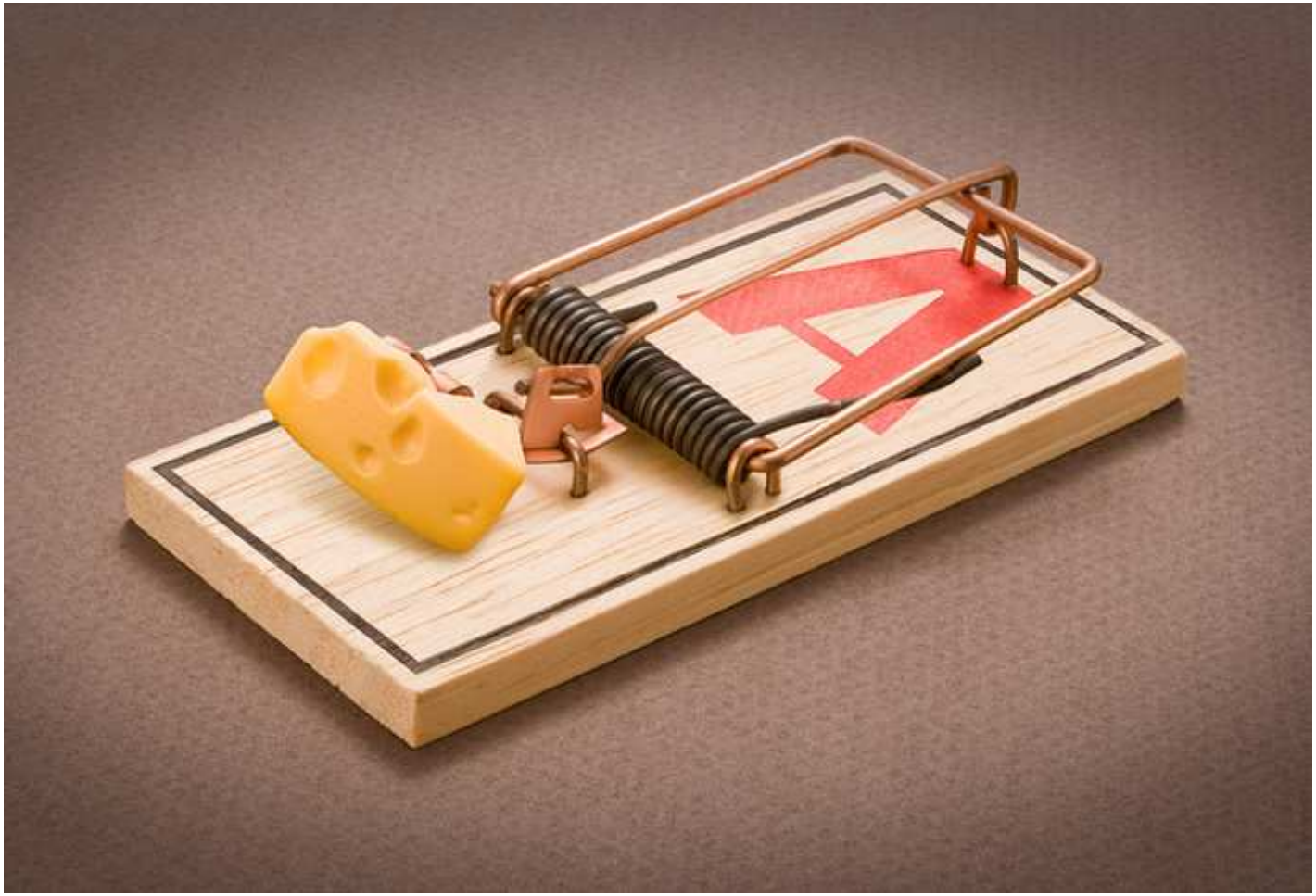
The Role of Human Creativity in Mechanized Verification

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



Delusion Mouse Trap (1876)

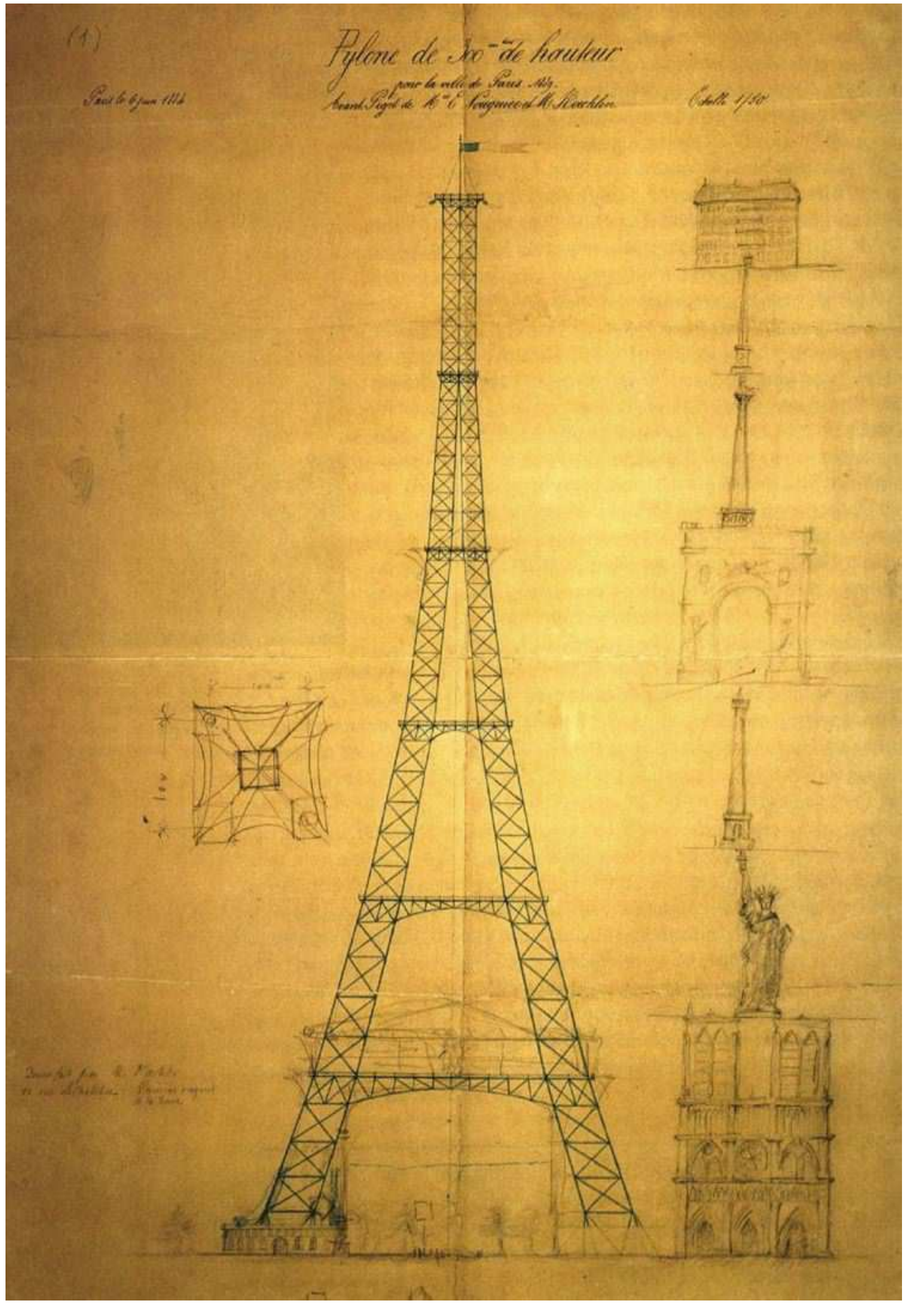
Royal Number 1 Trap (1879)

Hotchkiss 5-hole Choker (1890?)

US Patent Office has issued 4,400 patents for mousetraps since opening in 1838.

The Patent Office has 39 official categories of mousetraps, including choking, squeezing, impaling, trapping, killer bar, explosive, shock, etc.

(Aside: If you're into this sort of thing, see the *Bunny Suicides* books by Andy Riley.)


(1)
Pylone de 300m de hauteur
pour la ville de Paris 1889
Avant Projet de MM. E. Nouguier et M. Koechlin
Paris le 6 juin 1884
Echelle 1/50

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 17^{th} and 18^{th} centuries was proved again more cleanly and rigorously in the 19^{th} 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)
```

"Inductive proofs require general theorems. Many theorems you'll want to prove are actually too specific to admit inductive proofs." — J Moore

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.

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:

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

Induction Step:

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

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 something involving a "smaller" instance of itself*.

So, our story resumes...

Class ended.

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

I woke up at the usual time and *knew* I should change the class' 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)))))
```

Two Questions

(a) Is it fair to redefine `subset`? After all, it means we're not trying to prove the same `crux` anymore!

(b) Why might redefining `subset` help?

(a) Yes, It is Fair!

`Crux` is not the goal.

`Subset` is not involved in the goal.

The definitional principle is conservative.

So how `subset` 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))
```

(b) Redefining Subset Helps because...

```
(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)))))
```

Both remove elements of x from y.

Insight 2: Re-state `crux`

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

Note: The hypothesis is removing elements of `b` from `a`, but the conclusion is removing elements of `a` from `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)))
```

Note: Both the hypothesis and the conclusion are removing elements of `b` from `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)))
```

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 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

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

Insight 3: No Generalization Needed

```
(defthm goal
 (set-equal (rm! (car a) (append a
                                 a))
            (rm! (car a) a)))
```

Insight 3: No Generalization Needed

```
(defthm goal
 (set-equal (append (rm! (car a) a)
                    (rm! (car a) a))
            (rm! (car a) a)))
```

Insight 3: No Generalization Needed

Using the rules already developed, 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 Good Story Ruined

Before proceeding I should say that much of the thought process just described is codified by

Rippling: a heuristic for guiding inductive proofs, Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill, *AI Journal*, **62**, pp. 188–253, 1993.

A Tale of Two Papers

Which is the better paper to write? Which might get published?

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 industrial customers just care about finding a proof, any proof.

Our research funders want to see published papers.

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