Milawa

an extensible proof checker

Jared Davis

ACL2 Seminar, November 16, 2005

Outline

- The Milawa logic
- A primitive proof checker
- An extended proof checker
- Soundness of the extended checker
- A reflection rule
- Pragmatics of building proofs
- Status and future directions

The Milawa Logic

- Goal: “a large subset” of the ACL2 logic
  - No strings, characters, symbol packages, or complex numbers, maybe not even rationals/negatives
- **Terms** are basically ACL2 expressions
  - Constants, variables, and (recursively) functions applied to other terms.
- **Formulas** are like in the ACL2 book
  - Equalities between terms   t1=t2
  - Negations of formulas   ~A
  - Disjunctions of formulas   A v B

The Milawa Logic: Primitive Rules

Propositional Axiom Schema *~A v A*

Expansion	*Derive B v A from A*
Contraction	*Derive A from A v A*
Associativity	*Derive (A v B) v C from A v (B v C)*
Cut	*Derive B v C from A v B and ~A v C*
Instantiation	*Derive A/σ from A*

The Milawa Logic: Primitive Rules

Reflexivity Axiom

$x = x$

Equality Axiom

$x1 \neq y1 \vee (x2 \neq y2 \vee (x1 \neq x2 \vee y1 = y2))$

Functional Equality Axiom Schema

$x1 \neq y1 \vee (x2 \neq y2 \vee (\ldots \vee (xn \neq yn \vee$

$(f\ x1 \ldots xn) = (f\ y1 \ldots yn)) \ldots))$

Induction Rule (haven't worked this out yet)

Reflection Rule (explained later)

The Milawa Logic: Lisp Axioms

t-not-nil	$t \neq nil$
if-when-nil	$x \neq nil \vee (if\ x\ y\ z) = z$
if-when-not-nil	$x = nil \vee (if\ x\ y\ z) = y$
definition-not	$(not\ x) = (if\ x\ nil\ t)$
definition-implies	$(implies\ x\ y) = (if\ x\ \ldots)$
definition-iff	$(iff\ x\ y) = (if\ x\ \ldots)$
equal-when-diff	$x = y \vee (equal\ x\ y) = nil$
equal-when-same	$x \neq y \vee (equal\ x\ y) = t$
...	...

The Milawa Logic: Formal Proofs

- A **Formal Proof** of a formula *F* in theory *T* is a rooted tree of formulas where:
  - The formula at the root of the tree is *F*
  - The formula at every leaf is a logical axiom or a non-logical axiom of T
  - The formula at every interior node, *n*, can be derived by applying some primitive rule of inference to the formulas of *n*'s children

- Once we have exhibited a formal proof of *F* in *T*, we say that *F* is a theorem of *T*.

A Primitive Proof Checker

- Lisp representation of our terms, and formulas:
  - **termp** is like pseudo-termp
  - **formulap** uses keywords

    (:pequal a b) for $a=b$

    (:pnot A) for $\sim A$

    (:por A B) for *A v B*

- Terms and formulas are distinct
  - Keyword symbols are not valid function symbols

A Primitive Proof Checker

- **Appeals** are our proof objects.
- They have the following structure:

  (method conclusion [subgoals] [extras])

  - **method** explains how the formula is justified
  - **conclusion** is a formula which this appeal asserts
  - **subgoals** is a list of appeals which justify the conclusion, if needed by this method
  - **extras** holds any additional information, e.g., substitution lists, if needed by this method

A Primitive Proof Checker

- We write functions to check each type of appeal.
- Note: only a local check – “assume subappeals”

```
(defun contraction-okp (x database arity-table)
  (declare (ignore database arity-table))
  (let ((method     (get-method x))
        (conclusion (get-conclusion x))
        (subgoals   (get-subgoals x))
        (extras     (get-extras x)))
    (and (equal method :contraction)
         (equal extras nil)
         (equal (len subgoals) 1)
         (let* ((subgoal (first subgoals))
                (subconc (get-conclusion subgoal)))
           (and (equal (first subconc) :por)
                (equal (second subconc) conclusion)
                (equal (third subconc) conclusion))))))
```

A Primitive Proof Checker

- We can then locally check any type of appeal by combining the checkers in the natural way:
- This basically just emulates a virtual function call in an inheritance hierarchy

```
(defun appeal-provisionally-okp (x database arity-table)
  (case (get-method x)
    (:axiom                   (axiom-okp                x database arity-table))
    (:propositional-schema (propositional-schema-okp x database arity-table))
    (:functional-equality  (functional-equality-okp  x database arity-table))
    (:expansion               (expansion-okp            x database arity-table))
    (:contraction             (contraction-okp          x database arity-table))
    (:associativity           (associativity-okp        x database arity-table))
    (:cut                     (cut-okp                  x database arity-table))
    (:instantiation           (instantiation-okp        x database arity-table))
    (:induction               (induction-okp            x database arity-table))
    (:reflection              (reflection-okp           x database arity-table))
    (otherwise                nil)))
```

A Primitive Proof Checker

- The full proof checker itself just extends this local check everywhere throughout the tree

```
(mutual-recursion

 (defun proofp (x database arity-table)
   (and (appealp x arity-table)
        (appeal-provisionally-okp x database arity-table)
        (proof-listp (get-subgoals x) database arity-table)))

 (defun proof-listp (xs database arity-table)
   (if (consp xs)
       (and (proofp (car xs) database arity-table)
            (proof-listp (cdr xs) database arity-table))
     (equal xs nil))))
```

An Extended Proof Checker

- Commute Or *Derive B v A from A v B*

```
(defun commute-or-okp (x database arity-table)
  (declare (ignore database arity-table))
  (let ((method     (get-method x))
        (conclusion (get-conclusion x))
        (subgoals   (get-subgoals x))
        (extras     (get-extras x)))
    (and (equal method :commute-or)
         (equal extras nil)
         (equal (len subgoals) 1)
         (let* ((subgoal (first subgoals))
                (subconc (get-conclusion subgoal)))
           (and (equal (first subconc) :por)
                (equal (first conclusion) :por)
                (equal (second conclusion) (third subconc))
                (equal (third conclusion) (second subconc)))))))
```

An Extended Proof Checker

- We add this rule to create *proofp-2*

```
(defund appeal-provisionally-okp-2 (x database arity-table)
  (case (get-method x)
    (:commute-or (commute-or-okp x database arity-table))
    (otherwise   (appeal-provisionally-okp x database
                                            arity-table))))

(mutual-recursion

 (defund proofp-2 (x database arity-table)
   (and (appealp x arity-table)
        (appeal-provisionally-okp-2 x database arity-table)
        (proof-listp-2 (get-subgoals x) database arity-table)))

 (defund proof-listp-2 (xs database arity-table)
   (if (consp xs)
       (and (proofp-2 (car xs) database arity-table)
            (proof-listp-2 (cdr xs) database arity-table))
     (equal xs nil))))
```

The Extended Checker is Sound

- We say a formula *F* is **provable** when there exists a formal proof of *F*.

```
(defun-sk provablep (formula database arity-table)
  (exists proof
    (and (proofp proof database arity-table)
         (equal (get-conclusion proof) formula))))
```

- We will show that whenever *proofp-2* accepts an appeal *X*, then the conclusion of *X* is provable.
  - Consequence: if *proofp* is sound, then so is *proofp-2*.

The Extended Checker is Sound

- The following lemma is not too difficult to prove:

```
(defthm soundness-of-appeal-provisionally-okp
  (implies (and (appealp x arity-table)
                (appeal-provisionally-okp x database arity-table)
                (provable-listp (strip-conclusions (get-subgoals x))
                                database arity-table))
           (provablep (get-conclusion x) database arity-table)))
```

- With that in place, we mainly just need:

```
(defthm soundness-of-commute-or-okp
  (implies (and (appealp x arity-table)
                (commute-or-okp x database arity-table)
                (provable-listp (strip-conclusions (get-subgoals x))
                                database arity-table))
           (provablep (get-conclusion x) database arity-table)))
```

The Extended Checker is Sound

- Derivation of Commute Or

```
1. A v B      Given
2. ~A v A     Propositional Axiom
3. B v A      Cut; 1,2
```

- *Magic compiler* based on this derivation

```
(defun magic-compiler (x database arity-table)
  (let* ((or-a-b        (get-conclusion (first (get-subgoals x))))
         (or-a-b-proof (provablep-witness or-a-b database
                                          arity-table))
         (a             (second or-a-b)))
    (cut or-a-b-proof
         (propositional-schema a))))
```

The Extended Checker is Sound

```
(defthm get-conclusion-of-magic-compiler
  (implies (and (appealp x arity-table)
                (commute-or-okp x database arity-table)
                (provable-listp (strip-conclusions (get-subgoals x))
                                database arity-table))
           (equal (get-conclusion
                     (magic-compiler x database arity-table))
                  (get-conclusion x))))

(defthm proofp-of-magic-compiler
  (implies (and (appealp x arity-table)
                (commute-or-okp x database arity-table)
                (provable-listp (strip-conclusions (get-subgoals x))
                                database arity-table))
           (proofp (magic-compiler x database arity-table)
                   database arity-table)))

(defthm soundness-of-commute-or-okp
  (implies (and (appealp x arity-table)
                (commute-or-okp x database arity-table)
                (provable-listp (strip-conclusions (get-subgoals x))
                                database arity-table))
           (provablep (get-conclusion x) database arity-table)))
```

The Extended Checker is Sound

```
(defthm soundness-of-appeal-provisionally-okp-2
  (implies (and (appealp x arity-table)
                (appeal-provisionally-okp-2 x database arity-table)
                (provable-listp (strip-conclusions (get-subgoals x))
                                database arity-table))
           (provablep (get-conclusion x) database arity-table)))

(defthm crux
  (if (equal flag :proof)
      (implies (proofp-2 x database arity-table)
               (provablep (get-conclusion x) database arity-table))
    (implies (proof-listp-2 x database arity-table)
             (provable-listp (strip-conclusions x) database
                             arity-table))))

(defthm proofp-2-is-sound
   (implies (proofp-2 x database arity-table)
            (provablep (get-conclusion x) database arity-table)))
```

The Extended Checker is Sound

- So we have an ACL2 proof that proofp-2 is sound with respect to proofp.
  - But this is not "formal" in the sense of *proofp*
- Goal: translate this into a proofp-checkable proof.
  - The ACL2 proof is a "roadmap" of useful lemmas to prove.
  - Now we just need to be able to construct these proofs. (more on this soon)

Adding a Reflection Rule

- Assume we have a proofp-checkable proof that proofp-2-is-sound.
- Assume we have used *proofp-2* to "prove" *F*.
- How do we get a formal *proofp* proof of *F*?
  - We could skip this, claim that proofp-2-is-sound is convincing enough
  - We could try to "compile" the proof
    - It might be too large to check
  - We could add a reflection rule

Adding a Reflection Rule

- The reflection rule will be something like this:

  Derive F from (provablep F …) = t

- Now, if we know proofp-2 proves F, we can:
  - Show that F is provable, by appealing to the lemma:

```
(defthm proofp-2-is-sound
   (implies (proofp-2 x database arity-table)
            (provablep (get-conclusion x) database arity-table)))
```

  - Use reflection to conclude that F is true, since it is provable

Pragmatics of Building Proofs

- Formal proofs are too big to create by hand, so I write functions to build them for me.
- These are like derived rules of inference

```
(defun commute-or-bldr (x)
  ;; Derive b v a from a proof of a v b.
  ;; Derivation.
  ;; 1. a v b       Given
  ;; 2. ~a v a      Propositional Axiom
  ;; 3. b v a       Cut; 1, 2
  (or (and (appeal-structureishp x)
           (let* ((or-a-b (get-conclusion-fast x))
                  (a      (second or-a-b)))
             (and (equal (first or-a-b) :por)
                  (cut x (propositional-schema a)))))
      (cw "[commute-or-bldr]: invalid argument: ~%~x0~%" x)))
```

```
(defun right-expansion-bldr (x b)
  ;; Derive (a v b) from a proof of a
  ;; Derivation.
  ;; 1. a           Given
  ;; 2. b v a       Expansion; 1
  ;; 3. a v b       Commute Or; 2
  (or (and (appeal-structureishp x)
           (formula-structurep b)
           (commute-or-bldr (expansion b x)))
      (cw "[right-expansion-bldr]: invalid args: ~%~x0~%~x1~%" x b)))

(defun modus-ponens-bldr (x y)
  ;; Derive b from proofs of a and ~a v b.
  ;; Derivation.
  ;; 1. a          Given
  ;; 2. a v b      Right Expansion; 1
  ;; 3. ~a v b     Given
  ;; 4. b v b      Cut; 2, 3
  ;; 5. b          Contraction; 4
  (or (and (appeal-structureishp x)
           (appeal-structureishp y)
           (let* ((a           (get-conclusion-fast x))
                  (or-not-a-b (get-conclusion-fast y))
                  (not-a      (second or-not-a-b))
                  (b          (third or-not-a-b)))
             (and (equal (second not-a) a)
                  (contraction
                   (cut (right-expansion-bldr x b)
                        y))))))
      (cw "[modus-ponens-bldr]: invalid args:~%~x0~%~x1~%" x y)))
```

```
;; Derive a v (c v b) from a proof of a v b
;; Derive a v (b v c) from a proof of a v b
;; Derive a v b from a v (b v b)
;; Derive a v (b v c) from (a v b) v c
;; Derive ~(a v b) v c from ~a v c and ~b v c
;; Schema: ~(a v b) v (b v a)
;; Derive a v (c v b) from a proof of a v (b v c)
;; Schema: ~(a v d) v ((a v b) v (c v d))
;; Schema: ~(b v c) v ((a v b) v (c v d))
;; Derive (a v b) v (c v d) from a proof of (a v d) v (b v c)
;; Derive a v (b v (c v d)) from a proof of a v ((b v c) v d)
;; Derive a v ((b v c) v d) from a proof of a v (b v (c v d))
;; Derive a v (c v d) from proofs of a v (b v c) and a v (~b v d)
;; Derive p v b from proofs of p v a and p v (~a v b)
;; Derive b from proofs of ~a and (a v b)
;; Derive P v b from proofs of P v ~a and P v (a v b)
;; Schema: a = a
;; Schema: a1 != b1 v (a2 != b2 v (a1 != a2 v b1 = b2))
;; Derive b = a from a = b
;; Schema: a != b v b = a
;; Derive b != a from a != b
;; Schema: ~(p v a = b) v (p v b = a)
;; Derive P v b = a from a proof of P v a = b
;; Derive a = c from a = b and b = c
;; Derive P v a = c from proofs of P v a = b and P v b = c
;; Derive c != b from proofs of a != b and c = a
;; Derive P v c != b from proofs of P v a != b and P v c = a
;; Derive b from a1, a2, ..., an, ~a1 v (~a2 v ... v (~an v b) ... )
;; Derive (f t1 ... tn) = (f s1 ... sn) from t1 = s1, ..., tn = sn.
;; Derive P v b from P v a1, ..., P v an, P v (~a1 v (... v (~an v b)))
;; Derive P v (f t1 ... tn) = (f s1 ... sn) from P v t1 = s1, ... P v tn = sn
;; Derive a from proofs of b v a and ~b v a
```


R

Some Important Rules

- Transitivity of Equal Builders
  - *Derive a = c from a = b and b = c*
  - *Derive P v a = c from P v a = b and P v b = c*
- Equal by Arguments Builders
  - *Derive (f t1 … tn) = (f s1 … sn)*

    from t1 = s1, …, tn = sn
  - *Derive P v (f t1 … tn) = (f s1 … sn)*

    from P v t1 = s1, …, P v tn = sn

SR, A Simple Rewriter

- I have a rewriter that can build some proofs
  - sr : term * rule list → proof

    Where a “rule” is a simple formula of the form lhs = rhs
  - (sr x rules) creates a proof of $x = x'$, if any rules can rewrite parts of x
- Basically unconditional inside-out rewriting with proof output
  - The equal-by-args and transitivity-of-equal builders construct the proof

Some Example Rules

- These are provable using our builders and the Lisp axioms

```
(if nil y z) = z
(if t y z) = y
(if x y y) = y
(if x (if x y w) z) = (if x y z)
(if x y (if x y z)) = (if x y z)
(if (if x y z) p q) = (if x (if y p q) (if z p q))
```

- With these (and definitions of *implies*, *not*), *sr* can prove the following is just t:

```
(IMPLIES (NOT (CONSP X))
         (NOT (IF (CONSP X)
                  (IF (EQUAL A (CAR X))
                      T
                    (MEMBERP A (CDR X)))
                NIL)))
```

Space and Time Considerations

- *(implies (not (consp x)) (not (memberp a x)))* = *t*
  - About 475 KB, 6200 lines when printed with ~f
  - About ½ second to check (excluding read time)
- *(if (if x y z) p q)* = *(if x (if y p q) (if z p q))*
  - About 225 KB, 3000 lines
- *(booleanp t)* = *t*
  - About 22 KB, 280 lines
- *(booleanp (equal x y))* = *t*
  - About 1MB, 13000 lines

Current Status

- Currently capabilities
  - Manipulate propositional formulas fairly easily
  - Unconditional rewriting of terms
  - Simple non-inductive theorems
- Short term goals
  - Developing conditional rewriter
  - Figure out induction rule, number representation
  - Well defined extension principle for new definitions
  - Actually begin proving lemmas on the way to proofp-2-is-sound

Future Directions (Long Term)

- Prove proofp-2-is-sound using proofp
- Develop useful extensions and verify them, to create more powerful proof checkers
- Perhaps consider ACL2 integration?
  - Local events, missing datatypes, etc.
  - Extending ACL2 to emit checkable proof objects?
  - Allowing ACL2 to accept checked proof objects?

Thanks

- Useful Papers and Books
  - Computer Aided Reasoning: An Approach, Chapter 6
  - A Precise Description of the ACL2 Logic
  - Structured Theory Development for a Mechanized Logic
  - A Quick and Dirty Sketch of a Toy Logic
  - Mathematical Logic, Shoenfield
  - Metatheory and Reflection, John Harrison