Chapter 1

IVY: A PREPROCESSOR AND PROOF CHECKER FOR FIRST-ORDER LOGIC

William McCune
Mathematics and Computer Science Division
Argonne National Laboratory
mccune@mcs.anl.gov

Olga Shumsky
Department of Electrical and Computer Engineering
Northwestern University
shumsky@ece.nwu.edu

Abstract This case study shows how non-ACL2 programs can be combined with ACL2 functions in such a way that useful properties can be proved about the composite programs. Nothing is proved about the non-ACL2 programs. Instead, the results of the non-ACL2 programs are checked at run time by ACL2 functions, and properties of these checker functions are proved. The application is resolution/paramodulation automated theorem proving for first-order logic. The top ACL2 function takes a conjecture, preprocesses the conjecture, and calls a non-ACL2 program to search for a proof or countermodel. If the non-ACL2 program succeeds, ACL2 functions check the proof or countermodel. The top ACL2 function is proved sound with respect to finite interpretations.

Introduction

Our ACL2 project arose from a different kind of automated theorem proving. We work with fully automatic resolution/paramodulation theo-

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

rem provers for (quantified) first-order logic with equality. Such provers are quite distinct from ACL2, both in use and in applicability. Let's call them *Otter-class provers* [McCune, 1994c, McCune and Wos, 1997]. Otter-class provers rely heavily on search and are usually coded in low-level languages such as C, with numerous tricks, hacks, and optimizations, so formally proving things about them is not practical. But we are starting to rely on (e.g., publish, see [McCune, 1998]) complicated proofs from Otter-class provers, so we need a high level of confidence that the proofs are correct.

Fortunately, the resolution/paramodulation parts of Otter-class proofs can be presented explicitly, as *proof objects* that can be easily checked by relatively simple programs. If we write, in ACL2, a proof checker for these proof objects, we can prove, with ACL2, that if the checker accepts a proof object, the proof object is correct. To accomplish this, we define a first-order logic in ACL2 and use it to prove various soundness theorems about the checker.

Unfortunately, Otter-class provers have to preprocess the conjectures they receive (typically by normal-form transformation and Skolemization), and it is impractical to include the preprocessing steps in the proof objects. Therefore, we write the preprocessing functions in ACL2 and prove those functions sound in our logic. This approach leads to a hybrid system we call Ivy,[1] in which part of the proof burden (preprocessing) is on ACL2 functions, and the rest (resolution/paramodulation search) on an Otter-class prover. To drive the proof attempt, we have a top ACL2 function `(proved conjecture)` in which calls to the preprocessing functions, external Otter-class prover, and the checker are embedded in such a way that we can prove the soundness of `proved` without relying on any properties of the Otter-class prover.

Otter-class provers include programs such as MACE [McCune, 1994a, McCune, 1994b] that attempt to disprove conjectures by searching for finite countermodels. A secondary program in Ivy `(disproved conjecture)` incorporates an external disprover in the same way that `(proved conjecture)` incorporates an external prover.

A deficiency of Ivy is that the soundness proofs are with respect to interpretations with finite domains. We chose finite domains because recursion on the domain allows a straightforward evaluation function. However, we believe that Ivy is sound for all domains, because our proof procedures and soundness proofs do not seem to depend on finiteness

[1]Ivy stands for "Ivy verifies your <problem type>."

in important ways. Section 4. contains remarks on generalizing our approach for infinite domains.

We assume familiarity with first-order logic. Some knowledge of resolution style theorem proving [Wos et al., 1992] will be helpful as well. The reader should keep in mind that we are using ACL2 as a metalogic to specify and prove things about a first-order logic. Although the logic of ACL2 is also a first-order logic, it has no direct connection to our defined logic.

Disclaimer. We are not ACL2 experts. This is our first substantial project. Although we believe our definitions and higher-level theorems are nice and reasonable, readers should not look to our ACL2 books for examples of good ACL2 style or economical proofs.

1. BASIC DEFINITIONS

The ACL2 book `"base"` contains the specification of our first-order logic. It contains the core definitions which must be accepted by users in order for the soundness theorems to be meaningful. It includes definitions of well-formed formula, interpretation, and evaluation. A lack of space prevents us from presenting the full specification here, so some functions are omitted and others are described informally.

1.1 TERMS AND FORMULAS

A nonstandard property of our definition of well-formed term is that it includes members of the domains of interpretations as well as standard well-formed terms. The reason for this is that when we evaluate terms in an interpretation, we substitute members of the domain for variables, and we wish to retain well-formedness. The predicates `variable-term` and `domain-term` are defined so that the corresponding sets are disjoint. Aside from `domain-term`, our definitions of terms and formulas are straightforward.

```
(defun wft-list (l)  ;; well-formed list of terms
  (declare (xargs :guard t))
  (if (atom l)
      (null l)
    (and (or (variable-term (car l))
             (domain-term (car l))
             (and (consp (car l))
                  (function-symbol (caar l))
                  (wft-list (cdar l))))
         (wft-list (cdr l)))))
(defmacro wft (x)  ;; well-formed term
  (list 'wft-list (list 'list x)))
```

```
(defun wfatom (a)  ;; well-formed atomic formula
  (declare (xargs :guard t))
  (and (consp a)
       (relation-symbol (car a))
       (wft-list (cdr a))))
```

The connectives for well-formed formulas are `not`, `and`, `or`, `imp`, and `iff`. Conjunctions and disjunctions are binary only (see Exercise 5 for a relaxation of this constraint). The predicates `wfnot`, `wfand`, `wfor`, `wfimp`, and `wfiff` recognize `true-listp`s of the appropriate length whose first member is the corresponding connective.

The quantifiers are `all` and `exists`; the predicates `wfall` and `wfexists` recognize `true-listp`s of length 3 with a quantifier as the first member and a `variable-term` as the second member.

For situations in which the *type* of binary connective or quantifier is irrelevant, the predicates `wfbinary` and `wfquant` are recommended, because they cause fewer cases during proof attempts. These are used, for example, in the following definition of well-formed formula. `A1` and `a2` (meaning argument 1 and argument 2) are simple macros that retrieve the second and third members of a list.

```
(defun wff (f)  ;; well-formed formula
  (declare (xargs :guard t))
  (cond ((equal f 'true) t)
        ((equal f 'false) t)
        ((wfatom f) t)
        ((wfnot f) (wff (a1 f)))
        ((wfbinary f) (and (wff (a1 f)) (wff (a2 f))))
        ((wfquant f) (wff (a2 f)))
        (t nil)))
```

For example, the following expression satisfies `wff`.

```
(all x (iff (p x)
            (exists y (and (q y)
                           (all z (imp (r z)
                                       (or (not (s z (f (a))))
                                           (or (= z (b))
                                               (t x y z)))))))))
```

1.2 INTERPRETATIONS

A first-order interpretation is ordinarily defined as a nonempty domain, a set of (total) functions from the domain to the domain, and a set of relations over the domain. We have an implicit notion of well-formed interpretation, but we do not have a predicate that recognizes well-formed interpretations. If we attempt to evaluate a term or formula in a non-well-formed interpretation, or if the interpretation does not contain a relation or function for a symbol in the formula, a default value (`0` for terms and `nil` for formulas) is returned. If a default value is returned for any well-formed variable-free subterm or well-formed closed subformula, the result of the evaluation is as if the interpretation had been "fixed" into a well-formed interpretation.

A domain is a set of `domain-term`s that contains 0. We have organized it as a binary tree with `domain-term`s as leaves. (See the book `"base"` for notes on this choice.)

```
(defun domainp (dom)    ;; predicate that recognizes domains
  (declare (xargs :guard t))
  (and (domain-term-list (fringe dom))
       (setp (fringe dom))
       (member-equal 0 (fringe dom))))
```

Interpretations are arranged as (*domain* . (*relations* . *functions*)). The following three access functions retrieve the three components of interpretations.

```
(defun domain (i)
  (declare (xargs :guard t))
  (cond ((and (consp i) (domainp (car i))) (car i))
        (t 0)))
(defun relations (i)
  (declare (xargs :guard t))
  (cond ((and (consp i) (consp (cdr i))) (cadr i))
        (t nil)))
(defun functions (i)
  (declare (xargs :guard t))
  (cond ((and (consp i) (consp (cdr i))) (cddr i))
        (t nil)))
```

1.3 EVALUATION

We allow arity-overloading of function symbols and relation symbols. For example the two occurrences of `f` in term `(f (f) x)` are different function symbols, because they have different arities (2 and 0). Therefore, the functions (also relations) in an interpretation are identified by symbol-arity pairs. To look up the value of a function for a tuple (i.e., list) of `domain-term`s, we first find the function in the function list of the interpretation, then find the tuple in the function, and return the value. Because function lists and functions do not necessarily satisfy `alistp`, we have defined `fassoc`, an unguarded version of `assoc-equal`.

```
(defun flookup (fsym tuple i)    ;; function retrieval and application
  (declare (xargs :guard (and (function-symbol fsym)
                              (domain-term-list tuple))))
```

```
  (if (or (not (function-symbol fsym))
          (not (domain-term-list tuple)))
      0   ;; default value: bad arguments
    (let ((sym-func (fassoc (cons fsym (len tuple)) (functions i))))
      (if (not (consp sym-func))
          0   ;; function is not in function list
        (let ((val (fapply (cdr sym-func) tuple)))
          (if (member-equal val (fringe (domain i)))
              val
            0   ;; function value is not in the domain
            ))))))
```

If evaluation is applied to a `domain-term` that is not in the domain of the interpretation, or to a `variable-term`, or to a non-well-formed term, the default value `0` is returned.

```
(defun eval-term-list (l i)
  (declare (xargs :guard (wft-list l)))
  (if (atom l)
      nil
    (cons (cond ((domain-term (car l))
                 (if (member-equal (car l) (fringe (domain i)))
                     (car l)
                   0))   ;; default value
                ((variable-term (car l)) 0)   ;; default value
                ((wf-ap-term-top (car l))
                 (flookup (caar l) (eval-term-list (cdar l) i) i))
                (t 0))    ;; default value
          (eval-term-list (cdr l) i))))
(defmacro eval-term (tm i)
  (list 'car (list 'eval-term-list (list 'list tm) i)))
```

Evaluation of formulas is analogous to term evaluation. `Rlookup` retrieves a relation and applies it to a tuple.

```
(defun rlookup (rsym tuple i)   ;; relation retrieval and application
  (declare (xargs :guard (and (relation-symbol rsym)
                              (domain-term-list tuple))))
  (cond ((not (relation-symbol rsym)) nil)  ;; default value
        ((not (domain-term-list tuple)) nil)  ;; default value
        ((consp (fassoc (cons rsym (len tuple))
                        (relations i)))
         (rapply (cdr (fassoc (cons rsym (len tuple))
                              (relations i))) tuple))
        (t nil)))  ;; default value
```

Evaluation of atomic formulas involves equality. The predicate `wfeq` recognizes `true-listp`s of length 3 with `=` as the first member. An equality atom is true in an interpretation if and only if its two arguments evaluate to the same object.

```
(defun eval-atomic (a i)
  (declare (xargs :guard (wfatom a)))
```

```
  (cond ((or (not (consp a))
             (not (relation-symbol (car a)))
             (not (true-listp (cdr a))) nil))   ;; default value
        ((wfeq a) (equal (eval-term (a1 a) i)
                         (eval-term (a2 a) i)))
        (t (rlookup (car a) (eval-term-list (cdr a) i) i))))
```

Evaluation of nonatomic formulas is done by a pair of mutually recursive functions. `(Feval f i)` recurses through the formula; when it reaches a quantified subformula, it gets a fresh copy of the domain from the interpretation and calls `(feval-d f dom i)`, which recurses through the domain, substituting elements for variables and calling `(feval f i)`.[2]

```
(mutual-recursion
 (defun feval (f i)  ;; recurse through formula
   (declare (xargs :measure (cons (cons (wff-count f) 2) 0)
                   :guard (wff f)))
   (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
   (declare (xargs :measure (cons (cons (wff-count f) 1)
                                  (acl2-count dom))
                   :guard (and (wff f)
                               (wfquant f)
                               (subsetp-equal (fringe dom)
                                              (fringe (domain i))))))
   (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
)   ;; end of mutual recursion
```

Recall that proofs involving induction on mutually recursive functions require special-purpose induction schemes. We use function `(feval-i flg f dom i)` defined in the book `"base"`.

[2]The name `feval` stands for "finite evaluation".

2. THE PROOF PROCEDURE

Our refutation procedure starts with a closed well-formed formula that represents the negation of a conjecture. It consists of the following steps.

1. Convert to negation normal form. This transformation eliminates the connectives `imp` and `iff`, and moves all `not` connectives so that they apply to atomic formulas.
2. Rename bound variables. This transformation renames variables so that each quantifier occurrence has a unique variable.
3. Skolemize. This replaces all existentially quantified variables with terms containing new (Skolem) functions.
4. Move universal quantifiers to the top of the formula.
5. Convert to conjunctive normal form.
6. Search for a refutation by resolution and/or paramodulation.

Steps 1 through 5 are the preprocess phase; after step 5, the formula is a closed universal-prefix conjunctive normal form formula, that is, the universal closure of a conjunction of clauses. Step 6, the hard and interesting part of the procedure, is the search phase.

2.1 PREPROCESSING

Each of steps 1, 2, 4, and 5 produces an equivalent formula, and Skolemization produces an equiconsistent formula, so steps 1 through 5 together produce a formula that is unsatisfiable if and only if the input to step 1 is unsatisfiable.

Steps 1 through 5 are implemented as ACL2 functions, and three types of theorem are proved about the main ACL2 function for each step: syntactic-correctness theorems, preservation-of-property theorems, and soundness theorems. For example, negation normal form translation is done by ACL2 function `(nnf f)`, with guard `(wff f)`. A predicate `(nnfp f)` recognizes formulas in negation normal form, and the syntactic-correctness theorem is

```
(defthm nnf-nnfp
  (implies (wff x)
           (nnfp (nnf x)))).
```

The two preservation-of-property theorems we need for `nnf` are that it preserves well-formedness and closedness:

```
(defthm nnf-wff
```

```
  (implies (wff f)
           (wff (nnf f)))),
(defthm nnf-preserves-free-vars
  (equal (free-vars (nnf f))
         (free-vars f))).
```

The soundness theorem for `nnf` is

```
(defthm nnf-fsound
  (equal (feval (nnf f) i)
         (feval f i))
  :hints ...  ).
```

The ACL2 functions for the steps of the procedure have very restrictive guards, and the syntactic-correctness and preservation-of-property theorems are used extensively in verifying guards for the procedure that composes all of the steps. For example, the guard on `(skolemize f)` is

```
(and (wff f)
     (nnfp f)
     (not (free-vars f))
     (setp (quantified-vars f))).
```

Verification of that guard requires four theorems about the preceding step (variable renaming): preservation-of-property theorems for the first three conditions of the guard, and a syntactic-correctness theorem for the fourth.

The soundness theorem for `(skolemize f)` deserves special mention because Skolemization produces an equiconsistent formula rather than an equivalent formula. Skolemization introduces new function symbols that must be interpreted appropriately when evaluating the Skolemized formula. This task is handled by the function `(skolemize-extend f i)`, which takes a formula and an interpretation and extends the interpretation with functions for the function symbols introduced by `(skolemize f)`. The soundness theorem is

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

By using the soundness theorems for steps 1 through 5, we can show that the preprocess phase produces a formula that has a (finite) model if and only if the original formula has a (finite) model. However, we delay definition of a composite function and its soundness proof so that we can include the search function `refute-n-check` defined in Section 2.2.

By using the preservation-of-property and syntactic-correctness theorems for steps 1 through 5, we can show that the preprocess phase produces a formula with the properties expected by step 6, the search phase. This is proved as part of guard verification for the composite function `refutation-attempt`, defined on page 20.

2.2 SEARCHING

Up to this point, we could be describing a theorem prover implemented entirely in ACL2. But now we take advantage of Otter [McCune, 1994c, McCune and Wos, 1997], a theorem prover coded in another programming language (C). Step 6 is coded as an ACL2 function `(refute-n-check f)` that receives a closed `universal-prefix-cnf` formula. Within `(refute-n-check f)`, formula `f` is annotated, then given to the ordinary (non-ACL2) Common Lisp program `external-prover` that makes operating system calls to create an input file for Otter, run Otter, and extract any refutation from Otter's output. The refutation is returned by `external-prover` to the ACL2 function `refute-n-check`, which calls ACL2 function `check-proof` to check the refutation; if the refutation is approved, it is conjoined to `refute-n-check`'s input and returned; if a refutation is not found, or if the refutation is rejected, `refute-n-check` returns its input unchanged. In any case, the following soundness theorem shows that the output of `refute-n-check` is equivalent to its input.

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

From ACL2's point of view, `external-prover` is a defstub; that is, ACL2 knows it exists, but doesn't know anything else about it. We think of it as a black box. At load time, when preparing to run Ivy, the ACL2 code (including the `external-prover` defstub) is loaded first; then the Common Lisp program `external-prover` is loaded, overriding the defstub.[3] Because we cannot prove any properties of `external-prover`, we use properties of `check-proof` to prove the soundness of `refute-n-check`.

Otter can present its refutations is as *proof objects*, which are detailed line-by-line derivations, in which each line is justified as an application of a rule to preceding lines. The justification for initial steps is named (1) `input`. The other rules are (2) `instantiate`, which applies an explicit substitution to a clause, (3) `resolve`, which applies binary resolution on identical atoms to a pair of clauses, (4) `paramod`, which applies equality substitution on identical terms to a pair of clauses, (5)

[3]According to the ACL2 designers, having an ACL2 function call a Common Lisp function in this way is not officially endorsed, but it is acceptable in this situation.

`flip`, which swaps the arguments of an equality atom of a clause, and (6) `propositional`, which applies a propositional simplification to a clause (in particular, merging identical literals). The justifications for the resolution and paramodulation steps include the positions of the resolved atoms or paramodulated terms. Because instantiation is a separate step, and because resolution and equality substitution operate on identical atoms and terms, no unification is involved in proof objects.

Our ACL2 predicate `wfproof` recognizes well-formed proof objects (ignoring soundness), and the predicate `check-proof` (with guard `wfproof`) recognizes sound proof objects. For example, here is a form that satisfies both `wfproof` and `check-proof`.

```
 ((1 (input) (or (= (b) (a)) (p x)))
(2 (input) (p (a)))
(3 (input) (not (p (b))))
(4 (flip 1 (1)) (or (= (a) (b)) (p x)))
(5 (paramod 4 (1 1) 2 (1)) (or (p (b)) (p x)))
(6 (instantiate 5 ((x . (b)))) (or (p (b)) (p (b))))
(7 (propositional 6) (p (b)))
(8 (resolve 3 () 7 ()) false))
```

The function `check-proof` checks each step by simply applying the rule and checking the result. Excluding steps of type `input`, there are five types of step, and each has a checker. For example, for a `paramod` step, the checker retrieves the two parents from preceding steps, applies equality substitution at the indicated positions of the parents, and checks that the result is equal to the clause in the proof step. Soundness of `proof-check` is proved by proving that the checker for each type of step is sound. For `paramod`, this involves proving that if the universal closures of the two parents are true in some interpretation, and the checker for that step succeeds, then the paramodulant is true in that interpretation. If all steps are approved by the checkers, then the universal closure of the steps in the proof is a consequence of the universal closure of the `input` steps. Function `refute-n-check` also checks that `external-prover` does not modify any `input` steps. This gives us what we need to prove the theorem `refute-n-check-fsound` stated on page 18.

A Detailed Example. Consider the conjecture

```
(imp (and (all x (imp (p x) (q x)))
          (p (a)))
     (exists x (q x))).
```

Preprocessing the negation of the conjecture gives us the following, which is input to `refute-n-check`.

```
(all v1 (all v2 (and (or (not (p v1)) (q v1))
```

```
                    (and (p (a))
                         (not (q v2))))))
```

`Derive` strips off the universal quantifiers and builds the following initial proof object, which is sent to `external-prover`:

```
((1 (input) (or (not (p v1)) (q v1)))
 (2 (input) (p (a)))
 (3 (input) (not (q v2)))).
```

Suppose `external-prover` claims to have found a refutation:[4]

```
  ((1 (input) (or (not (p v1)) (q v1)) nil)
 (2 (input) (p (a)) nil)
 (3 (input) (not (q v2)) nil)
 (4 (instantiate 1 ((v1 . v0))) (or (not (p v0)) (q v0)) (1))
 (5 (instantiate 3 ((v2 . v0))) (not (q v0)) (2))
 (6 (instantiate 2 ()) (p (a)) (3))
 (7 (instantiate 4 ((v0 . (a)))) (or (not (p (a))) (q (a))) nil)
 (8 (resolve 7 (1) 6 ()) (q (a)) (4))
 (9 (instantiate 5 ((v0 . (a)))) (not (q (a))) nil)
 (10 (resolve 9 () 8 ()) false (5))).
```

The function `refute-n-check` calls `check-proof` on the preceding proof object, and it is approved. The clauses are extracted from the proof object and conjoined, and the universal closure is returned by `refute-n-check`:[5]

```
(all v1 (all v2 (all v0 (and (or (not (p v1)) (q v1))
                         (and (p (a))
                          (and (not (q v2))
                           (and (or (not (p v0)) (q v0))
                            (and (not (q v0))
                             (and (p (a))
                              (and (or (not (p (a))) (q (a)))
                               (and (q (a))
                                (and (not (q (a)))
                                 (and false true)))))))))))))).
```

The soundness theorem for `refute-n-check` (page 18) assures us that its output is equivalent to its input. The formula is then given to function `simp-tf`, which simplifies it to `false`.

The Top Procedures and Soundness Theorems. We compose all of the preprocessing steps, `refute-n-check`, and the simplification function into a function `refutation-attempt`, which takes the denial of a conjecture:[6]

---

[4]Proof objects built by Otter frequently contain extraneous instantiation steps. Also, steps in proof objects may contain additional data after the clause.

[5]The `true` at the end is an artifact of building a conjunction from a list. Exercise: fix this.

[6]A deficiency of Otter requires us to right-associate conjunctions and disjunctions.

```
(defun refutation-attempt (f)
  (declare (xargs :guard (and (wff f) (not (free-vars f)))))
  (simp-tf
   (refute-n-check
    (right-assoc
     (cnf
      (pull-quants
       (skolemize
        (rename-all
         (nnf f)))))))))).
```

The soundness theorem for `refutation-attempt` follows easily from the soundness of the components. Note that the soundness theorem for `skolemize` requires that we `skolemize-extend` the interpretation for the initial part of the refutation attempt:

```
(defthm refutation-attempt-fsound
  (equal (feval (refutation-attempt f)
                (skolemize-extend (rename-all (nnf f)) i))
         (feval f i))).
```

A formula is `refuted` if it is closed and well formed, and if `refutation-attempt` gives `false`. We check the guard, because this is a top function.

```
(defun refuted (f)
  (declare (xargs :guard (and (wff f) (not (free-vars f)))))
  (if (and (wff f) (not (free-vars f)))
      (equal (refutation-attempt f) 'false)
      nil))
```

A refuted formula is false in all (finite) interpretations:

```
(defthm refutation-is-fsound
  (implies (refuted f)
           (and (wff f)
                (not (free-vars f))
                (not (feval f i))))
  :hints ...  ).
```

Finally, by fiddling with `not`, we can easily define a proof procedure and prove it sound:

```
(defun proved (f)
  (declare (xargs :guard (and (wff f) (not (free-vars f)))))
  (if (and (wff f) (not (free-vars f)))
      (refuted (list 'not f))
      nil)),
(defthm proof-is-fsound
  (implies (proved f)
           (and (wff f)
                (not (free-vars f))
                (feval f i)))
  :hints ...  ).
```

That is, a proved formula is true in all (finite) interpretations.

3. DISPROVING CONJECTURES

Otter has a complementary companion MACE [McCune, 1994a, McCune, 1994b], which searches for finite models of first-order sentences. If MACE is given the denial of a conjecture, any models found are countermodels to the conjecture. Like Otter, MACE is coded in C, and we call it in the same way we call Otter.

MACE can receive its input as an initial proof object, and it can output models in the form of interpretations that can be given directly to our evaluation function `feval`. The operation of checking MACE's models is simply evaluation with the function `feval`.

`Model-attempt` is analogous to a combination of `refutation-attempt` and `refute-n-check`. `External-modeler` is a `defstub` that is analogous to `external-prover`.

```
(defun model-attempt (f)   ;; return a model of  f  or  nil
  (declare (xargs :guard (and (wff f) (not (free-vars f)))))
  (if (or (not (wff f)) (free-vars f))
      nil
    (let* ((preprocessed (cnf
                          (pull-quants
                           (skolemize (rename-all (nnf f))))))
           (mace-result (external-modeler
                         (assign-ids-to-prf
                          (initial-proof
                           (remove-leading-alls preprocessed))
                          1))))
      (if (feval f mace-result)
          mace-result
        nil))))
```

The soundness theorem for `model-attempt` is trivial, because the property we need to prove is checked by `model-attempt`.

```
(defthm model-attempt-fsound
  (implies (model-attempt f)
           (and (wff f)
                (not (free-vars f))
                (feval f (model-attempt f)))))
```

Or, we can state this positively, for unnegated conjectures:

```
(defun countermodel-attempt (f)
  (declare (xargs :guard (and (wff f) (not (free-vars f)))))
  (cond ((or (not (wff f)) (free-vars f)) nil)
        (t (model-attempt (list 'not f))))),
(defthm countermodel-attempt-fsound
  (implies (countermodel-attempt f)
           (and (wff f)
```

```
                (not (free-vars f))
                (not (feval f (countermodel-attempt f))))))
  :hints ...  ).
```

In other words, if `countermodel-attempt` produces an interpretation for a formula, the formula is false in that interpretation, that is, is not a theorem.

4. INFINITE DOMAINS

Our approach of proving soundness with respect to finite interpretations is certainly questionable. Consider the sentence

```
(imp (all x (all y (imp (= (f x) (f y))
                        (= x y))))
     (all x (exists y (= (f y) x)))),
```

that is, one-to-one functions are onto. It is *not* valid, but it is true for finite domains. Could Ivy claim to have a proof of such a nontheorem?

Any proof, for finite domains, of a such a sentence must use the finiteness hypotheses. But that seems inexpressible in a first-order language, so that the proof would have to be higher order or model theoretic.[7] Ivy works entirely within first-order logic. However, our function `(proved f)` is a "black box" in that the user is not supposed to know anything about it in order to be confident in using Ivy. One could argue that `proved` contains a bug that arises only for infinite interpretations, or that there might be something higher-order lurking there. So, even though we have high confidence that Ivy is sound, we are pursuing a general approach that covers infinite interpretations.

ACL2's encapsulation feature allows it to reason safely about incompletely specified functions. We believe we can use encapsulation to abstract the finiteness.[8] In our current specification, the important way in which finiteness enters the picture is by the definition of `feval-d`, which recurses through the domain. This function, in effect, expands universally quantified formulas into conjunctions and existentially quantified formulas into disjunctions. Instead of `feval-d`, we can consider a constrained function that chooses an element of the domain, if possible, that makes a formula true. When evaluating an existentially quantified formula, we substitute the chosen element for the existentially quantified variable and continue evaluating. (Evaluation of universally quantified variables requires some fiddling with negation.) However, proving the soundness of Skolemization may present complications. If this approach

[7] We are putting aside the argument that set theory and higher-order logics can be encoded in first-order logic.

[8] This approach was suggested by Matt Kaufmann.

succeeds, an interesting (and probably difficult!) exercise would be to try to use ACL2's functional instantiation rule to derive the soundness results for finite interpretations.

Note that the current soundness proofs for `(disproved conjecture)` and `(modeled formula)` are adequate because a finite model is a model.

5. COMPLETENESS

Traditionally, the "interesting" properties of proof procedures are about completeness rather than soundness. Aside from the syntactic correctness of the preprocessing functions, we have no results on completeness. In fact, it is impossible to prove completeness of `(proved conjecture)` unless we can prove completeness of the external Otter-class prover. Even if `(proved conjecture)` is sound, it may have bugs that block proofs, for example in the calling sequence for the external prover. As an extreme example, consider `(defun proved (f) nil)`—it is unquestionably sound, but not very useful. The user must rely on experience with Ivy for evidence that it is complete enough. The computer files that accompany this chapter contain everything needed to run Ivy, including examples, and we invite readers to check it out.

6. EXERCISES

Each exercise has two corresponding files in the `exercise` directory. Numbered startup files contain comments, relevant `include-book` forms, and definitions of related functions. Numbered solution books contain solutions.

1. Define a function to check whether a given variable occurs freely in a formula. Prove that substitution for a variable that does not occur in the formula has no effect.

2. Prove that if an interpretation contains a function *func*, and if a formula does not contain the corresponding function symbol, then evaluation of the formula in the interpretation is independent of the occurrence of *func*. Assume that *func* is the first function in the interpretation.

3. Define a function `(cnf f)` that converts negation normal form formulas (see book `"nnf"`) to conjunctive normal form and a predicate `(cnfp f)` that recognizes conjunctive normal form formulas. Prove that `cnf` (1) preserves the property `wff`, (2) converts `nnfp` formulas to `cnfp`, and (3) is sound.

4. The current proof checker for resolution steps generates all resolvents of the parent clauses and checks whether the clause from the proof object follows from the conjunction of all the resolvents. Define a proof-checking procedure that computes the specified resolvent directly. Prove that the procedure is sound.

5. Conjunctions and disjunctions are binary, which makes it inconvenient to write conjectures with several hypotheses. Define a function to convert a formula with multiple-arity conjunctions and disjunctions to a formula with binary conjunctions and disjunctions. Decide what properties have to be proved to demonstrate that your approach is acceptable, and prove those properties.

6. We rely on the ability to generate a new symbol with respect to a given symbol list in steps 2 and 3 of the search procedure. In variable renaming, step 2, we generate a new variable. In Skolemization, step 3, we generate a Skolem function name. Common Lisp has a function `gensym` , but it is state dependent and therefore not available in ACL2. Define an ACL2 function that generates a symbol that is not in a given list of symbols, and prove its correctness.

References

[McCune, 1994a] McCune, W. (1994a). A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Tech. Report ANL/MCS-TM-194, Argonne National Laboratory, Argonne, IL.

[McCune, 1994b] McCune, W. (1994b). MACE: Models and Counterexamples. URL http://www.mcs.anl.gov/AR/mace/.

[McCune, 1994c] McCune, W. (1994c). Otter 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL. See also URL http://www.mcs.anl.gov/AR/otter/.

[McCune, 1998] McCune, W. (1998). Automatic proofs and counterexamples for some ortholattice identities. *Information Processing Letters*, 65:285–291.

[McCune and Wos, 1997] McCune, W. and Wos, L. (1997). Otter: The CADE-13 Competition incarnations. *J. Automated Reasoning*, 18(2):211–220.

[Wos et al., 1992] Wos, L., Overbeek, R., Lusk, E., and Boyle, J. (1992). *Automated Reasoning: Introduction and Applications,* 2nd edition. McGraw-Hill, New York.