A Case Study in Program Verification: Soundness of Theorem Provers

William McCune

Mathematics and Computer Science Division
Argonne National Laboratory

http://www.mcs.anl.gov/~mccune

Why Are We Doing This Project?

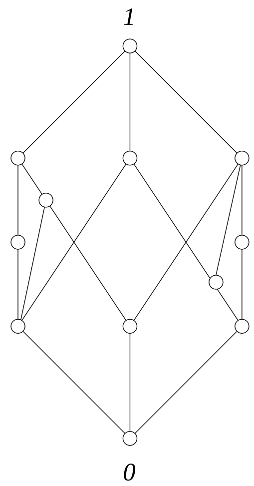
Axioms for ortholattices:

$$
\begin{array}{ll}
x \vee y = y \vee x & \\
(x \vee y) \vee z = x \vee (y \vee z) & \\
x \vee (y \vee y') = y \vee y' & \\
x'' = x & \\
x \wedge y = (x' \vee y')' & \textit{(definition of meet)} \\
x \wedge y = y \wedge x & \\
(x \wedge y) \wedge z = x \wedge (y \wedge z) & \\
x \vee (x \wedge y) = x &
\end{array}
$$

Does the following hold for ortholattices?

$$((x \vee y') \wedge (((x \wedge y) \vee (x' \wedge y)) \vee (x' \wedge y'))) = ((x \wedge y) \vee (x' \wedge y'))$$

No. Counterexample found by MACE:



Does the following hold for ortholattices?

$$(((x' \wedge y) \vee (x' \wedge y')) \vee (x \wedge (x' \vee y)))' \vee (x' \vee y) = 1$$

Yes. Proof found by EQP:

1	$x'' = x$	
2	$x' \vee (x \vee y) = x' \vee x$	
3	$(x \wedge y) \vee x = x$	
4	$(x' \vee y')' = x \wedge y$	
7	$x' \vee x = 1$	
8	$1 \vee x = 1$	[2,simp:7,7]
13	$b \vee (a' \vee ((b \wedge a') \vee ((a \wedge (b \vee a')) \vee (b' \wedge a')))') \neq 1$	
14	$x' \vee y' = (x \wedge y)'$	[4 → 1,flip]
15	$b \vee (a \wedge ((b \wedge a') \vee ((a \wedge (b \vee a')) \vee (b' \wedge a'))))' \neq 1$	[13,simp:14]
16	$(x' \vee y)' = y' \wedge x$	[1 → 4]
22	$(x' \wedge y)' = y' \vee x$	[16 → 1]
23	$x' \wedge y' = (x \vee y)'$	[1 → 16,flip]
24	$b \vee (a \wedge ((b \vee a)' \vee ((b \wedge a') \vee (a \wedge (b \vee a')))))' \neq 1$	[15,simp:23]
25	$(x' \vee y) \wedge y = y$	[3 → 16,simp:1,22,flip]
27	$(x \vee y) \wedge y = y$	[1 → 25]
28	$x' \vee ((y' \wedge x) \vee y) = 1$	[16 → 7]
31	$((x \vee y) \wedge z) \vee (y \wedge z) = (x \vee y) \wedge z$	[27' → 3]
32	$((x \wedge y) \vee z) \wedge (y \vee z) = (x \wedge y) \vee z$	[3' → 27]
33	$((x' \wedge y) \vee z)' = z' \wedge (y' \vee x)$	[16 → 16]
38	$((x' \vee y) \wedge z)' \vee x = z' \vee x$	[27' → 22,simp:22,flip]
146	$(x \wedge y)' \wedge (y' \vee (x' \wedge z)) = y' \vee (x' \wedge z)$	[14 → 32]
174	$x' \wedge (y' \vee (z' \wedge u)) = (((u' \vee z) \wedge y) \vee x)'$	[22 → 33,flip]
175	$((x' \vee y) \wedge z)' = z' \vee (y' \wedge x)$	[146,simp:174,31]
281	$x' \vee ((y' \wedge ((z' \vee u) \wedge x)) \vee (y \vee z)) = 1$	[38' → 28',simp:8]
2662	$x' \vee (((y \wedge z)' \wedge ((z' \vee u) \wedge x)) \vee z) = 1$	[3' → 281]
2663	$(((x \wedge y) \vee z)' \wedge (y' \vee u)) \vee (y \vee z) = 1$	[1 → 2662,simp:23]
2664	$x' \vee ((((x' \wedge y) \vee z)' \wedge (x \vee u)) \vee z) = 1$	[1 → 2663]
3141	$x' \vee ((((x' \wedge y) \vee ((x' \vee z) \wedge u))' \wedge (x \vee v)) \vee z) = 1$	[2664' → 3',simp:8,flip]
22528	$1 \neq 1$	[175 → 24,simp:3141]

We want to make sure the counterexample and proof are correct.

ACL2

(A Computational Logic for Applicative Common Lisp)

http://www.cs.utexas.edu/users/moore/acl2

ACL2 is a programming language and a theorem prover for proving properties of the programs.

Example: sorting a list.

– specification

– definition of sort function

– proof that the definition meets the specification

Sorting — Specification

Define ordered list and permutation of two lists:

```
(defun ordered (a)
  (cond ((atom a) t)
        ((atom (cdr a)) t)
        (t (and (<= (first a) (second a))
                (ordered (cdr a))))))

(defun remove1 (x a)
  (cond ((atom a) a)
        ((equal x (car a)) (cdr a))
        (t (cons (car a) (remove1 x (cdr a))))))

(defun permutation (a b)
  (cond ((atom a) (atom b))
        ((member-equal (car a) b)
         (permutation (cdr a) (remove1 (car a) b)))
        (t nil)))
```


Sorting — Definition

Definition of Insertion Sort (isort):

```
(defun insert (x a)
  (cond ((atom a) (cons x a))
        ((<= x (car a)) (cons x a))
        (t (cons (car a) (insert x (cdr a))))))

(defun isort (a)
  (cond ((atom a) a)
        (t (insert (car a) (isort (cdr a))))))
```

Correctness conjecture:

Isort produces an ordered permutation of its input.

Sorting — Proof of Ordered Property

We ask ACL2 to prove a theorem:

```
(defthm isort-ordered
  (ordered (isort a)))
```

ACL2 does this automatically.

– induction on (ISORT A)

– a subgoal cannot be proved by rewriting:

```
(IMPLIES (ORDERED (ISORT A2))
         (ORDERED (INSERT A1 (ISORT A2))))
```

– ACL2 generalizes the subgoal:

```
(IMPLIES (ORDERED B)
         (ORDERED (INSERT A1 B)))
```

– subgoal proved by induction on (INSERT A1 B)

Sorting — Proof of Permutation

```
(defthm isort-permutation
  (permutation a (isort a)))
```

ACL2 fails to prove this automatically.

– a subgoal cannot be proved by rewriting:

```
(IMPLIES (AND (MEMBER-EQUAL A1 (INSERT A1 (ISORT A2)))
              (PERMUTATION A2 (ISORT A2)))
         (PERMUTATION A2
                      (REMOVE1 A1 (INSERT A1 (ISORT A2)))))
```

– ACL2 generalizes the subgoal *too much*:

```
(IMPLIES (AND (CONSP C)
              (MEMBER-EQUAL A1 C)
              (PERMUTATION A2 B))
         (PERMUTATION A2 (REMOVE1 A1 C)))
```

– ACL2 fails, because this is not a theorem. The user sees what went wrong and asks ACL2 to prove a lemma:

```
(defthm remove1-insert
  (equal (remove1 x (insert x a)) a))
```

– ACL2 proves the lemma. Now ACL2 can prove `isort-permutation` (by doing 3 inductions).

ACL2 — Guards

- ACL2 is an untyped logic.
- Guards allow some type checking.
- Isort definition with guards:

```
(defun insert (x a)
  (declare (xargs :guard (and (integerp x)
                              (integer-listp a))))
  (cond ((atom a) (cons x a))
        ((<= x (car a)) (cons x a))
        (t (cons (car a) (insert x (cdr a))))))

(defun isort (a)
  (declare (xargs :guard (integer-listp a)))
  (cond ((atom a) a)
        (t (insert (car a) (isort (cdr a))))))
```

- Guards are verified when functions are introduced.
- Guards are *not* part of the logic. Their purpose is to make ACL2 compatible with Common Lisp.
- Additional benefit: guard verification catches lots of bugs before you attempt the interesting parts of the verification.

ACL2 — Proving Theorems

- heuristic rewriting
- case analysis
- generalization
- induction based on recursive definitions
- *lots of hints from the user*

ACL2 — Advanced Features

- congruence rules
- quantifiers
- encapsulation and functional instantiation
- single threaded objects: fast execution

Our Style of Resolution Theorem Proving

- Start with a closed WFF (the conjecture)
- Negate it (we prove by contradiction)
- Preprocess
  - negation normal form
  - rename variables
  - Skolemize
  - pull universal quantifiers to the top
  - conjunctive normal form
  - drop the universal quantifiers
- Now we have a quantifier-free CNF formula, that is, a conjunction of clauses.
- Search
  - Try to refute it (prove the conjecture), or
  - try to find a model (disprove the conjecture).

First-order Logic Defined in ACL2

Well-formed term and formula, interpretation, substitution, evaluation.

- Interpretation of a first-order formula:
  - a nonempty domain
  - a set of functions from the domain to the domain
  - a set of relations over the domain
- (Finite) interpretation in ACL2:
  - domain: leaves of a binary tree
  - functions: list of tables
  - relations: list of tables

Evaluation

Evaluate a formula (f) in a finite interpretation (i).

```
(mutual-recursion  ;; termination measure and guards omitted

 (defun feval (f i)  ;; recurse through formula
   (cond
    ((equal f 'true) t)
    ((equal f 'false) nil)
    ((wfnot f) (not     (feval (a1 f) i)))
    ((wfand f) (and     (feval (a1 f) i) (feval (a2 f) i)))
    ((wfor  f) (or      (feval (a1 f) i) (feval (a2 f) i)))
    ((wfimp f) (implies (feval (a1 f) i) (feval (a2 f) i)))
    ((wfiff f) (iff     (feval (a1 f) i) (feval (a2 f) i)))
    ((wfquant f) (feval-d f (domain i) i))
    (t (eval-atomic f i))))

 (defun feval-d (f dom i)  ;; recurse through domain
   (cond
    ((not (wfquant f)) nil)  ;; default value
    ((atom dom)   (feval (subst-free (a2 f) (a1 f) dom) i))
    ((wfall f)    (and (feval-d f (car dom) i)
                       (feval-d f (cdr dom) i)))
    ((wfexists f) (or  (feval-d f (car dom) i)
                       (feval-d f (cdr dom) i)))
    (t nil)))  ;; default value
 )
```

This is the core of the specification for our soundness theorems.

Negation-Normal Form (NNF)

- Definition of `nnf`
  - eliminate `'imp` and `'iff` connectives
  - move `'not` signs up against atomic formulas
- Soundness theorem

```
(defthm nnf-fsound
  (equal (feval (nnf f) i)
         (feval f i))
```

- Other theorems we will need

```
(defthm nnf-nnfp   ;; nnfp checks for NNF
  (implies (wff f)
           (nnfp (nnf f))))

(defthm nnf-preserves-wff
  (implies (wff f)
           (wff (nnf f))))

(defthm nnf-preserves-free-vars
  (equal (free-vars (nnf f))
         (free-vars f)))
```


Skolemization

- Introduce Skolem functions to replace existentially quantified variables.

$$\begin{array}{ll} & \forall x \forall y[x \not\subseteq y \rightarrow \exists z(z \in x \wedge z \notin y)] \\ nnf: & \forall x \forall y[x \subseteq y \vee \exists z(z \in x \wedge z \notin y)] \\ skolem: & \forall x \forall y[x \subseteq y \vee (f(x,y) \in x \wedge f(x,y) \notin y)] \end{array}$$

- Soundness theorem
  - We have to extend the interpretation so that it interprets the new Skolem functions.

```
(defthm skolemize-fsound
  (equal (feval (skolem f)
                (skolem-extend f i))
         (feval f i)))
```


Preprocessing

Definition

```
(defun preprocess (f)
  (declare (xargs
            :guard (and (wff f)
                        (not (free-vars f)))))
  (right-assoc
   (cnf
    (pull-quants
     (skolemize
      (rename-all
       (nnf f)))))))
```

Soundness theorem

```
(defthm preprocess-fsound
  (equal (feval (preprocess f)
                (skolemize-extend
                 (rename-all (nnf f)) i))
         (feval f i)))
```

Proof Objects

A detailed line-by-line proof that is easy to check.

```
((1 (input)                    . (or (not (P x)) (Q x)))
 (2 (input)                    . (P (a)))
 (3 (input)                    . (not (Q y)))
 (4 (instance 1 ((x . (a)))) . (or (not (P (a))) (Q (a))))
 (5 (resolve 2 4)              . (Q (a)))
 (6 (instance 3 ((y . (a)))) . (not (Q (a))))
 (7 (resolve 5 6)              . false))
```

Types of inference in proof objects

- input
- instantiate
- resolve (modus ponens)
- paramodulate (equality substitution)
- flip equalities
- propositional simplification

Interface to External Prover

Assert the existence of `external-prover`. We don't need any properties of it for the soundness proofs.

```
(defstub external-prover (prf) t)

(defun refute-n-check (f)

    (let* ((clauses (remove-leading-alls f))
           (input (initial-proof clauses))
           (output (external-prover input)))

      (if (and (wfproof output)
               (equal (input-steps input)
                      (input-steps output))
               (check-proof output))

          (universal-closure
               (proof-to-formula output))

          f)))
```

Soundness:

```
(defthm refute-n-check-fsound
  (equal (feval (refute-n-check f) i)
         (feval f i))
```


Example of `refute-n-check`

Input to function `refute-n-check`:

```
(all x (all y (and (and (or (not (P x)) (Q x))
                        (P (a)))
                        (not (Q y)))))
```

Input to `external-prover`:

```
((1 (input)                    . (or (not (P x)) (Q x)))
 (2 (input)                    . (P (a)))
 (3 (input)                    . (not (Q y))))
```

Output of `external-prover`:

```
((1 (input)                    . (or (not (P x)) (Q x)))
 (2 (input)                    . (P (a)))
 (3 (input)                    . (not (Q y)))
 (4 (instance 1 ((x . (a)))) . (or (not (P (a))) (Q (a))))
 (5 (resolve 2 4)              . (Q (a)))
 (6 (instance 3 ((y . (a)))) . (not (Q (a))))
 (7 (resolve 5 6)              . false))
```

Output of `refute-n-check` (if check succeeds):

```
(all x (all y (and (and (and (and (and (and
                (or (not (P x)) (Q x))
                (P (a)))
                (not (Q y)))
                (or (not (P (a))) (Q (a))))
                (Q (a)))
                (not (Q (a))))
                false)))
```


Top Functions and Soundness Theorems

```
(defun refutation-attempt (f)
  (simplify (refute-n-check (preprocess f))))

(defthm refutation-attempt-fsound
  (equal (feval (refutation-attempt f)
                (skolem-extend (rename-all (nnf f)) i))
         (feval f i)))

(defun refuted (f)
  (cond ((or (not (wff f)) (free-vars f)) nil)
        (t (equal (refutation-attempt f) 'false))))

(defthm refutation-is-fsound  ;; Top Theorem #1
  (implies (refuted f)
           (and (wff f)
                (not (free-vars f))
                (not (feval f i)))))
```

Or, stated positively,

```
(defun proved (f)
  (cond ((or (not (wff f)) (free-vars f)) nil)
        (t (refuted (list 'not f)))))

(defthm proof-is-fsound  ;; Top Theorem #2
  (implies (proved f)
           (and (wff f)
                (not (free-vars f))
                (feval f i))))
```

Are Finite Interpretations Good Enough?

Consider

$$[f(x) = f(y) \to x = y] \to [\forall x \exists y \ f(y) = x]$$

that is,

one-to-one $\Rightarrow$ *onto*

– This is true for finite interpretations, but not in general.

– *Could our system approve a claimed proof of this nontheorem?*

Counterexamples

Function `model-attempt` calls a function `external-modeler` to search for a model.

If a model is found, it is checked by calling the evaluation function.

```
(defthm model-attempt-fsound  ;; Top Theorem #3
  (implies (model-attempt f)
           (and (wff f)
                (not (free-vars f))
                (feval f (model-attempt f)))))
```

Or, stated positively,

```
(defun countermodel-attempt (f)
  (cond ((or (not (wff f)) (free-vars f)) nil)
        (t (model-attempt (list 'not f)))))

(defthm countermodel-attempt-fsound  ;; Top Theorem #4
  (implies (countermodel-attempt f)
           (and (wff f)
                (not (free-vars f))
                (not (feval f (countermodel-attempt f))))))
```

We don't have to worry about finiteness here.

A Hard Verification Theorem

Function `(alls vars f)` tacks on universally quantified variables, for example,

```
(alls '(x y z) 'f) = '(all x (all y (all z f)))
```

Function `(uc f)` gives the universal closure of a formula:

```
(defun uc (f)
  (alls (free-vars f) f))
```

We need to prove something like

```
(defthm instance-fsound-for-1-substitution
  (implies (and (quantifier-free f)
                (feval (uc f) i))
           (feval (uc (subst-free f x tm)) i)))
```


Sketch of Proof (1)

```
(defthm instance-term-sound
  (implies (and (ground tm)
                (feval (list 'all x f) i))
           (feval (subst-free f x tm) i)))

(defthm feval-alls-subset
  (implies (and (subsetp-equal vars1 vars2)
                (not (free-v (alls vars1 f))))
           (equal (feval (alls vars1 f) i)
                  (feval (alls vars2 f) i))))
```

To prove the preceding theorem,

– Introduce a new evaluation function

`(keval vars f dom n dom2 i)`

which evaluates `(alls vars f)` using domain `dom2` instead of `dom` for the $\mathtt{n}^{th}$ variable of `vars`.

Sketch of Proof (2)

```
(defthm instance-xsound-alls-1b
  (implies
   (and (quantifier-free f)
        (not (free-vars (list 'all x f)))
        (feval (list 'all x f) i)
        (not (free-vars (alls v (subst-free f x tm)))))
   (feval (alls v (subst-free f x tm)) i)))
```

To prove the preceding theorem we used a special induction scheme (`var-induct-tm vars f tm dom i`) that combines properties of `eval`, `alls`, and term substitution.

```
(defthm instance-xsound-alls-2b
  (implies
   (and (quantifier-free f)
        (not (free-vars (alls w (list 'all x f))))
        (not (member-equal x w))
        (feval (alls w (list 'all x f)) i)
        (not (free-vars (alls (append w v) (subst-free f x tm)))))
   (feval (alls (append w v) (subst-free f x tm)) i)))
```

Yet another special induction scheme was used.

Sketch of Proof (3)

Several lemmas about sets of variables, for example,

```
(defthm free-vars-in-subst
  (subsetp (free-vars (subst-free f x tm))
           (union (remove x (free-vars f))
                  (vars-in-term tm))))
```

Finally, we can prove the theorem we need:

```
(defthm instance-xsound-for-1-substitution
  (implies (and (quantifier-free f)
                (xeval (uc f) (domain i) i)
                (variable-term x))
           (xeval (uc (subst-free f x tm)) (domain i) i))
  :hints
  (("Goal"
    :do-not-induct t
    :in-theory (disable quantifier-free free-vars subst-free xeval)
    :use ((:instance instance-xsound-alls-2b
                     (g (subst-free f x tm))
                     (dom (domain i))
                     (w (remove-equal x (free-vars f)))
                     (v (set-difference-equal
                         (vars-in-term tm)
                         (remove-equal x (free-vars f)))))
          (:instance xeval-alls-subset
                     (b (append (remove-equal x (free-vars f))
                                (list x)))
                     (a (free-vars f)))
          (:instance xeval-alls-subset
                     (b (append (remove-equal x (free-vars f))
                                (set-difference-equal
                                 (vars-in-term tm)
                                 (remove-equal x (free-vars f)))))
                     (a (free-vars (subst-free f x tm)))
                     (f (subst-free f x tm)))
          )))
  :rule-classes nil)
```


Other Difficulties with ACL2

- The user has to know how rewriting works and manage the set of rewrite rules. For example, the Knuth-Bendix problem.
- There is no real search. If a heuristic leads to failure, ACL2 does not try another path.
- Case explosions: the user must carefully manage definitions.
- Counterexample search (analogous to MACE) would help.