Testing debug info of optimised programs

Preliminary / work in progress

KLEE 2022

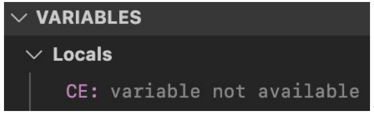
J. Ryan Stinnett
🏠 convolv.es
💼 King's College London

Stephen Kell
🏠 humprog.org
💼 King's College London

User experience

If you’ve tried debugging optimised programs before, you’ve probably seen these infamous messages…

```
(gdb) print expr
$1 = <optimized out>
```



User experience

If you’ve tried debugging optimised programs before, you’ve probably seen these infamous messages…

```
(gdb) print expr
$1 = <optimized out>
```

```
∨ VARIABLES
  ∨ Locals
      CE: variable not available
```

…which may trigger strong emotions such as…

😱 😓 😭 🤬

How it all goes wrong

Let's try compiling a small example…

```c
int example(int n) {
  int x = n * 2;
  int y = 0;
  for (unsigned int i = 0; i < n; i++) {
    y += x + 4 + n;
  }
  return y;
}
```

Clang 13 (O1)

```asm
_example:
  push    rbp
  mov     rbp, rsp
  test    edi, edi
  je      LBB0_1
  mov     eax, edi
  add     eax, -1
  lea     ecx, [rdi + 2*rdi]
  add     ecx, 4
  imul    ecx, eax
  lea     eax, [rdi + 2*rdi]
  add     eax, ecx
  add     eax, 4
  pop     rbp
  ret
LBB0_1:
  xor     eax, eax
  pop     rbp
  ret
```

How it all goes wrong

Let’s try compiling a small example…

```c
int example(int n) {
  int x = n * 2;
  int y = 0;
  for (unsigned int i = 0; i < n; i++) {
    y += x + 4 + n;
  }
  return y;
}
```

Clang 13 (O1)

6 / 17 instructions mapped to wrong source lines

```
_example:
  push   rbp
  mov    rbp, rsp
  test   edi, edi
  je     LBB0_1
  mov    eax, edi
  add    eax, -1
  lea    ecx, [rdi + 2*rdi]
  add    ecx, 4
  imul   ecx, eax
  lea    eax, [rdi + 2*rdi]
  add    eax, ecx
  add    eax, 4
  pop    rbp
  ret
LBB0_1:
  xor    eax, eax
  pop    rbp
  ret
```

How it all goes wrong

Let's try compiling a small example…

```c
int example(int n) {
  int x = n * 2;
  int y = 0;
  for (unsigned int i = 0; i < n; i++) {
    y += x + 4 + n;
  }
  return y;
}
```

Clang 13 (O1)

```asm
_example:
  push    rbp
  mov     rbp, rsp
  test    edi, edi
  je      LBB0_1
  mov     eax, edi
  add     eax, -1
  lea     ecx, [rdi + 2*rdi]
  add     ecx, 4
  imul    ecx, eax
  lea     eax, [rdi + 2*rdi]
  add     eax, ecx
  add     eax, 4
  pop     rbp
  ret
LBB0_1:
  xor     eax, eax
  pop     rbp
  ret
```

6 / 17 instructions mapped to wrong source lines

Variable y not available for 4 / 17 instructions

Lost in the pipes

- Optimisations today often corrupt or drop debug info
- Testing debug info is often manual, has poor coverage
  - SN Systems, LLVM contributors. Dexter. 2019.
- Recent work brings some automation, but uses imprecise value checking
  - Li et al. Debug information validation for optimized code. PLDI 2020.
  - Di Luna et al. Who's debugging the debuggers? Exposing debug information bugs in optimized binaries. ASPLOS 2021.
- Stronger testing would help spot more debug info handling bugs
  - Should lead to more reliable debugger experience overall

KLEE to the rescue!

- Perhaps KLEE can help us check debug info **more systematically**…
- KLEE explores paths through LLVM IR automatically
- Symbolic IR values are evaluated during KLEE's program execution
- LLVM IR supports debug info mappings from IR to source values

```
define i32 @example(i32 %n)
  %mul = shl i32 %n, 1, l2 c13
    ① KLEE tracks %mul as the symbolic value (Shl n 1) (simplified KQuery syntax)
  @dbg.value(i32 %mul, "x" l2)
    ② Debug info mapping states that source var x = IR value %mul
    ③ From ① and ②, it follows that x = symbolic value (Shl n 1)
```

Debug info example in abbreviated LLVM IR

Variable locations in DWARF

DWARF debug info generated by compiler (which we want to test) describes source variables via Turing-powerful stack machine with registers and memory as inputs

```
DW_TAG_variable
  DW_AT_name          ("y")
  DW_AT_decl_line     (3)
  DW_AT_type          (0x000000d5 "int")
  DW_AT_location
    [0x3f74, 0x3f7d):
      DW_OP_fbreg -12
    [0x3f7d, 0x3f90):
      <no location emitted>
    [0x3f90, 0x3f94):
      DW_OP_breg5 RDI+0, DW_OP_constu 0xffffffff, DW_OP_and,
      DW_OP_lit1, DW_OP_shl, DW_OP_stack_value
```

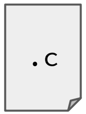
Simulated output illustrating expressivity of DWARF locations

Similar stack machine value expressions also appear in LLVM IR debug mappings

Locations are like a **symbolic mapping** of source variables to storage… Using KLEE's **symbolic values** from execution, we can ask an SMT solver to check the values in these locations!

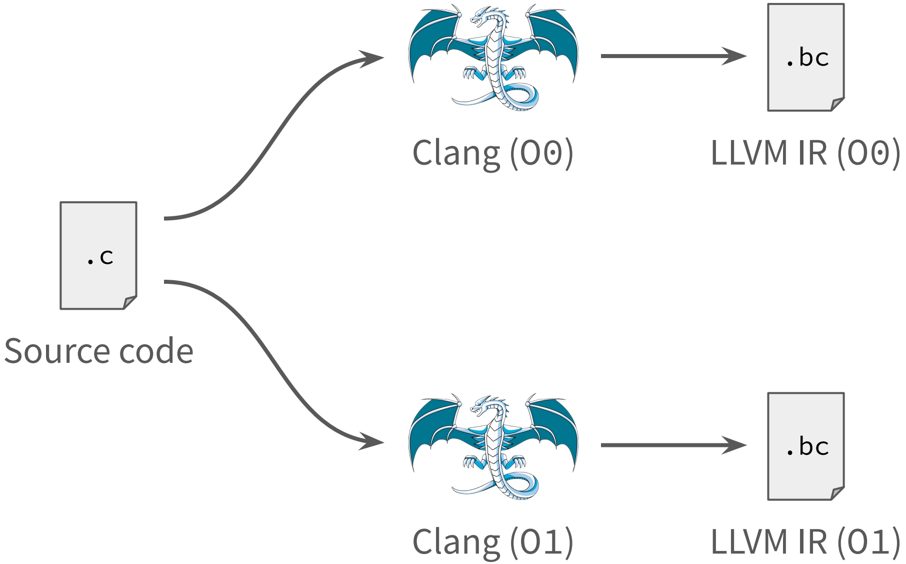
Approach

Debug info consistency check

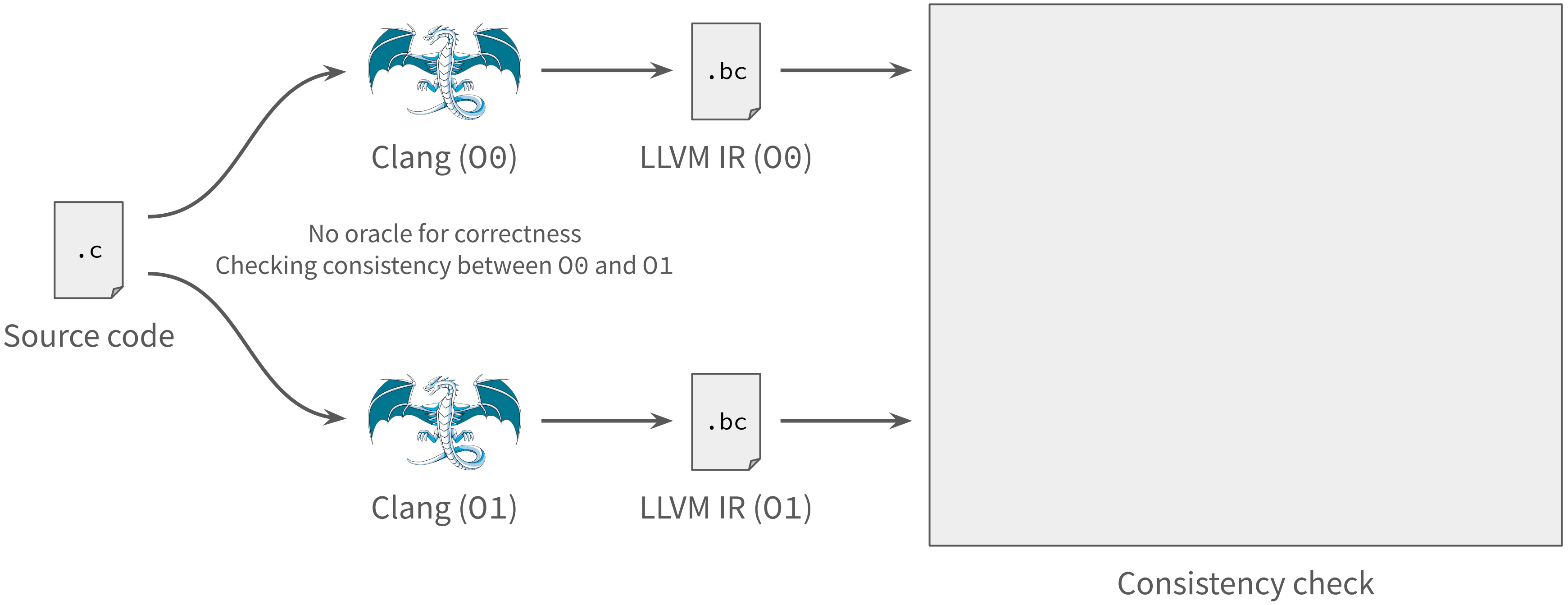


Source code

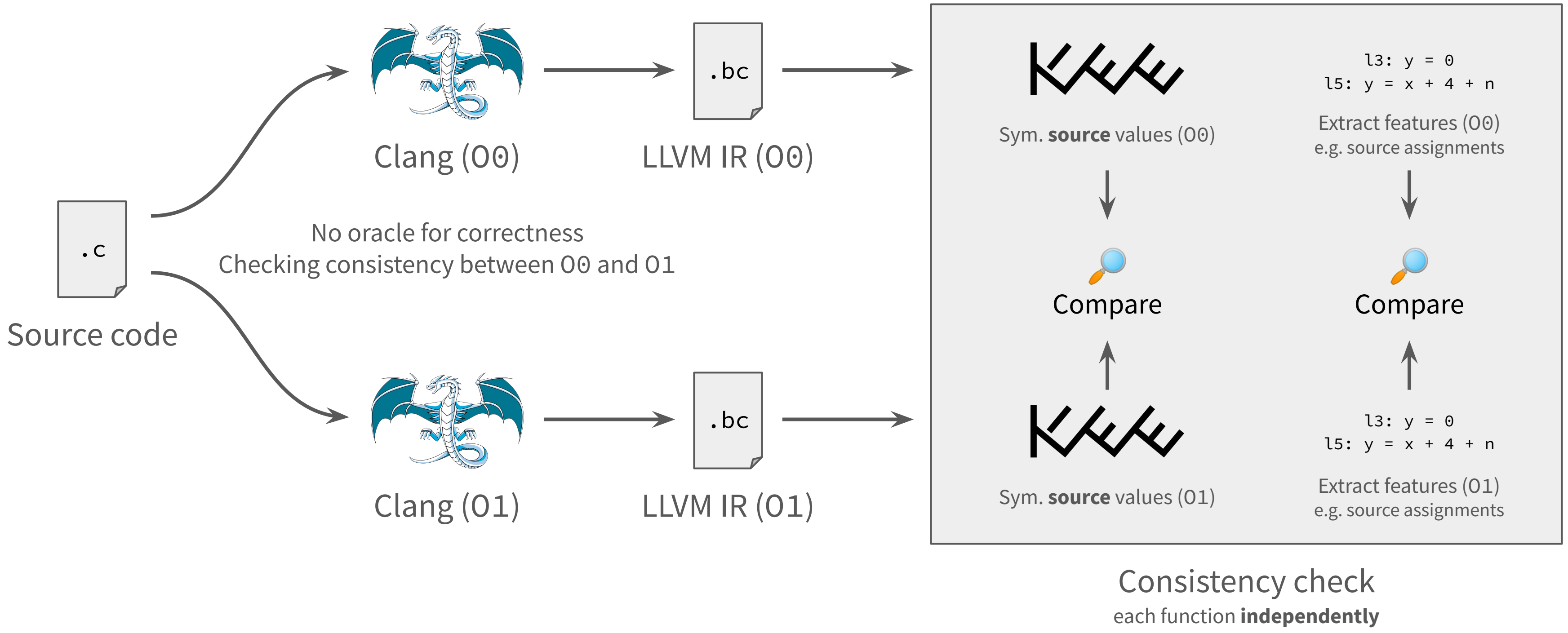
Debug info consistency check



Debug info consistency check



Debug info consistency check



Unusual application of symbolic execution

- Each function explored independently
- Each basic block visited at least once (similar to a compiler)
- Sufficient to gather generalised symbolic values for each source variable assignment from both target programs

Very different needs from
most applications of symbolic execution!

Examples

Consistency check examples

- Example 1: inconsistency found
  - Unoptimised LLVM IR (`O0`)
  - Optimised LLVM IR (`O1`)

Example 1: inconsistency found

```
1 int example(int n) {
2   int x = n * 2;
3   int y = 0;
4   for (unsigned int i = 0; i < n; i++) {
5     y += x + 4 + n;
6   }
7   return y;
8 }
```

Source code

```
define i32 @example(i32 %n) {
entry:
  %y = alloca i32 ① allocates stack space, %y points to this storage
  @dbg.declare(i32* %y, "y" l3) ② source var y is stored at %y
  store i32 0, i32* %y, l3
    ③ stores constant (0) for source var y

for.body:
  %3 = load i32, i32* %x, l5
  %add = add i32 %3, 4, l5
  %4 = load i32, i32* %n.addr, l5
  %add1 = add i32 %add, %4, l5
  %5 = load i32, i32* %y, l5
  %add2 = add i32 %5, %add1, l5
  store i32 %add2, i32* %y, l5
    ④ stores %add2 for source var y
```

Unoptimised LLVM IR (O0)

Example 1: inconsistency found

```
1 int example(int n) {
2   int x = n * 2;
3   int y = 0;
4   for (unsigned int i = 0; i < n; i++) {
5     y += x + 4 + n;
6   }
7   return y;
8 }
```

Source code

```
define i32 @example(i32 %n) {
entry:
  %y = alloca i32 ① allocates stack space, %y points to this storage
  @dbg.declare(i32* %y, "y" l3) ② source var y is stored at %y
  store i32 0, i32* %y, l3
    ③ stores constant (0) for source var y

for.body:
  %3 = load i32, i32* %x, l5
  %add = add i32 %3, 4, l5
  %4 = load i32, i32* %n.addr, l5
  %add1 = add i32 %add, %4, l5
  %5 = load i32, i32* %y, l5
  %add2 = add i32 %5, %add1, l5
  store i32 %add2, i32* %y, l5
    ④ stores %add2 for source var y
```

At source line 3:
`y = 0`

At source line 5:
`y = (Add 4 (Add (Mul 2 n) n))`

Unoptimised LLVM IR (O0)

Example 1: inconsistency found

```c
1 int example(int n) {
2   int x = n * 2;
3   int y = 0;
4   for (unsigned int i = 0; i < n; i++) {
5     y += x + 4 + n;
6   }
7   return y;
8 }
```

Source code

```llvm
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
```

① source var y = constant (0)

```llvm
for.cond.cleanup.loopexit:
  %0 = add i32 %n, -1, l4
  %add = add i32 %n, 4
  %mul = shl i32 %n, 1, l2
  %add1 = add i32 %add, %mul
  %1 = mul i32 %0, %add1, l4
  %2 = mul i32 %n, 3, l4
  %3 = add i32 %1, %2, l4
  %4 = add i32 %3, 4, l4
```

② should be mapped to y, but debug mapping lost!

```llvm
  @dbg.value(i32 undef, "y" l3)
```

③ dead debug mapping without an input value

Optimised LLVM IR (O1)

Example 1: inconsistency found

```
1 int example(int n) {
2   int x = n * 2;
3   int y = 0;
4   for (unsigned int i = 0; i < n; i++) {
5     y += x + 4 + n;
6   }
7   return y;
8 }
```

Source code

```
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
```

① source var y = constant (0)

```
for.cond.cleanup.loopexit:
  %0 = add i32 %n, -1, l4
  %add = add i32 %n, 4
  %mul = shl i32 %n, 1, l2
  %add1 = add i32 %add, %mul
  %1 = mul i32 %0, %add1, l4
  %2 = mul i32 %n, 3, l4
  %3 = add i32 %1, %2, l4
  %4 = add i32 %3, 4, l4
```

② should be mapped to y, but debug mapping lost!

```
  @dbg.value(i32 undef, "y" l3)
```

③ dead debug mapping without an input value

At source line 3:
`y = 0`

Value mapping lost, should be:

```
y = %4 = (Add 4
 (Add
  (Mul (Add -1 n)
   (Add 4
    (Add n (Shl n 1))))
  (Mul 3 n)))
```

Optimised LLVM IR (O1)

Example 1: inconsistency found

```
define i32 @example(i32 %n) {
entry:
  %y = alloca i32 ① allocates stack space, %y points to this storage
  @dbg.declare(i32* %y, "y" l3) ② source var y is stored at %y
  store i32 0, i32* %y, l3
    ③ stores constant (0) for source var y

for.body:
  %3 = load i32, i32* %x, l5
  %add = add i32 %3, 4, l5
  %4 = load i32, i32* %n.addr, l5
  %add1 = add i32 %add, %4, l5
  %5 = load i32, i32* %y, l5
  %add2 = add i32 %5, %add1, l5
  store i32 %add2, i32* %y, l5
    ④ stores %add2 for source var y
```

At source line 3:
`y = 0`

At source line 5:
`y = (Add 4 (Add (Mul 2 n) n))`

Unoptimised LLVM IR (O0)

```
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
    ① source var y = constant (0)
for.cond.cleanup.loopexit:
  %0 = add i32 %n, -1, l4
  %add = add i32 %n, 4
  %mul = shl i32 %n, 1, l2
  %add1 = add i32 %add, %mul
  %1 = mul i32 %0, %add1, l4
  %2 = mul i32 %n, 3, l4
  %3 = add i32 %1, %2, l4
  %4 = add i32 %3, 4, l4
    ② should be mapped to y, but debug mapping lost!
  @dbg.value(i32 undef, "y" l3)
    ③ dead debug mapping without an input value
```

At source line 3:
`y = 0`

Value mapping lost, should be:

```
y = %4 = (Add 4
 (Add
  (Mul (Add -1 n)
   (Add 4
    (Add n (Shl n 1))))
  (Mul 3 n)))
```

Optimised LLVM IR (O1)

Example 1: inconsistency found

```
define i32 @example(i32 %n) {
entry:
  %y = alloca i32 ① allocates stack space, %y points to this storage
  @dbg.declare(i32* %y, "y" l3) ② source var y is stored at %y
  store i32 0, i32* %y, l3
    ③ stores constant (0) for source var y

for.body:
  %3 = load i32, i32* %x, l5
  %add = add i32 %3, 4, l5
  %4 = load i32, i32* %n.addr, l5
  %add1 = add i32 %add, %4, l5
  %5 = load i32, i32* %y, l5
  %add2 = add i32 %5, %add1, l5
  store i32 %add2, i32* %y, l5
    ④ stores %add2 for source var y
```

At source line 3:
`y = 0`

At source line 5:
`y = (Add 4 (Add (Mul 2 n) n))`

Unoptimised LLVM IR (O0)

```
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
    ① source var y = constant (0)
for.cond.cleanup.loopexit:
  %0 = add i32 %n, -1, l4
  %add = add i32 %n, 4
  %mul = shl i32 %n, 1, l2
  %add1 = add i32 %add, %mul
  %1 = mul i32 %0, %add1, l4
  %2 = mul i32 %n, 3, l4
  %3 = add i32 %1, %2, l4
  %4 = add i32 %3, 4, l4
    ② should be mapped to y, but debug mapping lost!
  @dbg.value(i32 undef, "y" l3)
    ③ dead debug mapping without an input value
```

At source line 3:
`y = 0`

Value mapping lost, should be:

```
y = %4 = (Add 4
 (Add
  (Mul (Add -1 n)
   (Add 4
    (Add n (Shl n 1))))
  (Mul 3 n)))
```

Optimised LLVM IR (O1)

Assignments: wrong source line
Values: mapping lost
Inconsistency found!

Status

Current status

- Core approach implemented in new tool built on top of KLEE
- Expects two LLVM modules (`*.bc` or `*.ll`), one before and one after optimisation

```
$ debug-info-check example-00.ll example-01.ll
```

- Produces consistency report for first function found

```
## Variables

❌ After variable intrinsic with undef input, asm line 30
  @dbg.value(i32 undef, !18)
❌ After variable intrinsic with undef input, asm line 31
  @dbg.value(i32 undef, !17)
❌ After variable intrinsic with undef input, asm line 32
  @dbg.value(i32 undef, !17, !DIExpression(DW_OP_LLVM_arg, 0, DW_OP_LLVM_arg,
1, DW_OP_plus, DW_OP_stack_value))
❌ After variable intrinsic with undef input, asm line 33
  @dbg.value(i32 undef, !18, !DIExpression(DW_OP_plus_uconst, 1,
DW_OP_stack_value))
✅ 4 before variables found, 4 after variables found, 0 mismatched

## Assignments

❌ 6 before assignments found, 5 after assignments found, 3 mismatched
❌ Mismatched before `i` on src line 4 from store i32 %inc, i32* %i, l4
❌ Mismatched before `y` on src line 5 from store i32 %add2, i32* %y, l5
❌ Mismatched after `x` on src line 2 from %mul = shl i32 %n, 1, l2

🔔 Some assignment checks failed, value checks may be nonsensical…
```

Next steps

- Add support for more complex debug mapping cases
- Add function independent mode to KLEE to support consistency check with multi-function code samples
- File compiler bugs found
  - Several already found via simple test cases during tool implementation
  - Expecting many more to be revealed via randomised test generation
- Gather debuggability stats by optimisation pass
- Expand coverage to include machine code gen phase
  - Perhaps by lifting binaries back up to LLVM IR…?

Summary

- Debug info often gets lost during optimisation
- Current testing approaches…
  - …are often manual
  - …use imprecise value checks
- Comparing symbolic values gathered by KLEE enables automated consistency checks of debug value correctness
  - Relies on (possibly surprising) connection between debug location mappings and KLEE's symbolic values during execution

Thanks!

J. Ryan Stinnett
🏠 convolv.es
💼 King's College London

Stephen Kell
🏠 humprog.org
💼 King's College London

Workflows can do better

- Poor developer experience has trained many programmers to assume optimised debugging is somehow insurmountable
  - Some may avoid using debuggers entirely
  - In some cases, you can rebuild without optimisation and try debugging again…
- Real scenarios for optimised debugging
  - Core dumps collected in production
  - Resource heavy programs (e.g. video games) which are too slow without optimisation
  - Programs whose behavior depends on optimisation (e.g. Linux kernel)
  - Tracing unwanted behaviours (e.g. race conditions, memory errors) which may only occur with optimisation
  - **Any program … if you want to debug what actually ran!**

Priority of debug info for compiler authors

- Passes do try to preserve debug info…
  - e.g. LLVM's How to update debug info guide for optimisation pass authors
- Incentives not aligned for correct and complete debug info
  - Extra work to produce debug info on top of fast, correct run-time code
- No standard metrics for comparing debug info quality

Example 2: consistent

```c
1 int example(int n) {
2   int x = n * 2;
3   int y = 0;
4   for (unsigned int i = 0; i < n; i++) {
5     y += x + 4 + n;
6   }
7   return y;
8 }
```

Source code

```llvm
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
     ① source var y = constant (0)
  %mul = shl i32 %n, 1, l2
  %add = add i32 %n, 4
  %add1 = add i32 %add, %mul

for.body:
  %y.011 = phi i32 [%add2, %for.body], [0, %entry]
  @dbg.value(i32 %y.011, "y" l3)
     ② source var y = %add2 or 0
  %add2 = add i32 %add1, %y.011, l5
  @dbg.value(i32 %add2, "y" l3)
     ③ source var y = %add2
```

Partially optimised LLVM IR (O1)

Example 2: consistent

```
1 int example(int n) {
2   int x = n * 2;
3   int y = 0;
4   for (unsigned int i = 0; i < n; i++) {
5     y += x + 4 + n;
6   }
7   return y;
8 }
```

Source code

```
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
```
① source var y = constant (0)
```
  %mul = shl i32 %n, 1, l2
  %add = add i32 %n, 4
  %add1 = add i32 %add, %mul

for.body:
  %y.011 = phi i32 [%add2, %for.body], [0, %entry]
  @dbg.value(i32 %y.011, "y" l3)
```
② source var y = %add2 or 0
```
  %add2 = add i32 %add1, %y.011, l5
  @dbg.value(i32 %add2, "y" l3)
```
③ source var y = %add2

At source line 3:
`y = 0`

At source line 5:
`y = (Add 4 (Add n (Mul 2 n)))`

Partially optimised LLVM IR (O1)

Example 2: consistent

```
define i32 @example(i32 %n) {
entry:
  %y = alloca i32 ① allocates stack space, %y points to this storage
  @dbg.declare(i32* %y, "y" l3) ② source var y is stored at %y
  store i32 0, i32* %y, l3
    ③ stores constant (0) for source var y

for.body:
  %3 = load i32, i32* %x, l5
  %add = add i32 %3, 4, l5
  %4 = load i32, i32* %n.addr, l5
  %add1 = add i32 %add, %4, l5
  %5 = load i32, i32* %y, l5
  %add2 = add i32 %5, %add1, l5
  store i32 %add2, i32* %y, l5
    ④ stores %add2 for source var y
```

At source line 3:
y = 0

At source line 5:
y = (Add 4 (Add (Mul 2 n) n))

Unoptimised LLVM IR (O0)

```
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
    ① source var y = constant (0)
  %mul = shl i32 %n, 1, l2
  %add = add i32 %n, 4
  %add1 = add i32 %add, %mul

for.body:
  %y.011 = phi i32 [%add2, %for.body], [0, %entry]
  @dbg.value(i32 %y.011, "y" l3)
    ② source var y = %add2 or 0
  %add2 = add i32 %add1, %y.011, l5
  @dbg.value(i32 %add2, "y" l3)
    ③ source var y = %add2
```

At source line 3:
y = 0

At source line 5:
y = (Add 4 (Add n (Mul 2 n)))

Partially optimised LLVM IR (O1)

Example 2: consistent

```
define i32 @example(i32 %n) {
entry:
  %y = alloca i32 ① allocates stack space, %y points to this storage
  @dbg.declare(i32* %y, "y" l3) ② source var y is stored at %y
  store i32 0, i32* %y, l3
    ③ stores constant (0) for source var y

for.body:
  %3 = load i32, i32* %x, l5
  %add = add i32 %3, 4, l5
  %4 = load i32, i32* %n.addr, l5
  %add1 = add i32 %add, %4, l5
  %5 = load i32, i32* %y, l5
  %add2 = add i32 %5, %add1, l5
  store i32 %add2, i32* %y, l5
    ④ stores %add2 for source var y
```

At source line 3:
y = 0

At source line 5:
y = (Add 4 (Add (Mul 2 n) n))

Unoptimised LLVM IR (O0)

```
define i32 @example(i32 %n) {
entry:
  @dbg.value(i32 0, "y" l3)
    ① source var y = constant (0)
  %mul = shl i32 %n, 1, l2
  %add = add i32 %n, 4
  %add1 = add i32 %add, %mul

for.body:
  %y.011 = phi i32 [%add2, %for.body], [0, %entry]
  @dbg.value(i32 %y.011, "y" l3)
    ② source var y = %add2 or 0
  %add2 = add i32 %add1, %y.011, l5
  @dbg.value(i32 %add2, "y" l3)
    ③ source var y = %add2
```

At source line 3:
y = 0

At source line 5:
y = (Add 4 (Add n (Mul 2 n)))

Partially optimised LLVM IR (O1)

Assignments: consistent
Values: consistent
Debug info check passed! 🎉

Consistency check examples

- Example 1: inconsistency found
  - Unoptimised LLVM IR (`O0`)
  - Optimised LLVM IR (`O1`)
  - Assignments: wrong source line
  - Values: mapping lost
- Example 2: consistent
  - Unoptimised LLVM IR (`O0`)
  - Partially optimised LLVM IR (`O1`, stopping early)
    - Optimisation stopped just before induction variable simplification
  - Assignments: consistent
  - Values: consistent