Integrating SAT Solvers with ACL2 (part 2)

Erik Reeber

1/26/05

Review of Part 1

- SAT Solvers
  - Find satisfying instances of Boolean variables in conjunctive normal form
  - Used as an alternative to BDDs in fully automated hardware verification tools
- Decidable Fragment of ACL2
  - list structures and unrollable functions
  - detection algorithm
  - Can express hardware invariants

Overview

- Conversion Algorithm
- Results
- Conclusion
- General Mechanism For Integrating External Tools (Discussion)

Converting ACL2 into CNF

- Number variables
- Create clausal form & negate property
- unroll functions and create clauses
- Eliminate Destructors & add list axioms
- Remove **iff** expressions and constants
- Optimizations

Example

```
(defun not-list (n x)
  (if (zp n)
      nil
    (cons (not (car x)) (not-list (1- n) (cdr x)))))

(defun n-bleq (n x y)
  (if (zp n)
      t
    (if (iff (car x) (car y))
       (n-bleq (1- n) (cdr x) (cdr y))
      nil)))

;; The (not (not x)) == x

(thm (n-bleq 2 (not-list 2 (not-list 2 x)) x)

     :hints (("Goal" :sat nil)))
```

Numbering variables

- In internal form constants are quoted
- We use numbers to represent variables

```
(n-bleq 2 (not-list 2 (not-list 2 x)) x)
=>
(n-bleq ‘2 (not-list ‘2 (not-list ‘2 1)) 1)
```

Create Clause & Negate Property

- $\exists x_0, x_1, \ldots x_n$ (and (or …) (or …) …) ==
  $\forall\ x_0, x_1, \ldots x_n$ (not (and (or …) (or …) …)
- Our first step is to change to a negated and expression

```
(n-bleq '2 (not-list '2 (not-list '2 1)) 1)
=> {negate and add a variable}
(not (and (not 2)
          (bceq 2 (n-bleq '2 (not-list '2 (not-list '2
  1)) 1)))
```

BCEQ

- I use **bceq** rather than equal to emphasize that I only care about list structures.
- **bceq** is an equivalence relation
- Think of **bceq** as equal

```
(defun bceq (x y)
 (if (or (consp x) (consp y))
    (and (consp x) (consp y)
         (iff x y))
   (and (bceq (car x) (car y))
        (bceq (cdr x) (cdr y)))))
```

Converting to BC-CNF

- We’re done when every clause:
  - Contains 0 or 1 **bceq** expressions and
  - The second **bceq** argument is a constant or an expression of **car**, **cdr**, **consp**, and **not**.

Converting into BC-CNF (cont)

- Looking at the first ill-formed clause, let *f* be the top-level function of its second **bceq** argument:
  - If *f* is an **if** with a simple condition, break it into clauses assuming the condition or its negation
  - Otherwise if *f* is **if**, create a variable for the condition
  - If *f* is **cons**, break into clauses for **consp**, **car**, and **cdr**
  - If *f* is a defined function with simple arguments, open and simplify
  - Otherwise if *f* is defined, create variables for complex arguments
  - Otherwise, delete the clause

Example

```
(nand
 (not 2)
 (bceq 2 (n-bleq '2 (not-list '2 (not-list '2 1)) 1)))
=> {create variables for n-bleq’s args}
(nand
 (not 2)
 (bceq 3 (not-list '2 (not-list '2 1)) 1)
 (bceq 2 (n-bleq '2 3 1))))
⇒ {create variables for not-list’s args}
(not (and (not 2)
          (bceq 4 (not-list '2 1))
          (bceq 3 (not-list '2 4))
          (bceq 2 (n-bleq '2 3 1))))
=> {open not-list}
```

Example

```
(nand
 (not 2)
 (bceq 4 (cons (not (car 1)) (not-list '1 (cdr 1))))
 (bceq 3 (not-list '2 4))
 (bceq 2 (n-bleq '2 3 1))))
⇒ {break up cons}
(nand
 (not 2)
 (consp 4)
 (bceq (car 4) (not (car 1))
 (bceq (cdr 4) (not-list '1 (cdr 1)))
 (bceq 3 (not-list '2 4))
 (bceq 2 (n-bleq '2 3 1))))
⇒ {the next step is to open the next not-list, we’ll
⇒  skip ahead so that all the not-lists are gone}
```

Example

```
(nand
 (not 2)
 (consp 4)
 (bceq (car 4) (not (car 1))
 (consp (cdr 4))
 (bceq (cadr 4) (not (cadr 1)))
 (bceq (cddr 4) 'nil)
 (consp 3)
 (bceq (car 3) (not (car 4))
 (consp (cdr 3))
 (bceq (cadr 3) (not (cadr 4)))
 (bceq (cddr 3) 'nil)
 (bceq 2 (n-bleq '2 3 1))))
=> {open the n-bleq}
```

consp clauses form from breaking up **cons**

Eventually, not-list simplifies to **nil**

Example

```
(nand
  …
 (bceq 2 (if (iff (car 3) (car 1))
           (n-bleq '1 (cdr 3) (cdr 1))
          'nil))))
=> {create variable for if condition}
(nand
  …
 (bceq 5 (if (car 3) (car 1) (not (car 1))))
 (bceq 2 (if 5 (n-bleq '1 (cdr 3) (cdr 1)) 'nil))))
⇒ {break up if}
(nand
  …
 (or (bceq 5 (car 1)) (not (car 3)))
 (or (bceq 5 (not (car 1))) (car 3))
 (bceq 2 (if 5 (n-bleq '1 (cdr 3) (cdr 1)) 'nil))))
```

Example

```
(nand
 (not 2)
 (consp 4)
 (bceq (car 4) (not (car 1))
 (consp (cdr 4))
 (bceq (cadr 4) (not (cadr 1)))
 (bceq (cddr 4) 'nil)
 (consp 3)
 (bceq (car 3) (not (car 4))
 (consp (cdr 3))
 (bceq (cadr 3) (not (cadr 4)))
 (bceq (cddr 3) 'nil)
 (or (bceq 5 (car 1)) (not (car 3)))
 (or (bceq 5 (not (car 1))) (car 3))
 (or (bceq 6 (cadr 1)) (not (cadr 3)))
 (or (bceq 6 (not (cadr 1))) (cadr 3))
 (or (bceq 2 't) (not 5) (not 6))
 (or (bceq 2 'nil) (not 5) 6)
 (or (bceq 2 'nil) 5)))
```

Case where bleq is true

Cases where bleq is false

Destructor Elimination

- Remove **car**, **cdr**, and **consp**
  - Find the variable parts we need
  - Create new numbers for these parts
  - Add list structure axioms
  - Create new clauses from the old ones

Defining Γ

- Keep a data structure Γ for each variable *v*
- Γ is **nil** if *v* isn't needed
- Otherwise Γ is a four-tuple:
  - (car-sub, cdr-sub, consp-needed, atom-needed)
  - car-sub is a data structure Γ for (car v)
  - cdr-sub is a data structure Γ for (cdr v)
  - consp-needed is a Boolean which is true if we need to know (consp **v)**
  - Atom-needed is a Boolean which is true if we need to know whether *v* is non-**nil**

Finding Γ

- We start by setting Γ to **nil** for each variable
- For each non-**bceq** literal:
  - If it is a **consp**, we set get the Γ structure for its argument (or build it) and set its third value to **t**
  - If it is a **not**, we get the Γ structure for its argument (or build it) and set its fourth value to **t**
  - Otherwise we the Γ structure for the literal (or build it) and set its fourth value to **t**

Example

- The non-bceq clauses are (not 2), (consp 4), (consp (cdr 4)), (consp 3), (consp (cdr 3)).
- This leads to:

```
1: nil

2: (nil nil nil t)

3: (nil (nil nil t nil) t nil)

4: (nil (nil nil t nil) t nil)

5: nil

6: nil
```

Example

- The non-bceq literals in bceq clauses are: (not (car 3)), (car 3), (not (cadr 3)), (cadr 3), (not 6), 6, and 5. This leads to:

```
1: nil
2: (nil nil nil t)
3: ((nil nil nil t) ((nil nil nil t) nil t nil) t nil)
4: (nil (nil nil t nil) t nil)
5: (nil nil nil t)
6: (nil nil nil t)
```

Finding Γ (continued)

- Examine **bceq** literals in backwards order:
  - If the second argument is a **car** or **cdr** expression Γ for both arguments. We set anything which is **t** in the first argument's Γ to **t** in the second argument's.
  - If the second argument is a **consp** or a **not** expression, and the first arguments non**-nil** value is **t**, then we treat the second argument like a non**-bceq** literal
  - If the second argument is constant, we ignore it
- In forwards order, the first argument of a **bceq** is new

Example

- The first literal we look at is (bceq 2 'nil), which is ignored.
- The first interesting literal is (bceq 6 (not (cadr 1))). In this case, since the fourth value of the Γ for 6 is **t**, we treat (not (cadr 1)) like a non-**bceq** literal.
  - Γ_1: (nil ((nil nil nil t) nil nil nil) nil nil)
- The next literal is (bceq 6 (cadr 1)), which leads to no change since 6 and (cadr 1) both have the Γ: (nil nil nil t).

Example

- The Γ list for our example thus becomes:

```
1 (x):
((nil nil nil t) ((nil nil nil t) nil nil nil) nil nil)

2 (n-bleq (not-list ‘2 …) x):
(nil nil nil t)

3 (not-list ‘2 (not-list ‘2 …):
((nil nil nil t) ((nil nil nil t) nil t nil) t nil)

4 (not-list ‘2 …):
((nil nil nil t) ((nil nil nil t) nil t nil) t nil)

5 (car x):
(nil nil nil t)

6 (cadr x):
(nil nil nil t)
```