CS 6110 — Final Report, Alive2

Gavin Gray u1040250

1 Introduction

Software bugs are a real issue and as systems grow larger reasoning about correctness becomes more and more difficult. No type of bug is "better" than another, however, I would argue that compiler bugs pose a greater issue. Reasoning about a compiler bug is difficult as it most often requires sifting through assembly code which is tedious and obfuscates the problem. Developers are taught to trust the compiler which means that this is the last place they will look when tracking down software bugs. Alive2 is a tool created by Nuno P. Lopes[1] and John Regehr[2] which takes an unoptimized function in LLVM IR and proves whether or not the optimized version is a *refinement* or not. In the case it isn't, a counter example is given. Alive2 specifically talks about refinement when proving correctness, this is due to the nature of what a compiler does. For example, LLVM is an intermediate representation that currently supports the compiling of Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift [7]. All of these languages have different semantics and the reason LLVM can compile all of them is that it changes the meaning of the code in allowed ways. An example of an allowed refinement is shown in *listing 1*.

```
define i16 @src(i16 %x, i16 %y) {
    %r = sdiv i16 %x, %y
    ret i16 %r
}

define i16 @tgt(i16 %x, i16 %y) {
    %z = icmp eq i16 %y, 0
    br i1 %z, label %zero, label %nonzero
zero:
    ret i16 8888
nonzero:
    %q = sdiv i16 %x, %y
    ret i16 %q
}
```

Listing 1: Simple division refinement [5]

In the `src` function of *listing 1* the division between arguments `%x` and `%y` is returned. In the target function, we can see that this division is also returned except in the case when `%y = 0`, here the integer `8888` is returned. This is a valid refinement because in the source function a division by zero produces undefined behavior and in the target it has merely been

[1] https://web.ist.utl.pt/nuno.lopes/

[2] https://www.cs.utah.edu/~regehr/

refined to always return `8888`. The key here is that all properties of the source function are preserved and no additional undefined behavior is introduced.

2 Bug 39861

The exploration of Alive2 and how it goes about proving refinements was done through one particular bug report [3]. The relevant source and target functions are provided in *listing 2*.

```
define i1 @src(i8 %x, i8 %y) {
  %tmp0 = lshr i8 255, %y
  %tmp1 = and i8 %tmp0, %x
  %ret = icmp sge i8 %tmp1, %x
  ret i1 %ret
}

define i1 @tgt(i8 %x, i8 %y) {
  %tmp0 = lshr i8 255, %y
  %1 = icmp sge i8 %tmp0, %x
  ret i1 %1
}
```

Listing 2: Test case used to find bug 39861

Let's try to develop some intuition behind this particular peephole optimization included by LLVM. In the source function we are taking a 16 bit integer `%x`, masking it, then comparing the masked value with the original. If the highest 'set' bit of the mask is greater than or equal to the highest set bit of `%x` then the masked result will always be `%x`; on the other hand, the result will be less than `%x`. Let's see this in an example where `x = 22`:

```
mask = 01111111
x    = 00010110
x & mask >= x    ; true (lhs equal to x)

mask = 00011111
x    = 00010110
x & mask >= x    ; true (lhs equal to x)

mask = 00001111
x    = 00010110
x & mask >= x     ; false (lhs less than x)
```

In all of the above examples, we would get the exact same results if we skipped the step of `x & mask` and simply did the comparison against `mask`. If you stare at this problem long

enough you can convince yourself that this is a valid transformation for *most inputs*, however, there is one case in which this fails. Before discussing how and why this transformation fails, let's first see how Alive2 encodes the problem in SMTLIB and passes it to Z3.

3 Z3 Encoding

In the *Introduction 1* we saw that Alive2 deals with *refinement* and that this is an important thing to understand. When talking about function transformations it is always important to recognize the possibility of undefined behavior, namely *poison* and *undef* values. In short, an *undef* value represents a random value from the set of all possible defined values. For integers, one can think of this as an uninitialized register. *Poison* values represent invalid assumptions, such as `x < x + 1` when a signed overflow occurs [1]. What makes reasoning about code transformation difficult is preserving defined behavior while ensuring that no new undefined behavior is introduced. Let's look at a simple example of transforming the bug shown in *Listing 2* while first *ignoring* possible undefined behavior.

```
1 ; ./alive-tv --disable-poison-input --disable-undef-input --smt-verbose
    src.ll tgt.ll
2 (declare-fun %y () (_ BitVec 8))
3 (declare-fun %x () (_ BitVec 8))
4 (assert (let ((a!1  (=
5                 (bvsle %x (bvand %x (bvlshr #xff %y))) ; src
6                 (bvsle %x (bvlshr #xff %y)))))         ; tgt
7   (not (or (not (bvule %y #x07)) a!1))))
```

Listing 3: Z3 Query with Poison and Undef Inputs Disabled

As can be seen in *listing 3* the source and target functions are translated almost instruction for instruction from LLVM to SMTLIB. This query is strictly avoiding both poison and undef inputs. This is shown on line 6 where it is checked that `%y <= #x07`. If `%y` were any larger than seven this would cause a poison value when performing the right shift because the shift is larger than the width of the left-hand-side [2].

```
1 ; ./alive-tv --disable-undef-input --smt-verbose src.ll tgt.ll
2 (declare-fun %y () (_ BitVec 8))
3 (declare-fun %x () (_ BitVec 8))
4 (declare-fun np_%x () Bool)
5 (declare-fun np_%y () Bool)
6 (assert (let ((a!1 (not (and np_%y (bvule %y #x07) np_%x)))
7               (a!2 (= (bvsle %x (bvand %x (bvlshr #xff %y)))
8                       (bvsle %x (bvlshr #xff %y)))))
9   (not (or a!1 a!2))))
```

Listing 4: Z3 Query with Poison Values

In *listing 4* we have allowed the addition of poison inputs. We can see that the overall query was not altered very much. There was an addition of two variables `%np_%x` and `%np_%y` which stand for "not poison %x/%y". Other than this variable addition, the query remains almost unchanged. The value `a!2` represents the same value as in *listing 3*, namely the equivalence of function outputs, and the value of `a!1` is simply checking whether `%x` or `%y` is a poison value, or if `%y` is large enough to cause a poison value. Poison values are fairly uninteresting as they merely get propagated throughout the code. Let's start to look at some code that handles undef values and see how this complicates the queries tremendously.

```
; ./alive-tv --disable-poison-input --smt-verbose src.ll tgt.ll
(declare-fun isundef_%y () (_ BitVec 1))
(declare-fun isundef_%x () (_ BitVec 1))
(declare-fun undef!4 () (_ BitVec 8))
(declare-fun undef!3 () (_ BitVec 8))
(declare-fun %x () (_ BitVec 8))
(declare-fun %y () (_ BitVec 8))
; ... truncated code ...
  ; undef!0 : %x from src for 'bvand'
  ; undef!1 : %y from src
  ; undef!2 : %x from src for 'bvsge'
  ; undef!3 : %x from tgt
  ; undef!4 : %y from tgt
  (and (forall ((undef!0 (_ BitVec 8))
                (undef!1 (_ BitVec 8))
                (undef!2 (_ BitVec 8)))
       (! (let ((a!1 ; a!1 = ! ( undef!1 <= 7  &&  undef!4 <= 7 )
                 (not (and (= ((_ extract 7 3) undef!1) #b00000)
                           (= ((_ extract 7 3) undef!4) #b00000))))
               (a!2
                ; a!2 = ~( ~undef!0 | ~( 255 >> undef!1 ) )
                ; a!2 = ( undef!0 & ( 255 >> undef!1 ) )
                (bvnot (bvor (bvnot undef!0)
                             (bvnot (bvlshr #xff undef!1))))))
           (let ((a!3
                  (or a!1 ; creating a poison value
                   (=
                    (bvsle undef!2 a!2) ; undef!2 is %x from src
                    (bvsle undef!3 (bvlshr #xff undef!4))))))
             (not a!3)))
        :weight 0))
   (= isundef_%x #b1)
   (= isundef_%y #b1))))
; ... truncated code ...
```

Listing 5: Z3 Query with Undef Values

In *listing 5* we can see part of the query used when undef values are allowed. Code annotations have been added to help understanding but let's walk through it to see how these undef values are handled. The first thing to note, is that this is one of 2^2 cases that need to be handled, specifically, the case when both `%x` and `%y` are undef. This can bee seen on lines $31, 32$. The first obvious change is the introduction of variables `undef!0-4`, why are these necessary? Reflecting on the definition of undef tells us that undef values are essentially random variables. On line 13 there is the start to a `forall` quantifier which quantifies over the variables `undef!0, undef!1, undef!2`, which correspond to the 3 variables reads from the source function. Likewise, the variables `undef!3, undef!4` correspond to the 2 variable reads in the target function. These are defined at the global scope whereas the source variables are defined in the local quantifier because for all possible values in the source they need to refine the target. We also observe the inclusion of the `(! ... :weight 0)` annotation and it so happens that this can merely be ignored. Setting the weight of a quantifier can be helpful in some circumstances but for this particular encoding all of the quantifiers have weight 0 meaning that they don't have any influence whatsoever. Besides the `forall` and `(! ...)` constructs the logic within the query remains almost unchanged, however, now instead of using the values for `%x` and `%y` we use those for the `undef` variables instead.

We've now seen what happens when both `%x` and `%y` are undefined values but what about the other 3 cases? The code for these will not be included here but they follow the general structure as shown in *listing 5* except the undefined values are traded with the defined `%x` and `%y` where appropriate. Now that we have seen how this particular problem gets encoded in Z3 from several different angles, we can finally find out why this transformation is not a refinement. It turns out that the problem is not at all with undefined behavior but rather with plain old defined values themselves! We can run the query provided in *listing 3* and get the following output.

```
sat
(
  (define-fun %x () (_ BitVec 8)
    #x00)
  (define-fun %y () (_ BitVec 8)
    #x00)
)
```

If we add to the query the assertion `(assert (not (= %y #x00)))` the query then becomes unsatisfiable. Why does a values of `%y = 0` cause this query to break and why was it not found by the LLVM test suite? Looking back on *listing 2* at lines $4, 10$ a *signed* comparison is taking place. This means that if `%y` doesn't shift our mask to the right at all we are always

comparing `%x` to -127. This particular bug slipped through the LLVM test suite as well as the initial pass of Alive[3] because the value of `%y` was restricted to powers of two, which 0 is not.

This exploration of Alive2 and Z3, though small, shows the power of automated solvers. Simple bugs can remain uncovered in large systems for years and even testing suites prove to fall short of ensuring bug-free code. Compilers make a great target for using formal methods and verification tools because their specification is relatively simple and correctness is truly crucial. For additional notes and code produced during this small project visit the GitHub page.

References

[1] Juneyoung Lee. *Undef and Poison: Present and Future*. 2020. URL: `https://llvm.org/devmtg/2020-09/slides/Lee-UndefPoison.pdf` (visited on 03/01/2021).

[2] LLVM. *LLVM Language Reference Manual*. 2021. URL: `https://llvm.org/docs/LangRef.html` (visited on 03/15/2021).

[3] Nuno P. Lopes. *Bug 39861*. 2018. URL: `https://bugs.llvm.org/show_bug.cgi?id=39861#c0` (visited on 03/15/2021).

[4] Nuno P. Lopes. *Verifying Optimizations using SMT Solvers*. 2013. URL: `https://llvm.org/devmtg/2013-11/slides/Lopes-SMT.pdf`.

[5] John Regehr and Nuno P. Lopes. *Alive2 Part 1: Introduction*. 2020. URL: `https://blog.regehr.org/archives/1722` (visited on 04/27/2021).

[6] Microsoft Research. *Z3 - Guide*. 2021. URL: `https://rise4fun.com/z3/tutorialcontent/guide#h28` (visited on 03/01/2021).

[7] Wikipedia. *LLVM*. 2021. URL: `https://en.wikipedia.org/wiki/LLVM` (visited on 04/27/2021).

[3] An example of this pass is found at `https://rise4fun.com/Alive/I3O`