Bit-Blasting ACL2 Theorems

Sol Swords and Jared Davis

Centaur Technology, Inc.

November, 2011
Some guy on the Internet says that this C code counts bits:

\[
\begin{align*}
    v &= v - ((v >> 1) \& 0x55555555); \\
    v &= (v \& 0x33333333) + ((v >> 2) \& 0x33333333); \\
    c &= ((v + (v >> 4) \& 0x0F0F0F0F) * 0x01010101) >> 24;
\end{align*}
\]

He’s right.

Can you prove it in ACL2?

What would it look like?
Our proof, by bit-blasting

(defun fast-logcount-32 (v)
  (let* ((v (- v (logand (ash v -1) #x55555555)))
         (v (+ (logand v #x33333333)
               (logand (ash v -2) #x33333333))))
         (ash (32* (logand (+ v (ash v -4)) #x0F0F0F0F)
                    #x01010101)
              -24)))

(def-gl-thm fast-logcount-32-correct
  :hyp       (unsigned-byte-p 32 x)
  :concl     (equal (fast-logcount-32 x) (logcount x))
  :g-bindings ‘((x , (g-int 0 1 33))))
The bit-blasting approach

Bit-blasting lets you automatically prove “finite” theorems
- You get a real ACL2 theorem (no trust-tags)
- You get counterexamples to non-theorems
- You don’t need to understand the implementation
- You don’t have to change the proof when the implementation changes

We have used it to verify industrial hardware designs
- Scalar and packed integer operations (easy)
- Float/integer conversions, comparisons (easy)
- Floating point addition (requires case splitting)
- Integer and FP multiplication (requires decomposition)
1. How bit-blasting works
   - Bit-level objects
   - Symbolic objects
   - Computing with symbolic objects
   - Proving theorems with symbolic execution

2. How to get started!
Imagine representing integers as lists of bits

(t nil t nil) means 5
(t t t nil nil nil) means 7

And writing functions that operate on this representation

(defun bitlist-logand (x y)
  (if (or (atom x) (atom y))
      nil
    (cons (and (car x) (car y))
          (bitlist-logand (cdr x) (cdr y))))
Bit-level ACL2 objects

We could extend this idea to represent other ACL2 objects

\[
\begin{align*}
\text{(:int } t \text{ nil } t \text{ nil)} & \quad \text{means 5} \\
\text{(:char } t \text{ nil ...)} & \quad \text{means } \#\textbackslash\text{A} \\
\text{(:bool } t) & \quad \text{means } t
\end{align*}
\]

And write bit-level analogues of the ACL2 primitives

\[
\begin{align*}
\text{(defun my-integerp (x) )} \\
& \quad \text{(equal (car x) :int))}
\end{align*}
\]

\[
\begin{align*}
\text{(defun my-ifix (x) )} \\
& \quad \text{(if (my-integerp x) x '(:int nil)))}
\end{align*}
\]

\[
\begin{align*}
\text{(defun my-logand (x y) )} \\
& \quad \text{(cons :int (bitlist-logand (cdr (my-ifix x))}} \\
& \quad \quad \quad \text{(cdr (my-ifix y)))))
\end{align*}
\]
Symbolic objects are like this, but have **Boolean expressions** instead of bits

- `(:bool X_0)` can mean `t` or `nil`
- `(:int X_0 false true false)` can mean 4 or 5
- `(:int X_0 X_1 false false)` can mean 0, 1, 2, or 3
- `(:int X_0 \neg X_0 false)` can mean 1 or 2
- `(:int (X_0 \wedge X_1) false)` can mean 0 or 1

The value of a symbolic object depends on an **environment**

\[ \text{eval}(\text{symbolic object, env}) \rightarrow \text{ACL2 object} \]

The environment just binds \( X_0, X_1, \ldots \), to `t` or `nil`
Computing with symbolic objects

You can compute with symbolic objects without an environment.

Example 1
- Let \( A = (:\text{int} \quad X_0 \quad \text{false}) \); 0 or 1
- Let \( B = (:\text{int} \quad X_1 \quad \text{false}) \); 0 or 1
- \( A \& B = (:\text{int} \quad (X_0 \land X_1) \quad \text{false}) \); 0 or 1

Example 2
- Let \( A = (:\text{int} \quad \text{true} \quad X_0 \quad \text{false}) \); 1 or 3
- Let \( B = (:\text{int} \quad X_1 \quad \text{true} \quad \text{false}) \); 2 or 3
- \( A \& B = (:\text{int} \quad X_1 \quad X_0 \quad \text{false}) \); 0, 1, 2, or 3

Example 3
- Let \( A = (:\text{int} \quad X_0 \quad X_1 \quad \text{false}) \); 0, 1, 2, or 3
- Let \( B = (:\text{int} \quad X_2 \quad \text{false} \quad \text{false}) \); 0 or 1
- \( A == B = (:\text{bool} \quad (X_0 \leftrightarrow X_2) \land \neg X_1) \); t or nil
The main change we need

(defun bitlist-logand (x y)
  ;; x and y are lists of bits
  (if (or (atom x) (atom y))
      nil
      (cons (and (car x) (car y))
            (bitlist-logand (cdr x) (cdr y))))
⇒

(defun symbolic-bitlist-logand (x y)
  ;; x and y are lists of Boolean expressions
  (if (or (atom x) (atom y))
      nil
      (cons (and-exprs (car x) (car y))
            (symbolic-bitlist-logand (cdr x) (cdr y))))
Symbolic execution

We write symbolic analogues for most ACL2 primitives

Correctness example:

\[(\text{eval } (\text{symbolic-logand } x \ y) \ \text{env}) = (\text{logand } (\text{eval } x \ \text{env}) (\text{eval } y \ \text{env}))\]

We write a McCarthy style interpreter that can symbolically execute terms

\[\text{interp}(\text{term}, \text{symbolic bindings}) \rightarrow \text{symbolic object}\]

Example:

\[(\text{interp '}(\text{consp } x) '((x \ . \ x_{\text{sym}}))) = (\text{symbolic-consp } x_{\text{sym}})\]
We have certain symbolic objects

(:bool X₀) can mean t or nil
(:int X₀ false true false) can mean 4 or 5

The value of a symbolic object depends on an environment
eval(symbolic object, env) → ACL2 object

But we can compute on them without an environment.
interp(term, symbolic bindings) → symbolic object
Proving theorems by symbolic execution

Symbolic execution can be used as a proof procedure ("bit blasting")

Example:

\[(\text{implies} \ (\text{unsigned-byte-p} \ 32 \ x) \ \\
\quad \ (\text{equal} \ (\text{fast-logcount-32} \ x) \ \\
\quad \ (\text{logcount} \ x)))\]

- Choose a symbolic object, \(x_{sym}\), that covers the hypothesis, i.e.,
  \[\forall x, \ (\text{unsigned-byte-p} \ 32 \ x) \rightarrow (\exists \text{ env} . \ (\text{eval} \ x_{sym} \ \text{env}) = x)\]

- Symbolically execute the conclusion on \(x_{sym}\)
  \[(\text{interp} \ '(\text{equal} \ (\text{fast-logcount-32} \ x) \ (\text{logcount} \ x)) \ \\
  \quad '((x . \ x_{sym})))\]

- Inspect the result. Can it evaluate to \text{nil}?
  - Yes — You have just found a counterexample
  - No — You have just proved the theorem
Proving the example theorem

(implies (unsigned-byte-p 32 x)
  (equal (fast-logcount-32 x)
    (logcount x)))

We need a symbolic object \( x_{\text{sym}} \) that can represent every value that satisfies the hypothesis, i.e., 0, 1, \ldots, \( 2^{32} - 1 \).

This is easy:
Let \( x_{\text{sym}} = (:\text{int} \ X_0 \ X_1 \ \ldots \ X_{31} \ X_{32}) \) (yes, 33 bits)

(def-gl-thm fast-logcount-32-correct
  :hyp (unsigned-byte-p 32 x)
  :concl (equal (fast-logcount-32 x) (logcount x))
  :g-bindings '((x ,(g-int 0 1 33))))
Def-gl-thm is our interface for bit-blasting ACL2 theorems
- It is based on a verified clause processor (no trust tags)
- It gives you a real ACL2 defthm on success
- It gives you good counterexamples to non-theorems

It splits your proof into two parts:
- Coverage — do your symbolic objects cover the whole hypothesis? (a “normal” ACL2 proof, usually automatic)
- Symbolic execution of the conclusion (automatic, but can be computationally hard)
You can use this stuff!

To get started, see books/centaur/README

To learn to use it effectively, see the paper

- Optimizing GL execution
- Debugging performance problems
- Splitting proofs into cases
- Using AIG versus BDD representations
- Pointers to :doc topics and Sol's dissertation