LLVM 컴파일러의 Alias 분석 검산하기

윤용호* 강지훈** 허충길**

서울대학교

2015.01.28 뻔개발표 @ ROSAEC Workshop

*ROPAS **Software Foundations Lab.

아~ 최적화 하고싶다~ 1

```
int foo(int q) {
  p = 4;
  p = q;                    ?
  return p;
}
```

아~ 최적화 하고싶다~ 1

```
int foo(int q) {
  p = 4;
  p = q;
  return p;
}
```

Good!

최적화 하고싶다 2

```
int foo(int *p, int *q) {
  *p = 4;
  *p = *q;                ?
  return *p;
}
```

최적화 못 함

```
int foo(int *p, int *q) {
  *p = 4; // ??
  *p = *q;
  return *p;
}
```

최적화 못 함

```
int foo(int *p, int *q) {
  *p = 4; // *x = 4
  *p = *q; // *x = *x
  return *p;
}

int x = 0;
foo(&x, &x);
```

이럴 수도 있으니까.

최적화 하려면

```
int p[100];
int q[100];

int foo() {
  *p = 4;
  *p = *q; // p, q: no alias
  return *p;
}
```

"두 전역 변수는 alias될 수 없다" -> p≠q

근거 필요

Alias Analysis가 여러 최적화의 근거로 쓰임

LLVM 구현에서는

```
bool DSE::runOnBasicBlock(BasicBlock &BB) {
      ...
      // If we find a write that is a) removable (i.e., non-volatile), b) is
      // completely obliterated by the store to 'Loc', and c) which we know that
      // 'Inst' doesn't load from, then we can remove it.
      if (isRemovable(DepWrite) && isCompleteOverwrite(Loc, DepLoc, *AA) &&
          !isPossibleSelfRead(Inst, Loc, DepWrite, *AA)) {

        // Delete the store and now-dead instructions that feed it.
        DeleteDeadInstruction(DepWrite, *MD);
        MadeChange = true;
      ...
}
```

```
static bool isPossibleSelfRead(Instruction *Inst,
                               const AliasAnalysis::Location &InstStoreLoc,
                               Instruction *DepWrite, AliasAnalysis &AA) {
  // Self reads can only happen for instructions that read memory.  Get the
  // location read.
  AliasAnalysis::Location InstReadLoc = getLocForRead(Inst, AA);
  if (InstReadLoc.Ptr == 0) return false;  // Not a reading instruction.

  // If the read and written loc obviously don't alias, it isn't a read.
  if (AA.isNoAlias(InstReadLoc, InstStoreLoc)) return false;
  ...
}
```

clang llvm 컴파일 과정

```
int p[100];
int q[100];

int foo() {
  *p = 4;
  *p = *q;
  return *p;
}
```

clang -O0 →

```
p = global(100*32)
q = global(100*32)

foo() {
  *p = 4
  t1 = *q
  *p = t1
  t2 = *p
  return t2
}
```

dse →

```
p = global(100*32)
q = global(100*32)

foo() {
  *p = 4
  t1 = *q
  *p = t1
  t2 = *p
  return t2
}
```

↓ store-load

```
p = global(100*32)
q = global(100*32)

foo() {
  t1 = *q
  *p = t1
  t2 = *p
  return t1
}
```

← dce

```
p = global(100*32)
q = global(100*32)

foo() {
  t1 = *q
  *p = t1
  t2 = *p // DCE
  return t1
}
```

```
p = global(100*32)
q = global(100*32)

foo() {
  t1 = *q
  *p = t1
  return t1
}
```

clang llvm 컴파일 과정

```
int p[100];
int q[100];

int foo() {
  *p = 4;
  *p = *q;
  return *p;
}
```

clang -O0 →

```
p = global(100*32)
q = global(100*32)

foo() {
  *p = 4
  t1 = *q
  *p = t1
  t2 = *p
  return t2
}
```

dse →

```
p = global(100*32)
q = global(100*32)

foo() {
  *p = 4
  t1 = *q
  *p = t1
  t2 = *p
  return t2
}
```

p와 q는 서로 다른 전역 변수
-> p와 q는 No Alias
-> t1 = *q
-> *p = t1은 *p = 4 결과를 완전히 덮어쓴다
-> *p = 4는 Dead Store이다. -> 지우자!

↓ store-load

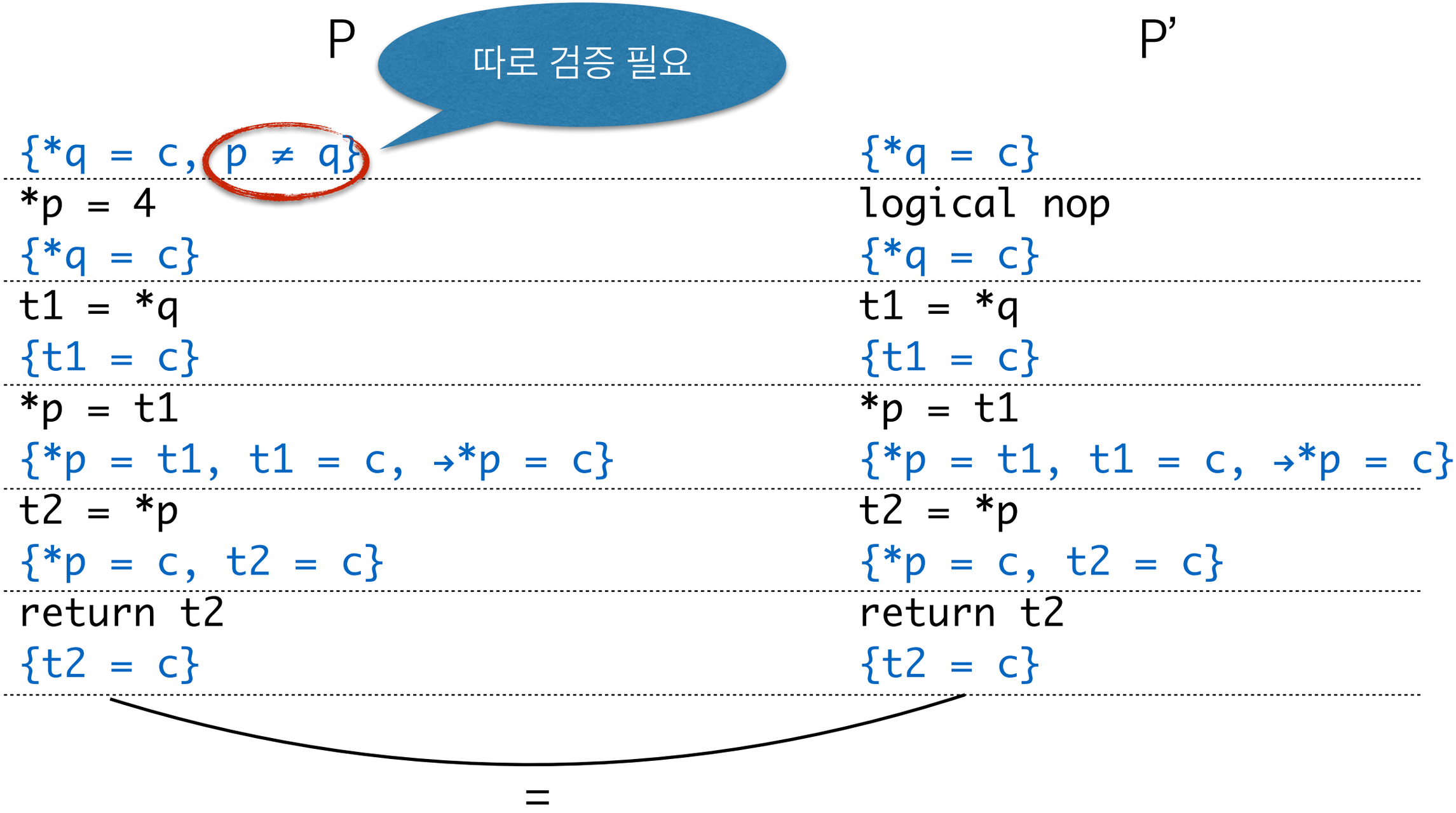
```
p = global(100*32)
q = global(100*32)

foo() {
  t1 = *q
  *p = t1
  t2 = *p // DCE
  return t1
}
```

```
p = global(100*32)
q = global(100*32)

foo() {
  t1 = *q
  *p = t1
  t2 = *p
  return t1
}
```

SimpleBerry에서 DSE를 검산하려면 예를 들어



SimpleBerry와 조금 다른

- SimpleBerry는 두 프로그램 P와 P’의 시뮬레이션 관계를
- Alias 분석 검산에서는 한 프로그램에서 두 포인터의 관계를

이런 식으로?

```
p = global(100*32)
{alc(p)}
q = global(100*32)
{alc(q), p ≠ q}
…
{p ≠ q}
*p = 4
```

결론

- Alias 분석은 제어 흐름을 고치지 않는 최적화 패스에서 중요한 한 축을 담당

- 검증된 검산기를 붙일 것임