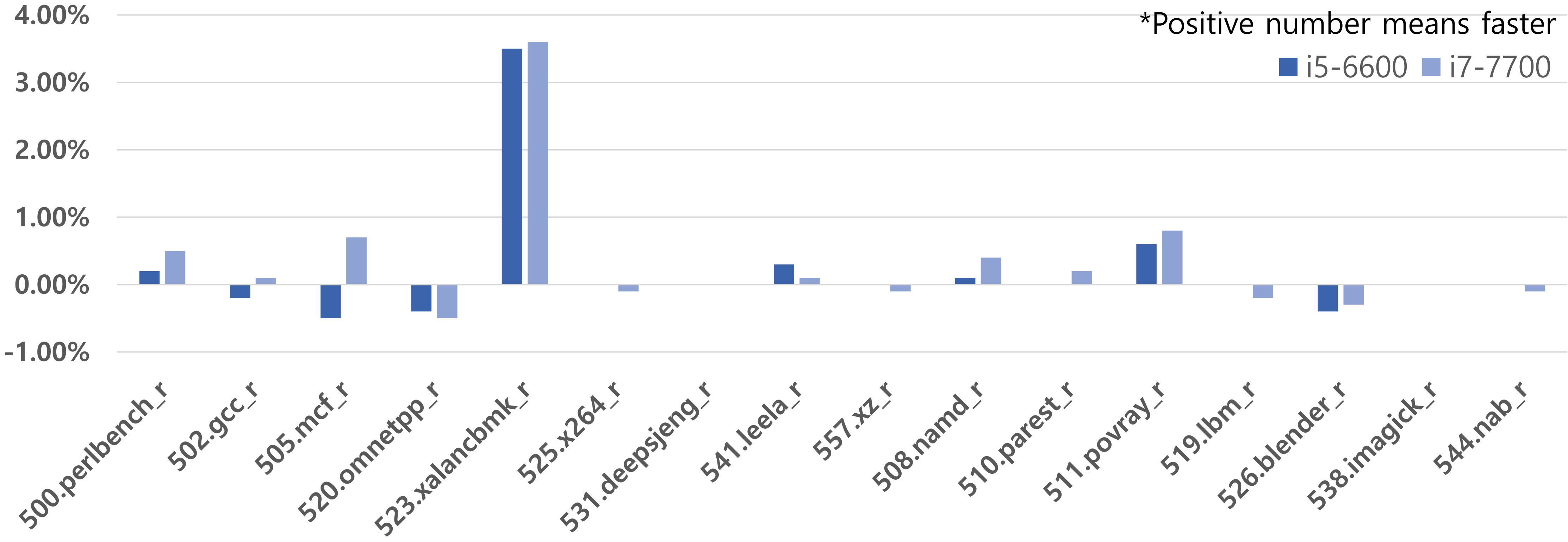
Add psub, stop load/store to int

Conditionally allow cast elim.

		Baseline (LLVM 8.0)	**No Cast Fold**	**Reduce Cast Introduction**	**Conditionally Fold**
Before O3	# of ptrtoints	44K	44K	14K	14K
	# of inttoptrs	1.5K	1.5K	1.5K	1.5K
After O3	# of ptrtoints	57K	66K	11K	11K
	# of inttoptrs	29K	45K	5K	4.8K

- C/C++ benchmarks of SPEC2017rate + LLVM Nightly Tests used
- 81% of ptrtoints / 83% of inttoptrs removed (compared to baseline)

Evaluation: Performance Impact



<SPEC2017rate Speedup>

- LLVM Nightly Tests (C/C++): ~0.1% avg. slowdown (-1% ~ 3.6%)

Conclusion

- Provenance helps compiler do more optimizations on pointers
- Integer with provenance works badly with integer optimizations
- We suggest separating pointers/integers conceptually
- We show how to regain performance after removing invalid optimizations

https://github.com/aqjune/eurollvm19

Conclusion

- Provenance helps compiler do more optimizations on pointers

We're updating Alive
to support
pointer-integer casts! ☺

```
PROGRAM: Name: ptrintload3
  ENTRY:
    v16 = ptrtoint i8* p1 to i16
    p2 = inttoptr i16 v16 to i8*
    v2 = load i8* p2
    v1 = load i8* p1
PRECONDS:
        Instruction "v2 = load i8* p2" has no UB.
CHECK:
        Instruction "v1 = load i8* p1" has no UB?
        v1 === v2?
Result: INCORRECT
```

https://github.com/aqjune/eurollvm19

supplementary slides

Constant Propagation and Readonly function

```
char p[1],q[1] = {0};


if (foo(p, q)) { //readonly
  *(p+i) = 10;
  print(q[0]);
}
```

constant prop.

```
char p[1],q[1] = {0};


if (foo(p, q)) { //readonly
  *(p+i) = 10;
  print(0);
}
```

Constant Propagation and Readonly function

```
char p[1],q[1] = {0};
```

return (int)(p+1) == (int)q?

```
if (foo(p, q)) { //readonly
  *(p+i) = 10;
  print(q[0]);
}
```

1?

constant prop.

```
char p[1],q[1] = {0};


if (foo(p, q)) { //readonly
  *(p+i) = 10;
  print(0);
}
```

Integer Equality Propagation and Performance

- Performed by many optimizations
  - CVP, Instruction Simplify, GVN, Loop Exit Value Rewrite, …
- Reduces code size

  -10% in minisat, -6% in smg2000, -4% in simple_types_constant_folding, …
- Boosts performance in small benchmarks
  - x2000 speedup in nestedloop

Sound Optimizations that are already in LLVM

```
gep(p, -(int)q)  ➡  (void*)((int)p-(int)q)
```

```
select (p==null), p, null  ➡  null // null=(void*)0
```

Rationale

It is safe to replace p with (void*)(int)p.

Delayed Inbounds Checking

```
p = (char*)0x100 // p=(0x100,*)
p2 = gep p, 1    // p=(0x101,*)

p3 = gep inbounds p, 1
                 // p = (0x101,*,{0x100,0x101})

load p3          // 0x100, 0x101 should be
                 // in-bounds addrs of the
                 // object at 0x101
```