Bit-Blasting ACL2 Theorems

Sol Swords and Jared Davis

Centaur Technology, Inc.

November, 2011

A simple challenge

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

```
v = v - ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
c = ((v + (v >> 4) & 0x0F0F0F0F) * 0x01010101) >> 24;
```

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)

The rest of this talk

1. How bit-blasting works
   - Bit-level objects
   - Symbolic objects
   - Computing with symbolic objects
   - Proving theorems with symbolic execution
2. How to get started!

Bit-level integers

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

`(:int  t nil t nil)`	means 5
`(:char t nil ...)`	means `#\A`
`(:bool t)`	means t

And write bit-level analogues of the ACL2 primitives

```
(defun my-integerp (x)
  (equal (car x) :int))

(defun my-ifix (x)
  (if (my-integerp x) x '(:int nil)))

(defun my-logand (x y)
  (cons :int (bitlist-logand (cdr (my-ifix x))
                             (cdr (my-ifix y)))))
```

Symbolic objects

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

$$\texttt{eval}(\textit{symbolic object}, \textit{env}) \rightarrow \textit{ACL2 object}$$

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 = (:int X_0 *false*) *; 0 or 1*
- Let B = (:int X_1 *false*) *; 0 or 1*
- A & B = (:int $(X_0 \wedge X_1)$ *false*) *; 0 or 1*

Example 2

- Let A = (:int *true* X_0 *false*) *; 1 or 3*
- Let B = (:int X_1 *true* *false*) *; 2 or 3*
- A & B = (:int X_1 X_0 *false*) *; 0, 1, 2, or 3*

Example 3

- Let A = (:int X_0 X_1 *false*) *; 0, 1, 2, or 3*
- Let B = (:int X_2 *false* *false*) *; 0 or 1*
- A == B = (:bool $(X_0 \leftrightarrow X_2) \wedge \neg X_1$) *; 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)))))
```

$\Longrightarrow$

```
(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:

```
(eval (symbolic-logand x y) env)
  =
(logand (eval x env) (eval y env))
```

We write a McCarthy style interpreter that can symbolically execute terms

$$\text{interp}(\textit{term}, \textit{symbolic bindings}) \rightarrow \textit{symbolic object}$$

Example:

(interp '(consp x) '((x . x_{sym})))

  =

(symbolic-consp x_{sym})

Review!

We have certain symbolic objects

`(:bool` X_0`)`	can mean t or nil
`(:int` X_0 *false* *true* *false*`)`	can mean 4 or 5

The value of a symbolic object depends on an environment

$$\texttt{eval}(\textit{symbolic object}, \textit{env}) \rightarrow \textit{ACL2 object}$$

But we can compute on them without an environment.

$$\text{interp}(\textit{term}, \textit{symbolic bindings}) \rightarrow \textit{symbolic object}$$

Proving theorems by symbolic execution

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

Example:

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

- Choose a symbolic object, x_{sym}, that covers the hypothesis, i.e.,

  $\forall x$, `(unsigned-byte-p 32 x)` $\rightarrow$ ($\exists$ `env` . `(eval` x_{sym} `env)` $=$ `x`)

- Symbolically execute the conclusion on x_{sym}

  ```
  (interp '(equal (fast-logcount-32 x) (logcount x))
          '((x . x_sym)))
  ```

- Inspect the result. Can it evaluate to `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_{sym} that can represent every value that satisfies the hypothesis, i.e., $0, 1, \ldots, 2^{32} - 1$.

This is easy:

Let x_{sym} = (: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))))
```

Proving ACL2 theorems by bit-blasting

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