C 编译器优化过程中的 Bug

一个朋友向我指出一个最近他们发现的 GCC 编译器优化过程（加上 -O3 选项）里的 bug，导致他们的产品出现非常诡异的行为。这使我想起以前见过的一个 GCC bug。当时很多人死活认为那种做法是正确的，跟他们说不清楚。简言之，这种有问题的优化，喜欢利用 C 语言的“未定义行为”（undefined behavior）进行推断，最后得到奇怪的结果。

这类优化过程的推理方式都很类似，他们使用一种看似严密而巧妙的推理，例如：“现在有一个整数 `x`，我们不知道它是多少。但 `x` 出现在一个条件语句里面，如果 `x > 1`，那么程序会进入未定义行为，所以我们可以断定 `x` 的值必然小于或者等于 1，所以现在我们利用 `x ≤ 1` 这个事实来对相关代码进行优化……”

看似合理，然而它却是不正确的，你能看出来这样的推理错在何处吗？我一时想不起来之前具体的例子了（如果你知道的话告诉我）。上网搜了一下相关话题，发现这篇 Chris Lattner (LLVM 和 Swift 语言 的设计者) 写于 2011 年的文章。文中指出，编译器利用 C 语言的“未定义行为”进行优化，是合理的，对于性能是很重要的，并且举出这样一个例子：

```
void contains_null_check(int *P) {
  int dead = *P;
  if (P == 0)
    return;
  *P = 4;
}
```

这例子跟我之前看到的 GCC bug 不大一样，但大致是类似的推理方式：这个函数依次经过这样两个优化步骤（RNCE 和 DCE），之后得出“等价”的代码：

```
void contains_null_check_after_RNCE(int *P) {
  int dead = *P;
  if (false)  // P 在上一行被访问，所以这里 P 不可能是 null
    return;
  *P = 4;
}
```

```
void contains_null_check_after_RNCE_and_DCE(int *P) {
  //int dead = *P;     // 死代码消除
  //if (false)         // 死代码
  //  return;          // 死代码
  *P = 4;
}
```

他的推理方式是这样：

1. 首先，因为在 `int dead = *P` 里面，指针 `P` 的地址被访问，如果程序顺利通过了这一行而没有出现未定义行为（比如当掉），那么之后 `P` 就不可能是 null，所以我们可以把 `P == 0` 优化为 `false`。
2. 因为条件是 `false`，所以整个 if 语句都是死代码，被删掉。
3. dead 变量赋值之后，没有被任何其它代码使用，所以对 dead 的赋值是死代码，可以消去。

最后函数就只剩下一行代码 `*P = 4`。然而经我分析，发现这个优化转换是根本错误的做法（unsound 的变换），而不只是像他说的“存在安全隐患”。现在我来考考你，你知道这为什么是错的吗？值得庆幸的是，现在如果你把这代码输入到 Clang，就算加上 -O3 选项，它也不会给你进行这个优化。这也许说明 Lattner 的这个想法后来已经被 LLVM 团队抛弃。

我写这篇文章的目的其实是想告诉你，不要盲目的相信编译器的作者们做出的变换都是正确的，无论它看起来多么的合理，只要打开优化之后你的程序出现奇葩的行为，你就不能排除编译器进行了错误优化的可能性。Lattner 指出这样的优化完全符合 C 语言的标准，这说明就算你符合国际标准，也有可能其实是错的。有时候，你是得相信自己的直觉……