technology
from seed

Verifying Optimizations using SMT Solvers

Nuno Lopes

TÉCNICO LISBOA

Why verify optimizations?

- Catch bugs before they even exist
- Corner cases are hard to debug
- Time spent in additional verification step pays off
- Technology available today, with more to follow

Not a replacement for testing

“Beware of bugs in the above code; I have only proved it correct, not tried it”

Donald Knuth, 1977

Outline

- SAT/SMT Solvers
- InstCombine
- Assembly
- ConstantRange
- Future developments

Outline

- SAT/SMT Solvers
- InstCombine
- Assembly
- ConstantRange
- Future developments

SAT Solvers

- A SAT solver takes a Boolean formula as input:
  - $(a \vee b \vee c) \wedge (\neg b \vee c)$
- And returns:
  - SAT, if the formula is satisfiable
  - UNSAT, if the formula is unsatisfiable
- If SAT, we also get a model:
  - $a = \text{true},\ b = \text{false},\ c = \text{false}$

SMT Solvers

- Generalization of SAT solvers
- Variables can take other domains:
  - Booleans
  - Bit-vectors
  - Reals (linear / non-linear)
  - Integers (linear / non-linear)
  - Arrays
  - Data types
  - Floating point
  - Uninterpreted functions (UFs)
  - …

Available SMT Solvers

- Boolector
- CVC4
- MathSAT 5
- STP
- Z3 (http://rise4fun.com/Z3/)

Bit-Vector Theory

- Operations between bit-vector variables:
  - Add/Sub/Mul/Div/Rem
  - Shift and rotate
  - Zero/sign extend
  - Bitwise And/or/neg/not/nand/xor/…
  - Comparison: ge/le/…
  - Concat and extract
- Includes sign/unsigned variants
- Variables of **fixed** bit width

Bit-vector theory: example

- Let's prove that the following are equivalent:
  - $(x - 1)\&x = 0$
  - $x\&(-x) = x$

- Thinking SMT:
  - "Both formulas give the same result for all x"
  - "There isn't a value for x such that the result of the formulas differs"

Example in SMT-LIB 2


```
(declare-fun x () (_ BitVec 32))

(assert (not (=
  ; x&(x-1) == 0
  (= (bvand x (bvsub x #x00000001)) #x00000000)

  ; x&(-x) == x
  (= (bvand x (bvneg x)) x))
)))

(check-sat)
```

```
> unsat
```

http://rise4fun.com/Z3/2YFz

Example: really testing for power of 2?


```
(declare-fun x () (_ BitVec 4))

(assert (not (=
  ; x&(x-1) == 0
  (= (bvand x (bvsub x #x1)) #x0)

  ; x == 1 or x == 2 or x == 4 or x == 8
  (or (= x #x1) (= x #x2) (= x #x4) (= x #x8))
)))

(check-sat)
(get-model)
```

```
> sat
> (model
    (define-fun x () (_ BitVec 4)
      #x0))
```

http://rise4fun.com/Z3/qGl2

Outline

- SAT/SMT Solvers
- InstCombine
- Assembly
- ConstantRange
- Future developments

InstCombine

- Optimizes sequences of instructions
- Perfect target for verification with SMT solvers

InstCombine Example


```
; (A ^ -1) & (1 << B) != 0
%neg = xor i32 %A, -1
%shl = shl i32 1, %B
%and = and i32 %neg, %shl
%cmp = icmp ne i32 %and, 0
```

⇒

```
; (1 << B) & A == 0
%shl = shl i32 1, %B
%and = and i32 %shl, %A
%cmp = icmp eq i32 %and, 0
```

InstCombine Example


```
(declare-fun A () (_ BitVec 32))
(declare-fun B () (_ BitVec 32))

(assert (not (=
  ; (1 << B) & (A ^ -1) != 0
  (not (= (bvand (bvshl #x00000001 B)
                 (b
  ; (1 << B) & A ==
  (= (bvand (bvshl
)))

(check-sat)
```

```
> sat
> (model
    (define-fun A () (_ BitVec 32)
      #x00000000)
    (define-fun B () (_ BitVec 32)
      #x00020007)
  )
```

http://rise4fun.com/Z3/OmRP

InstCombine Example


```
(declare-fun A () (_ BitVec 32))
(declare-fun B () (_ BitVec 32))

(assert (bvule B #x0000001F))
(assert (not (=
  ; (1 << B) & (A ^ -1) != 0
  (not (= (bvand (bvshl #x00000001 B)
                 (bvxor A #xffffffff)) #x00000000))

  ; (1 << B) & A == 0
  (= (bvand (bvshl #x00000001 B) A) #x00000000)
)))

(check-sat)
```

> unsat

http://rise4fun.com/Z3/pj2B

Outline

- SAT/SMT Solvers
- InstCombine
- Assembly
- ConstantRange
- Future developments

IR to Assembly

- PR16426: poor code for multiple __builtin_*_overflow()

```
// returns x * y + z
// 17 instructions on X86
unsigned foo(unsigned x, unsigned y, unsigned z) {
  unsigned res;
  if (__builtin_umul_overflow(x, y, &res) |
      __builtin_uadd_overflow(res, z, &res)) {
    return 0;
  }
  return res;
}
```

PR16426: IR


```
define i32 foo(i32 %x, i32 %y, i32 %z) {
entry:
  %0 = call { i32, i1 } @llvm.umul.with.overflow.i32(i32 %x, i32 %y)
  %1 = extractvalue { i32, i1 } %0, 1
  %2 = extractvalue { i32, i1 } %0, 0
  %3 = call { i32, i1 } @llvm.uadd.with.overflow.i32(i32 %2, i32 %z)
  %4 = extractvalue { i32, i1 } %3, 1
  %or3 = or i1 %1, %4
  br i1 %or3, label %return, label %if.end

if.end:
  %5 = extractvalue { i32, i1 } %3, 0
  br label %return

return:
  %retval.0 = phi i32 [ %5, %if.end ], [ 0, %entry ]
  ret i32 %retval.0
}
```

PR16426: Current X86 Assembly (17 instructions)


```
foo:
# BB#0:                                 # %entry
        pushl   %esi
        movl    8(%esp), %eax
        mull    12(%esp)
        pushfl
        popl    %esi
        addl    16(%esp), %eax
        setb    %dl
        xorl    %ecx, %ecx
        pushl   %esi
        popfl
        jo      .LBB0_3
# BB#1:                                 # %entry
        testb   %dl, %dl
        jne     .LBB0_3
# BB#2:                                 # %if.end
        movl    %eax, %ecx
.LBB0_3:                                # %return
        movl    %ecx, %eax
        popl    %esi
        ret
```

PR16426: Proposed X86 Assembly (8 instructions)


```
        movl    8(%esp), %eax
        mull    12(%esp)
        addl    16(%esp), %eax
        adcl    %edx, %edx
        jne     .LBB0_1
.LBB0_2:
        # result already in EAX
        ret
.LBB0_1:
        xorl    %eax, %eax
        jmp     .LBB0_2
```

PR16426: Michael says my proposal has a bug


```
        movl    8(%esp), %eax
        mull    12(%esp)
        addl    16(%esp), %eax
        adcl    0, %edx
        jne     .LBB0_1
.LBB0_2:
        # result already in EAX
        ret
.LBB0_1:
        xorl    %eax, %eax
        jmp     .LBB0_2
```

PR16426: Asm in SMT


```
; movl    8(%esp), %eax
; mull    12(%esp)
(assert (let ((mul (bvmul ((_ zero_extend 32) x)
                          ((_ zero_extend 32) y))))
  (and
    (= EAX ((_ extract 31 0) mul))
    (= EDX ((_ extract 63 32) mul))
  )))
```

PR16426: Asm in SMT


```
; addl    16(%esp), %eax
(assert (and
   (= EAX2 (bvadd EAX z))
   (= CF ((_ extract 32 32)
          (bvadd ((_ zero_extend 1) EAX)
                 ((_ zero_extend 1) z))))
))
```

PR16426: Asm in SMT


```
; adcl    %edx, %edx
(assert (and
   (= EDX2 (bvadd EDX EDX ((_ zero_extend 31) CF)))
   (= ZF (= EDX2 #x00000000))
))
```

PR16426: Asm in SMT


```
        jne     .LBB0_1  # Jump if ZF=0
.LBB0_2:
        ret
.LBB0_1:
        xorl    %eax, %eax
        jmp     .LBB0_2
```

```
(assert (= asm_result
  (ite ZF EAX2 #x00000000)
))
```

PR16426: IR in SMT


```
(assert (= llvm_result
  (let ((overflow
    (or (bvugt
          (bvmul ((_ zero_extend 32) x)
                 ((_ zero_extend 32) y))
          #x00000000FFFFFFFF)
        (bvugt
          (bvadd ((_ zero_extend 4) (bvmul x y))
                 ((_ zero_extend 4) z))
          #x0FFFFFFFF))))
    (ite overflow #x00000000 (bvadd (bvmul x y) z)))
))
```

PR16426: Correctness


```
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(declare-fun z () (_ BitVec 32))

(assert (not (=
  asm_result
  llvm_result
)))

(check-sat)
(get-model)
```

```
> sat
> (model
   (define-fun z () (_ BitVec 32)
     #x15234d22)
   (define-fun y () (_ BitVec 32)
     #x84400100)
   (define-fun x () (_ BitVec 32)
     #xf7c5ebbe)
  )
```

http://rise4fun.com/Z3/VIxt

Outline

- SAT/SMT Solvers
- InstCombine
- Assembly
- ConstantRange
- Future developments

ConstantRange

- Data-structure that represents ranges of integers with overflow semantics (i.e., bit-vectors)
  - [0,5) – from 0 to 4
  - [5,2) – from 5 to INT_MAX or from 0 to 1
- Used by Lazy Value Info (LVI), and Correlated Value Propagation (CVP)
- Several bugs in the past (correctness and optimality)

ConstantRange::signExtend()

- 8 lines of C++
- Is it correct?

```
442 /// signExtend - Return a new range in the specified integer type, which must
443 /// be strictly larger than the current type.  The returned range will
444 /// correspond to the possible range of values as if the source range had been
445 /// sign extended.
446 ConstantRange ConstantRange::signExtend(uint32_t DstTySize) const {
447   if (isEmptySet()) return ConstantRange(DstTySize, /*isFullSet=*/false);
448
449   unsigned SrcTySize = getBitWidth();
450   assert(SrcTySize < DstTySize && "Not a value extension");
451   if (isFullSet() || isSignWrappedSet()) {
452     return ConstantRange(APInt::getHighBitsSet(DstTySize,DstTySize-SrcTySize+1),
453                          APInt::getLowBitsSet(DstTySize, SrcTySize-1) + 1);
454   }
455
456   return ConstantRange(Lower.sext(DstTySize), Upper.sext(DstTySize));
457 }
```

Auxiliary definitions in SMT


```
(define-sort Integer () (_ BitVec 32))
(define-sort Interval () (_ BitVec 64))
(define-sort Interval2 () (_ BitVec 72))

(define-fun L ((I Interval)) Integer
  ((_ extract 63 32) I))

(define-fun H ((I Interval)) Integer
  ((_ extract 31 0) I))

(define-fun isFullSet ((I Interval)) Bool
  (and (= (L I) (H I)) (= (L I) #xFFFFFFFF)))
```

signExtend() in SMT


```
(define-fun signExtend ((I Interval)) Interval2
  (ite
    (isEmptySet I)
    EmptySet
    (ite (or (isFullSet I) (isSignWrappedSet I))
      (getInterval #xF80000000
                   (bvadd #x07FFFFFF #x000000001))
      (getInterval ((_ sign_extend 4) (L I))
                   ((_ sign_extend 4) (H I))))
  )
)
```

```
446 ConstantRange ConstantRange::signExtend(uint32_t DstTySize) const {
447   if (isEmptySet()) return ConstantRange(DstTySize, /*isFullSet=*/false);
448
449   unsigned SrcTySize = getBitWidth();
450   assert(SrcTySize < DstTySize && "Not a value extension");
451   if (isFullSet() || isSignWrappedSet()) {
452     return ConstantRange(APInt::getHighBitsSet(DstTySize,DstTySize-SrcTySize+1),
453                          APInt::getLowBitsSet(DstTySize, SrcTySize-1) + 1);
454   }
455
456   return ConstantRange(Lower.sext(DstTySize), Upper.sext(DstTySize));
```

Correctness of signExtend()


```
(declare-fun n () Integer)
(declare-fun N () Interval)

(assert
  (and
    (contains n N)
    (not (contains2 ((_ sign_extend 4) n)
                    (signExtend N)))
  )
)
(check-sat)
```

```
> unsat
```

http://rise4fun.com/Z3/wLFX

Optimality of signExtend()

- It’s correct, cool, but..
- Does signExtend() always returns the tightest range?
- Or are we missing optimization opportunities?

Optimality of signExtend() in SMT


```
(declare-fun N () Interval)
(declare-fun R () Interval2)

(assert (bvult (getSetSize R)
               (getSetSize (signExtend N))))

(assert
  (forall ((n Intege
    (=> (contains n
      (contains2 ((_
  ))

(check-sat-using qfbv)
```

```
> sat
> (model
    (define-fun N () (_ BitVec 64)
      #x8000010080000000)
    (define-fun R () (_ BitVec 72)
      #xe010000004009e0d04)
  )
```

http://rise4fun.com/Z3/wLFX

Debugging with SMT models


```
(eval (L N))
(eval (H N))
(eval (L2 R))
(eval (H2 R))
(eval (L2 (signExtend N)))
(eval (H2 (signExtend N)))
```

```
#x80000100
#x80000000
#xe01000000
#x4009e0d04
#xf80000100
#xf80000000
```

http://rise4fun.com/Z3/wLFX

Optimality of signExtend() Fixed


```
--- llvm/trunk/lib/Support/ConstantRange.cpp      2013/10/28 16:52:38 193523
+++ llvm/trunk/lib/Support/ConstantRange.cpp      2013/10/31 19:53:53 193795
@@ -445,6 +445,11 @@

   unsigned SrcTySize = getBitWidth();
   assert(SrcTySize < DstTySize && "Not a value extension");
+
+  // special case: [X, INT_MIN) -- not really wrapping around
+  if (Upper.isMinSignedValue())
+    return ConstantRange(Lower.sext(DstTySize), Upper.zext(DstTySize));
+
   if (isFullSet() || isSignWrappedSet()) {
     return ConstantRange(APInt::getHighBitsSet(DstTySize,DstTySize-SrcTySize+1),
                          APInt::getLowBitsSet(DstTySize, SrcTySize-1) + 1);
```

http://rise4fun.com/Z3/4pl9s
http://rise4fun.com/Z3/OGAW

Outline

- SAT/SMT Solvers
- InstCombine
- Assembly
- ConstantRange
- Future developments

Future work

- Automatic translation from *.cpp to *.smt2
- Recursive functions in SMT (Horn clauses)
- Floating point in SMT (for OpenCL?)
- Verify more complex stuff (SCEV, …?)
- Termination checking: do InstCombine and LegalizeDAG (and other canonicalization passes) terminate for all inputs?

Conclusion

- Software verification technology (namely SMT solvers) is ready to verify **some** parts of compilers
- InstCombine, DAG Combiner, LegalizeDAG, etc can be verified today
- Ideal for answering “What if I do this change..?” questions

- Syntax of SMT-LIB 2:
  - Bit-vectors: http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2
  - Arrays: http://smtlib.cs.uiowa.edu/theories/ArraysEx.smt2
  - Floating point: https://groups.google.com/forum/#!forum/smt-fp
- Stack Overflow: #smt, #z3

Trip sponsored by:

technology
from seed

inesc id
lisboa