Term-Level Reasoning in Support of Bit-blasting

Sol Swords
Centaur Technology, Inc.

Introduction

The GL Bit-blasting Library

- Tool for proving ACL2 theorems via bit-level methods
- Supports BDD and AIG/SAT based reasoning
- Implemented as a verified clause processor*

*modulo unverified parts, e.g. external SAT solver

“Traditional” GL usage

```
(def-gl-thm fast-bitcount-implements-logcount
   :hyp (unsigned-byte-p 32 x)
   :concl (equal (my-fast-bitcount x)
                 (logcount x))
   :g-bindings `((x ,(gl::g-int 0 1 33))))
```

“Traditional” GL paradigm

Symbolically interpret `(my-function a b)`:

- Recursively derive a *symbolic value* for each subterm
- Dive into definitions of most functions
- Call special functions for “primitives”
  - E.g., `binary-+`, `logbitp`, `ash`, etc.

“Traditional” GL -- Symbolic Objects

Symbolic value derived for each subterm could be:

- Symbolic Boolean (BDD/AIG)
- Symbolic integer (List of BDD/AIG)
- Concrete value
- Cons of symbolic values
- If-then-else of symbolic values (case split)
- Function call (bad)
- Variable (bad)

“Traditional” GL problems

- Only encodes Booleans and integers using Boolean functions (AIGs/BDDs) → anything else causes case splits if not constant
- Theorem variables must be constrained to some *shape* by hypotheses and bound to a corresponding *shape-spec* in :g-bindings
- Limited higher-level reasoning
- Limited abstraction

Motivating Example 1

Suppose `xi` are Boolean variables...

```
(s :a x0 nil) →  (if x0 `((:a . ,x0)) nil)

(s :b x1 (s :a x0 nil)) →

     (if x1

         (if x0 `((:a . ,x0) (:b . ,x1))) `((:b . ,x1))

       (if x0 `((:a . ,x0)) nil)))
```

Motivating Example 1

Setting multiple keys to symbolic Boolean values in a record →

exponential object size!

All just because of an implementation detail.

Motivating Example 2

```
(def-gl-thm simple-tautology
  :hyp t
  :concl (iff (xor (xor a b) a) b)
  :g-bindings `((a ,(gl::g-boolean 0))
                (b ,(gl::g-boolean 1))))
```

Doesn’t work (why?)

Motivating Example 2

```
(def-gl-thm simple-tautology
  :hyp (and (booleanp a) (booleanp b))
  :concl (iff (xor (xor a b) a) b)
  :g-bindings `((a ,(gl::g-boolean 0))
                (b ,(gl::g-boolean 1))))
```

Have to assume types for variables even when this shouldn’t be necessary.

Term-level Extensions

*Fight Problems with Bigger Problems**

*Elliott Smith, “Baby Britain”

- Solution: make GL more like ACL2…
- Try to make sense of term-like symbolic “values” -- rewriter
- Let users disable functions
- Add more rule classes to deal with problems as they crop up.

Rewriting and disabling functions

```
(gl-set-uninterpreted g)
(gl-set-uninterpreted s)

(def-gl-rewrite g-of-s-for-gl
  (equal (g k1 (s k2 v r))
         (if (equal k1 k2) v (g k1 r))))

(def-gl-rewrite s-of-s-equal-keys
  (implies (equal k1 k2)
           (equal (s k1 v1 (s k2 v2 r))
                  (s k1 v1 r))))
```

Rewriting example

```
(def-gl-thm simple-record-thm
  :hyp (unsigned-byte-p 16 a)
  :concl (b* ((st (s :a a st))
              (st (my-dumb-machine '((add :a :a)
                                     (add :a :a))
                                   st)))
           (equal (g :a st) (* 4 a)))
  :g-bindings `((a ,(gl::g-int 0 1 17))
                (st ,(gl::g-var 'st))))
```

Interpreting Terms as Boolean Variables

```
(def-gl-thm simple-tautology
  :hyp t
  :concl (iff (xor (xor a b) a) b)
  :g-bindings `((a ,(gl::g-var 'a))
                (b ,(gl::g-var 'b))))
```

Term-level objects as IF tests generate fresh Boolean variables

Variable Generation

```
(def-gl-rewrite expand-loghead-bits
  (implies (syntaxp (and (integerp n) (term-gobj-p x)))
           (equal (loghead n x)
                  (if (zp n)
                      0
                    (logcons (if (logbitp 0 x) 1 0)
                             (loghead (1- n)
                                      (ash x -1)))))))
```

Variable Generation

```
(def-gl-thm add8-reverse
  :hyp t
  :concl (equal (+ (loghead 8 a)
                   (loghead 8 b)
                   (loghead 8 c))
                (+ (loghead 8 c)
                   (loghead 8 b)
                   (loghead 8 a)))
  :g-bindings `((a ,(gl::g-var 'a))
                (b ,(gl::g-var 'b))
                (c ,(gl::g-var 'c))))
```

Counterexamples & Constraints

SAT solver produces:

```
a                          ← NIL
(logbitp 3 x)              ←  T
(logbitp 5 (g :a r))       ← NIL
(integerp x)               ← NIL
```

- Need to determine counterexample for theorem variables from bit-level counterexample
- Need to inform SAT solver that `(logbitp n x)` implies `(integerp x)`

Heuristic Counterexample Translation

```
(def-glcp-ctrex-rewrite
   ((logbitp n i) v)
   (i (install-bit n (bool->bit v) i)))

(def-glcp-ctrex-rewrite
   ((g field rec) v)
   (rec (s field v rec)))
```

“If a term matching … is assigned value …,
then replace the current value for … with …”

Adding Boolean Constraints

```
(def-gl-boolean-constraint logbitp-implies-integerp
   :bindings ((bit0 (logbitp n x))
              (intp (integerp x)))
   :body (implies bit0 intp))
```

“For every set of generated Boolean variables matching …, assume constraint …”

“Forward-chaining” version:

```
(def-gl-boolean-constraint logbitp-implies-integerp
   :bindings ((bit0 (logbitp n x)))
   :body (implies bit0 (integerp x)))
```

Fighting Case-Splits

```
(let* ((st (if c1 (s :a v1 st) st))
       (st (if c2 (s :b v2 st) st)))
    st)
```

→

```
(if c2
    (if c1
        (s :b v2 (s :a v1 st))
      (s :b v2 st))
  (if c1 (s :a v1 st) st))
```

Fighting Case-Splits: Branch Merge Rules

```
(def-gl-branch-merge merge-if-of-s
  (equal (if c (s k v rec1) rec2)
         (s k (if c v (g k rec2)) (if c rec1 rec2))))
```

→

```
(s :b (if c2 v2 (g :b st))
   (s :a (if c1 v1 (g :a st))
       st))
```

(Often better, especially if values are of a type that can also be merged.)

Example: simple-machine-bitcount

```
(defun simple-machine-step (instr st)
  (case-match instr
    (('add dest src1 src2) (s dest
                              (loghead 32 (+ (loghead 32 (g src1 st))
                                             (loghead 32 (g src2 st))))
                              st))
    (('sub dest src1 src2) (s dest
                              (loghead 32 (- (loghead 32 (g src1 st))
                                             (loghead 32 (g src2 st))))
                              st))
    (('mask dest src maskval) (s dest
                                (logand (loghead 32 (g src st)) maskval)
                                st))
    (('rsh dest src shamt)   ...)
    (('cmul dest src const)  ...)
    (& st)))
```

Example: simple-machine-bitcount

```
(defconst *simple-machine-bitcount*
  '((rsh tmp1 v 1)
    (mask tmp1 tmp1 #x55555555)
    (sub v v tmp1)
    (mask tmp1 v #x33333333)
    (rsh tmp2 v 2)
    (mask tmp2 tmp2 #x33333333)
    (add v tmp1 tmp2)
    (rsh tmp1 v 4)
    (add tmp1 v tmp1)
    (mask tmp1 tmp1 #x0f0f0f0f)
    (cmul tmp1 tmp1 #x01010101)
    (rsh c tmp1 24)))
```

Algorithm found at:
Sean Aron Anderson. *Bit Twiddling Hacks*, 1997-2005,
Available at:
https://graphics.stanford.edu/~seander/bithacks.html

Example: simple-machine-bitcount

```
(def-gl-thm simple-machine-computes-logcount
  :hyp t
  :concl (b* ((v (g 'v initst))
              (finalst (simple-machine-run *simple-machine-bitcount*
                                           initst))
              (c (g 'c finalst)))
           (equal c (logcount (loghead 32 v))))
  :g-bindings nil)
```

Example: simple-machine-bitcount

```
(def-gl-thm simple-machine-logcount-final-st-wormhole-abstracted
  :hyp t
  :concl (b* ((v (g 'v initst))
              (finalst (simple-machine-run *simple-machine-bitcount* initst)))
           (equal finalst
                  ;; answer is correct:
                  (s 'c (logcount (loghead 32 v))
                     ;; scratch variables are whatever they are:
                     (s 'tmp1      (hide (g 'tmp1 finalst))
                        (s 'tmp2   (hide (g 'tmp2 finalst))
                           (s 'v   (hide (g 'v finalst))
                              ;; everything else is unchanged:
                              initst))))))
  :g-bindings nil)
```

Conclusions

Is it usable?

Focuses of future work:

- Debugging
- Improving rule libraries
- Maintainability
  - Term-level support may allow simpler implementation
- Will find out as we get more miles on it.

Record equality rules

```
(def-gl-rewrite equal-of-s
  (equal (equal (s k v x) y)
         (and (equal v (g k y))
              (gs-equal-except (list k) x y))))

(def-gl-rewrite gs-equal-except-of-s
  (equal (gs-equal-except lst (s k v x) y)
         (if (member k lst)
             (gs-equal-except lst x y)
           (and (equal v (g k y))
                (gs-equal-except (cons k lst) x y)))))

(def-gl-rewrite gs-equal-except-same
  (equal (gs-equal-except lst x x) t))
```