The Role of Human Creativity in Mechanized Verification

J Strother Moore
Department of Computer Science
University of Texas at Austin

John McCarthy(Sep 4, 1927 – Oct 23, 2011)

Contributions

Lisp, mathematical semantics for programming languages, "Artificial Intelligence," garbage collection, if-then-else, circumscription for non-monotonic logic, . . .

In order for a program to be capable of learning something it must first be capable of being told it.

— *John McCarthy, "Programs with Common Sense" (aka "The Advice Taker"), 1959*

Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program.

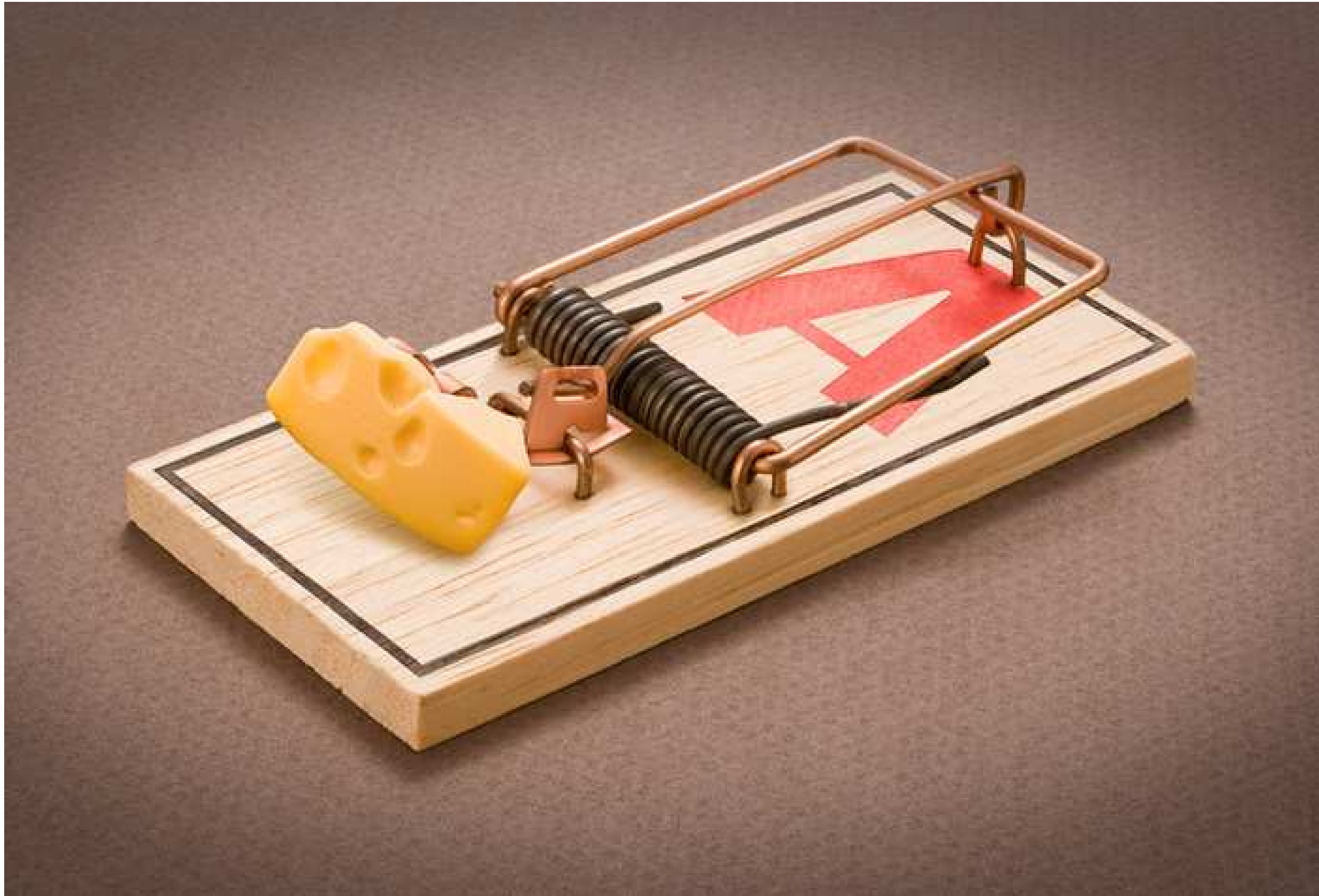
— *John McCarthy, "A Basis for a Mathematical Theory of Computation," 1961*

The meaning of a program is defined by its effect on the state vector.

– *John McCarthy, "Towards a Mathematical Science of Computation," 1962*

If you'd given this talk in 1981, I would have said 'What took so long?'

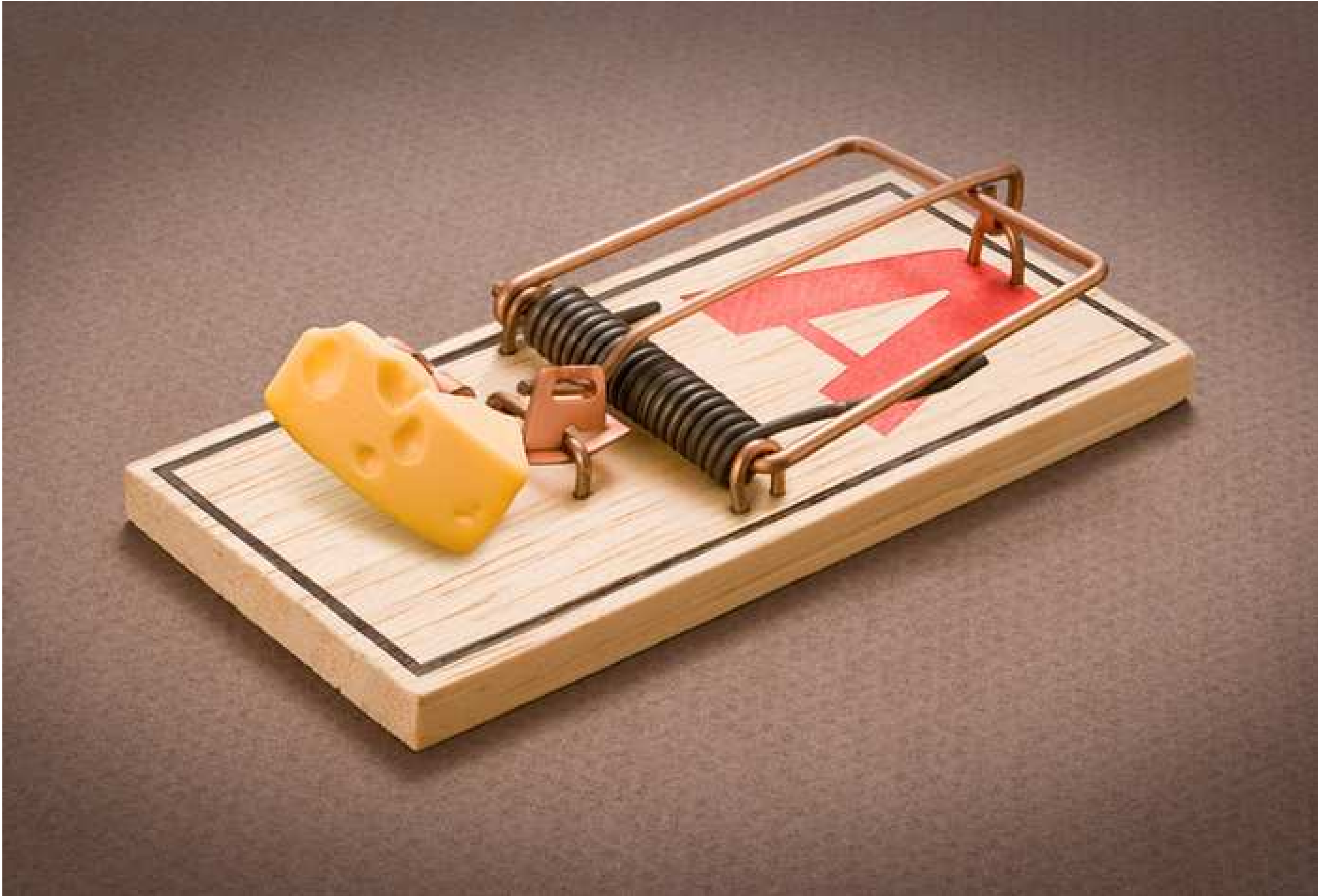
— *John McCarthy, after a talk by J Moore on applications of ACL2 in the mid-1990s*



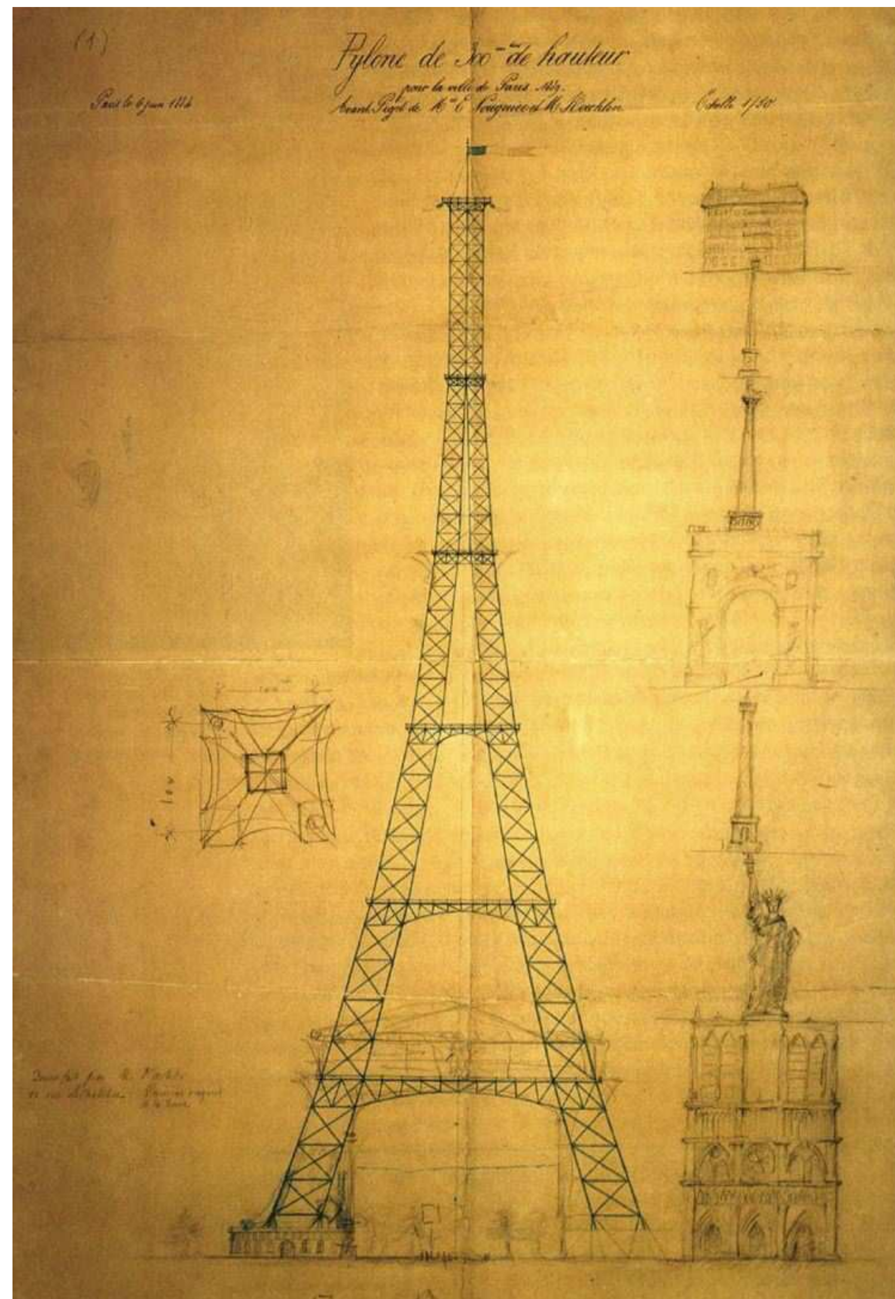
Delusion Mouse Trap (1876)

Royal Number 1 Trap (1879)

Hotchkiss 5-hole Choker (1890?)



FreeFoto.com


(1)
Pylone de 300m de hauteur

Mathematicians Do It Too

Virtually every textbook proof has been cleaned up, sometimes to the point where the original proof (or even the original theorem) is completely absent.

Probably every theorem of analysis proved in the 17th and 18th centuries was proved again more cleanly and rigorously in the 19th century using the “epsilon-delta” approach.

“The original proof of CRT [the Church–Rosser theorem] was fairly long and very complicated. . . . Newman generalized the universe of discourse . . . . He proved a result similar to CRT by topological arguments. Curry . . . generalized the Newman result . . . .

Unfortunately, it turned out that neither the Newman result nor the Curry generalization entailed CRT. . . . This was discovered by Schroer . . . . Schroer derived still further generalizations of the Newman and Curry results, which indeed do entail CRT. . . . Schroer 1965 is 627 typed pages . . . .

Chapter 4 of Curry and Feys 1958 is devoted to a proof of CRT for λ-calculus and . . . is not recommended for light reading. . . . Meanwhile a genuine simplification of the proof of CRT had come in sight. See Martin-Löf 1972.

It is agreed that Martin-Löf got some of his ideas from lectures by Tait. An exposition of the proof of CRT according to Tait and Martin-Löf appears in Appendix I of Hindley, Lercher and Seldin 1972." – *J.B. Rosser*

It is (apparently) in our natures to polish our work to make it more beautiful, elegant, and understandable.

It is (apparently) in our natures to polish our work to make it more beautiful, elegant, and understandable.

This is great if your only concern is the beauty/elegance/clarity of the final product.

It is (apparently) in our natures to polish our work to make it more beautiful, elegant, and understandable.

This is great if your only concern is the beauty/elegance/clarity of the final product.

But it is *harmful* in our business!

Our Business

Formal methods research is not about proving hardware and software correct.

Formal methods research is about *mechanizing creativity*.

By polishing our results we *obscure* the problems we're really trying to solve.

A Trivial Example from My Class

- `(endp` x`)` — determines if x is empty
- `(car` x`)` — first element of x (when x is non-empty)
- `(cdr` x`)` — rest of x (when x is non-empty)

- `(member` e x`)` — determines whether e occurs as an element of list x
- `(rm!` e x`)` — deletes every occurrence of e as a element from x

A Student's Definition

```
(defun set-equal (x y)
 (if (endp x)
     (endp y)
     (and (member (car x) y)
          (set-equal (rm! (car x) x)
                     (rm! (car x) y)))))
```

This function determines whether x and y have the same elements, ignoring order and duplication.

The Student's Goal Theorem

```
(set-equal (append a a) a)
```

The Student's Goal Theorem

```
(set-equal (append a a) a)
```

```
(defun append (x y)
 (if (endp x)
     y
     (cons (car x)
           (append (cdr x) y))))
```

The Student's Goal Theorem

```
(set-equal (append a a) a)
```

Axiom

```
(append x y)
=
(if (endp x)
    y
    (cons (car x)
          (append (cdr x) y)))
```

The Student's Goal Theorem

```
(set-equal (append a a) a)
```

Axiom *Instance*

```
(append a a)
=
(if (endp a)
    a
    (cons (car a)
          (append (cdr a) a)))
```

We tackled this interactively in class.

Here is our more general theorem:

```
(defthm crux
 (implies (subset b a)
          (set-equal (append a b) a)))

(defthm goal
 (set-equal (append a a) a))
```

The Definition of `Subset`

```
(defun subset (x y)
 (if (endp x)
     t
     (and (member (car x) y)
          (subset (cdr x) y))))
```

In class we proved several beautiful and helpful lemmas, e.g.,

```
(rm! e (append a b))
 =
(append (rm! e a) (rm! e b))
```

But with no time remaining in class our *still unproved* `crux` looked like this:

```
(defthm crux
 (implies (subset b a)
          (set-equal (append a b) a))
 :hints
 (("Goal" :induct (set-equal a b))
  ("Subgoal *1/2''"
   :use (:instance subset-rm!
                   (x b)
                   (y a)
                   (e (car a))))
  ("Subgoal *1/3'"
   :expand ((set-equal (append a b) a)))))
```

```
(defthm crux
 (implies (subset b a)
          (set-equal (append a b) a))
 :hints
 (("Goal" :induct (set-equal a b))
  ("Subgoal *1/2'''"
   :use (:instance subset-rm!
                   (x b)
                   (y a)
                   (e (car a))))
  ("Subgoal *1/3'"
   :expand ((set-equal (append a b) a)))))
```

Class ended.

I went home. I ate, watched TV, read, showered, slept.

I woke up with the alarm and *knew* I should change my approach in two ways.

Insight 1: Redefine `subset`

```
(defun subset (x y)
 (if (endp x)
     t
     (and (member (car x) y)
          (subset (cdr x)
                  y))))
```

Insight 1: Redefine `subset`

```
(defun subset (x y)
  (if (endp x)
      t
      (and (member (car x) y)
           (subset (cdr x)
                   y))))
```

Insight 1: Redefine `subset`

```
(defun subset (x y)
 (if (endp x)
     t
     (and (member (car x) y)
          (subset (rm! (car x) x)
                  (rm! (car x) y))))))
```

This is Fair

It does not change the goal theorem.

The definitional principle is conservative.

`Subset` is not mentioned in the final theorem.

So how it is defined doesn't matter – except to the proof.

The Proof Plan

```
(defthm crux
 (implies (subset b a)
          (set-equal (append a b) a)))

(defthm goal
 (set-equal (append a a) a))
```

Redefining Subset is a Good Move

```
(defun subset (x y)
 (if (endp x)
     t
     (and (member (car x) y)
          (subset (rm! (car x) x)
                  (rm! (car x) y))))))

(defun set-equal (x y)
 (if (endp x)
     (endp y)
     (and (member (car x) y)
          (set-equal (rm! (car x) x)
                     (rm! (car x) y))))))
```

Insight 2: Re-state `crux`

```
(defthm crux                          ; Old
  (implies (subset b a)
           (set-equal (append a b) a)))
```

Insight 2: Re-state `crux`

```
(defthm crux                          ; Old
  (implies (subset b a)
           (set-equal (append a b) a)))
```

Insight 2: Re-state `crux`

```
(defthm crux                               ; New
  (implies (subset b a)
           (set-equal (append b a) a)))
```

The Proof Plan Still "Works"

```
(defthm crux                        ; New
 (implies (subset b a)
          (set-equal (append b a) a)))

(defthm goal
 (set-equal (append a a) a))
```

But the New Crux is Easier to Prove

```
(defthm crux                         ; Old
 (implies (subset b a)
          (set-equal (append a b) a)))

(defthm crux                         ; New
 (implies (subset b a)
          (set-equal (append b a) a)))
```

About Induction

To prove $\phi(x, y)$ by induction on x:

Base:

$(\texttt{endp}\ x) \rightarrow \phi(x, y)$

Induction Step:

$(\neg(\texttt{endp}\ x) \wedge \phi(x', y')) \rightarrow \phi(x, y)$

where x' is "shorter than" x.

About Induction

To prove $\phi(x, y)$ by induction on x:

Base:

$(\texttt{endp}\ x) \ \rightarrow \ \phi(x, y)$

Induction Step:

$(\neg(\texttt{endp}\ x) \ \wedge \ \phi(x', y')) \ \rightarrow \ \phi(x, y)$

where x' is "shorter than" x.

About Induction

To prove $\phi(x, y)$ by induction on x:

Base:

$\texttt{(endp } x\texttt{)} \;\rightarrow\; \phi(x, y)$

Induction Step:

$(\neg\texttt{(endp } x\texttt{)} \;\wedge\; \phi(x', y')) \;\rightarrow\; \phi(x', y')$

where x' is “shorter than” x.

About Induction

To prove $\phi(x, y)$ by induction on x:

Base:

$(\texttt{endp}\ x) \ \rightarrow \ \phi(x, y)$

Induction Step:

$(\neg(\texttt{endp}\ x) \ \wedge \ \phi(x', y')) \ \rightarrow \ \phi(x', y')$

where x' is “shorter than” x.

So the key to proving $\phi(x, y)$ by induction is finding a ϕ with the property that it can be *rewritten to an instance of itself*.

Rewrite to an Instance?

```
(defthm crux                          ; Old
 (implies (subset b a)
          (set-equal (append a b) a)))

(defthm crux                          ; New
 (implies (subset b a)
          (set-equal (append b a) a)))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset b a)
         (set-equal (append a b) a))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset b
                 a)
         (set-equal
           (append a
                   b)
           a))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset b
                 a)
         (set-equal
           (append a
                   b)
           a))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append a
                   b)
           a))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append a
                   b)
           a))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (rm! (car a) (append a
                                b))
           (rm! (car a) a)))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car a) a)
                   (rm! (car a) b))
           (rm! (car a) a)))
```

The Old Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car a) a)
                   (rm! (car a) b))
           (rm! (car a) a)))
```

The Old Crux:

```
(implies (subset b
                 a)
         (set-equal
           (append a
                   b)
           a))
```

The Old Rewritten Crux: Not an Instance!

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car a) a)
                   (rm! (car a) b))
           (rm! (car a) a)))
```

The Old Rewritten Crux: Not an Instance!

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car a) a)
                   (rm! (car a) b))
           (rm! (car a) a)))
```

The Old Rewritten Crux: Not an Instance!

```
(implies (subset (rm! (CAR B) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car a) a)
                   (rm! (CAR A) b))
           (rm! (car a) a)))
```

The Old Crux...

is hard to prove by induction because some of its subterms remove `(car b)` but others remove `(car a)`, so we need "inconsistent instantiations", sometimes replacing `b` by one term, `(rm! (car b) b)`, and sometimes by another, `(rm! (car a) b)`.

The New Crux: Rewrite to an Instance?

```
(implies (subset b a)
         (set-equal (append b a) a))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset b
                 a)
         (set-equal
           (append b
                   a)
           a))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset b
                 a)
         (set-equal
           (append b
                   a)
           a))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append b
                   a)
           a))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append b
                   a)
           a))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (rm! (car b) (append b
                                  a))
           (rm! (car b) a)))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car b) b)
                   (rm! (car b) a))
           (rm! (car b) a)))
```

The New Crux: Rewrite to an Instance?

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car b) b)
                   (rm! (car b) a))
           (rm! (car b) a)))
```

The New Crux

```
(implies (subset b
                 a)
         (set-equal
           (append b
                   a)
           a))
```

The New Rewritten Crux: an Instance!

```
(implies (subset (rm! (car b) b)
                 (rm! (car b) a))
         (set-equal
           (append (rm! (car b) b)
                   (rm! (car b) a))
           (rm! (car b) a)))
```

The New Crux

The improved formulation is easy to prove because we remove `(car b)` uniformly from `b` and from `a` everywhere.

So after breakfast, I typed in the new formulation of `subset` and `crux` and the proof was done.

Then, while driving to campus...

Insight 3: No Generalization Needed

Using the rules developed for the proof above, we can prove

```
(defthm goal
  (set-equal (append a a) a))
```

directly by induction on a by `(rm! (car a) a)`. There is no need for `subset` or `crux`!

A Tale of Two Papers

Which is the better paper to write?

An Automatic Proof of Goal

or

How Not to Prove Goal, and Why

Which paper might lead somebody to breakthrough research?

Other Examples

- How do you model the system in question? Should you include the behavior of resource x in your model? Why not?
- What is the right specification?

- How do you define the concepts used in the specification? What “goes wrong” if you adopt some equally obvious alternative?

- What “obvious” variable orderings did you try before the one that worked? Why were they “wrong?”

- What “obvious” canonical forms did you adopt before finding the ones that worked? Why were they “wrong?”

- What modeling/testing/proof debugging tools did you use?

By highlighting such issues we facilitate automation.

Summary

Our customers rightly want to see the elegant solution.

But we should be showing each other the failures and false starts.