Proving Theorems about Commercial Microprocessors
Recent Results with ACL2

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Office: TAY 4.140A
Email: moore@cs.utexas.edu

http://www.cs.utexas.edu

Abstract

ACL2 is an interactive mechanical theorem prover designed for use in modeling hardware and software and proving properties about those models. ACL2 stands for "A Computational Logic for Applicative Common Lisp."

It has been used to prove theorems about a variety of algorithms and designs, including floating-point division and square root on both the AMD K5 (where the operations are done in microcode) and K7 (where they are done using different algorithms in hardware). ACL2 has also been used in microprocessor modeling, including the Motorola CAP DSP chip and the Rockwell- Collins JEM1 Java Virtual Machine microprocessor.

In this talk I discuss ACL2 and some of its recent applications.

Outline of This Talk

- The Logic
- The System
- Some Applications
  - Floating Point
  - Microprocessors
  - Year 2000 Compliance
  - ACL2

ACL2: A Computational Logic for Applicative Common Lisp

ACL2 is an industrial-strength version of the Boyer-Moore theorem prover, Nqthm, written by Matt Kaufmann and me.

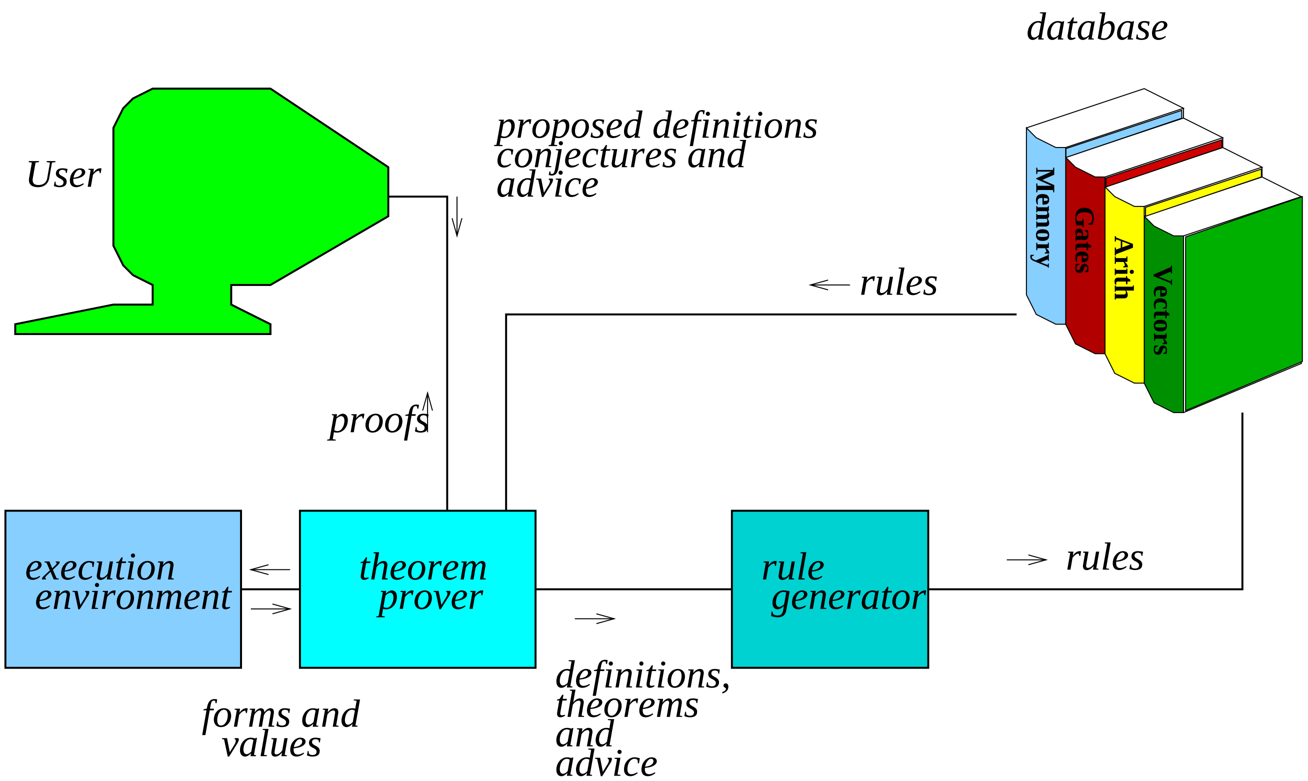
Guiding Principles

- There is tension between the expressive power of a logic and the ease with which proofs can be found mechanically. Compromise is necessary.
- Soundness-preserving extension principles make a simple logic useful.
- The duality between recursion and induction can guide proofs.
- Previously proved lemmas can guide proofs.
- *The logic must be efficiently executable.*
- Decision procedures should be used whenever practical.

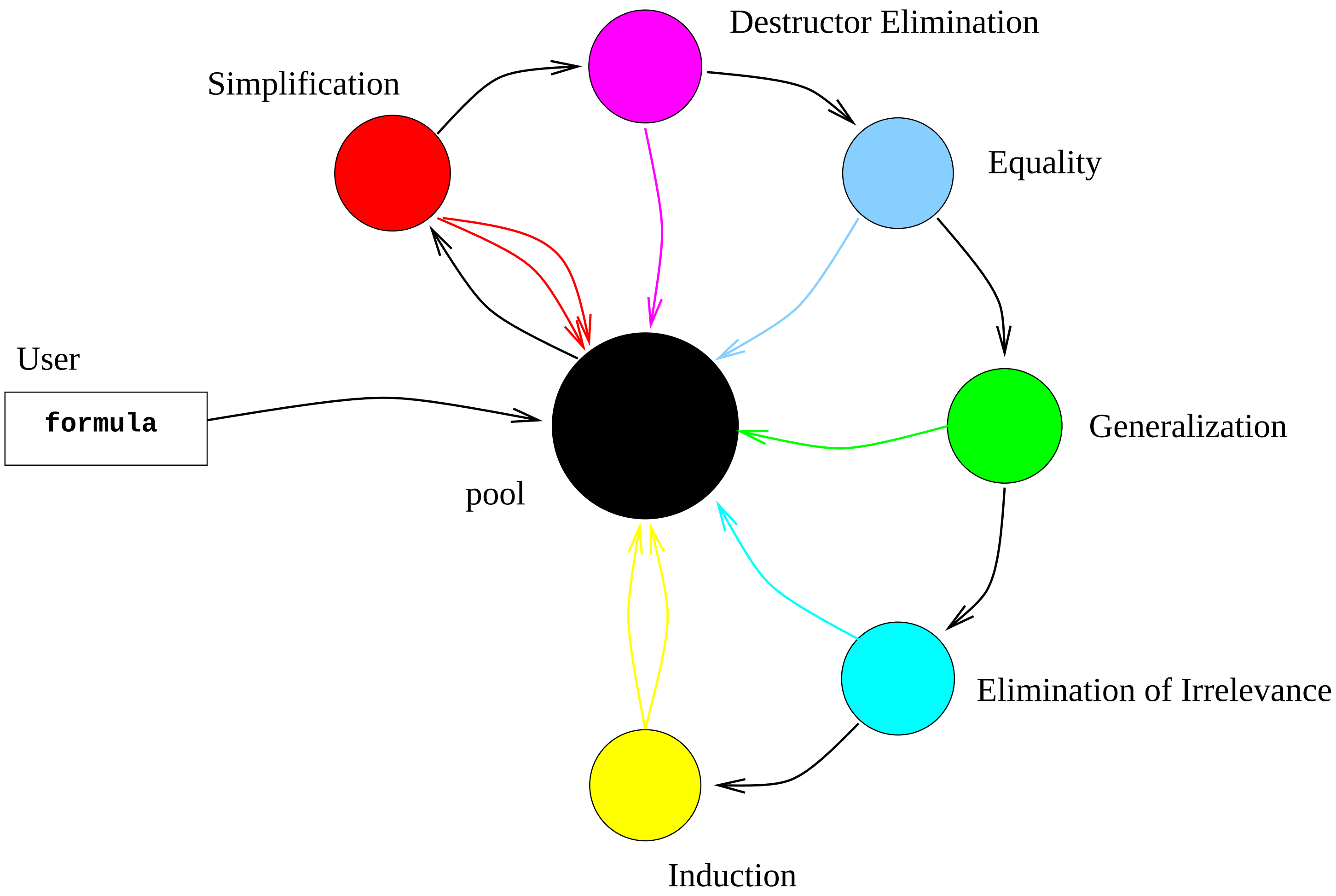
The Logic is Applicative Common Lisp

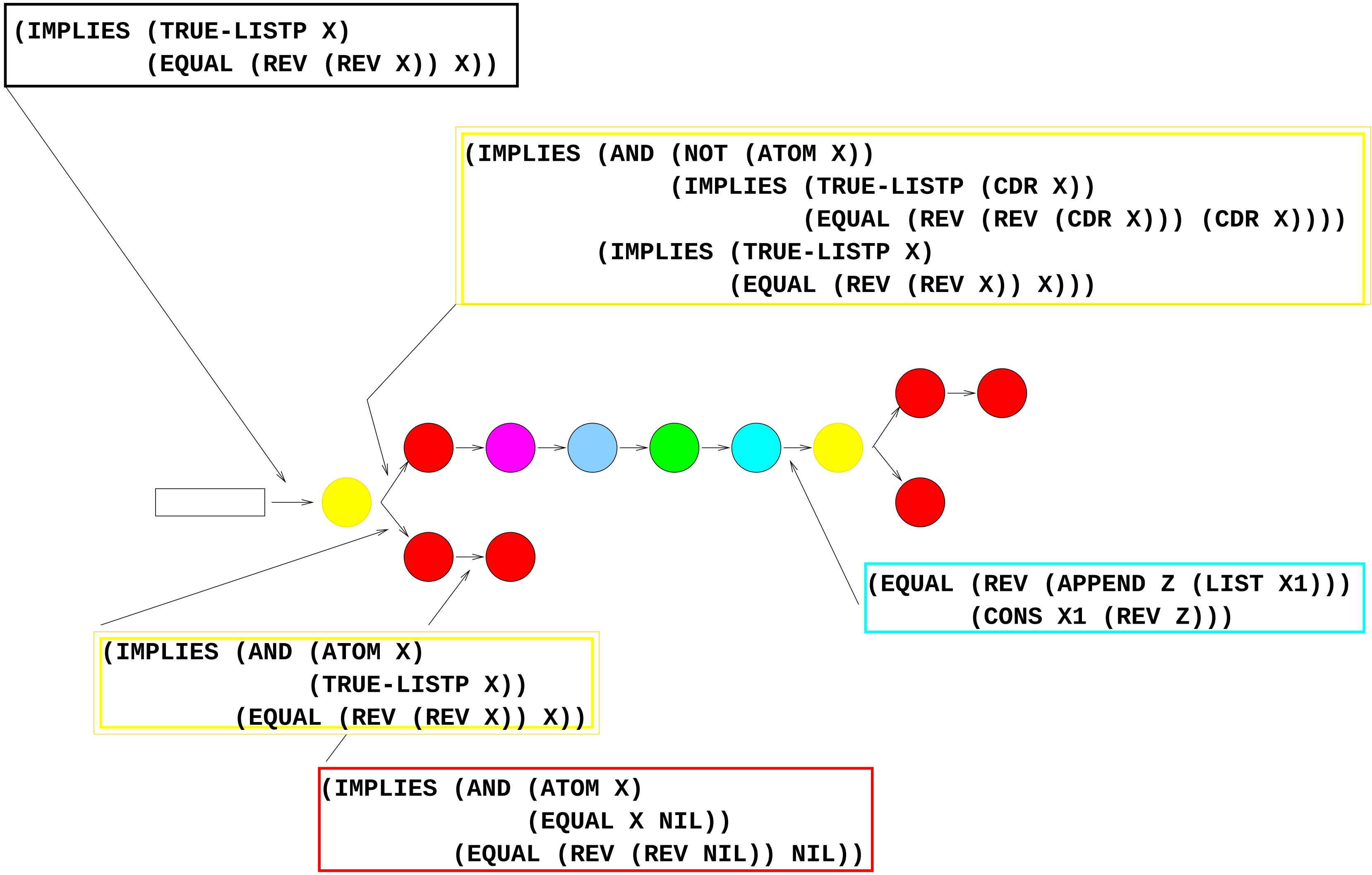
```
(defun append (x y)
  (cond ((atom x) y)
        (t (cons (car x) (append (cdr x) y)))))

(defun rev (x)
  (cond ((atom x) nil)
        (t (append (rev (cdr x))
                   (list (car x))))))

(defun true-listp (x)
  (if (atom x)
      (equal x nil)
      (true-listp (cdr x))))

(defthm rev-rev
  (implies (true-listp x)
           (equal (rev (rev x)) x)))
```

The ACL2 System



The Theorem Prover




(IMPLIES (TRUE-LISTP X)
(EQUAL (REV (REV X)) X))
(IMPLIES (AND (NOT (ATOM X))
(IMPLIES (TRUE-LISTP (CDR X))
(EQUAL (REV (REV (CDR X))) (CDR X))))
(IMPLIES (TRUE-LISTP X)
(EQUAL (REV (REV X)) X)))
(EQUAL (REV (APPEND Z (LIST X1)))
(CONS X1 (REV Z)))
(IMPLIES (AND (ATOM X)
(TRUE-LISTP X))
(EQUAL (REV (REV X)) X))
(IMPLIES (AND (ATOM X)
(EQUAL X NIL))
(EQUAL (REV (REV NIL)) NIL))

Floating Point Division Microcode for AMD K5

Algorithm divide$(p, d, mode)$

1.	sd_0	$= \mathrm{lookup}(d)$	[exact 17 8]
2.	d_r	$= d$	[away 17 32]
3.	sdd_0	$= sd_0 \times d_r$	[away 17 32]
4.	sd_1	$= sd_0 \times \mathrm{comp}(sdd_0, 32)$	[trunc 17 32]
5.	sdd_1	$= sd_1 \times d_r$	[away 17 32]
6.	sd_2	$= sd_1 \times \mathrm{comp}(sdd_1, 32)$	[trunc 17 32]
7.	dh	$= d$	[trunc 17 32]
8.	dl	$= d - dh$	[exact 17 32]
9.	p_0	$= p$	[exact 17 64]
10.	ph_0	$= p_0$	[trunc 17 32]
11.	q_0	$= sd_2 \times ph_0$	[away 17 24]
12.	qdh_0	$= q_0 \times dh$	[exact 17 64]
13.	qdl_0	$= q_0 \times dl$	[exact 17 64]
14.	pt_1	$= p_0 - qdh_0$	[exact 17 64]
15.	p_1	$= pt_1 - qdl_0$	[exact 17 64]
...	...	$= \ldots$	...
29.	q_3	$= sd_2 \times ph_3$	[trunc 17 24]
30.	qq_2	$= q_2 + q_3$	[sticky 17 64]
31.	qq_1	$= qq_2 + q_1$	[sticky 17 64]
32.	$divide$	$= qq_1 + q_0$	$mode$

Using the Reciprocal

```
        3 6.
  +       -.1 7
  +         .0 0 3 4
  +       - .0 0 0 0 6 6
        ------------------
        3 5.8 3 3 3 3 4
       ------------------
1 2   ) 4 3 0.0 0 0 0 0 0 0
        4 3 2.
        ------
          -2.
          -2.0 4
          ------
            .0 4
            .0 4 0 8
            --------
          - .0 0 0 8
          - .0 0 0 7 9 2
            ------------
          - .0 0 0 0 0 8
```

Reciprocal Calculation:

$$1/12 \;=\; 0.083\overline{3} \approx \text{0.083} = sd_2$$

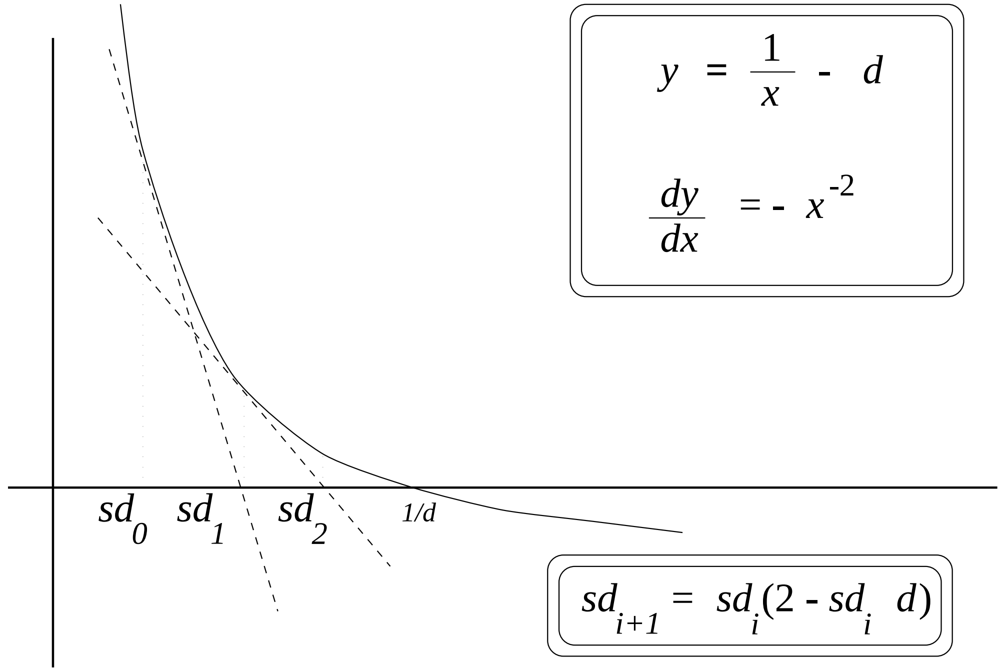
Quotient Digit Calculation:

$$\begin{array}{llll}
0.083 \times \text{430.0000} = & \text{35.6900000} \approx & \text{36.000000} = & q_0 \\
0.083 \times \text{-2.0000} = & \text{-.1660000} \approx & \text{-.170000} = & q_1 \\
0.083 \times \text{.0400} = & \text{.0033200} \approx & \text{.003400} = & q_2 \\
0.083 \times \text{-.0008} = & \text{-.0000664} \approx & \text{-.000067} = & q_3
\end{array}$$

Summation of Quotient Digits:

$$q_0 + q_1 + q_2 + q_3 = 35.833333$$

Computing the Reciprocal



$$y = \frac{1}{x} - d$$

$$\frac{dy}{dx} = - x^{-2}$$

$$sd_{i+1} = sd_i(2 - sd_i\ d)$$

The Lookup Table

top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse
1.0000000_2	0.11111111_2	1.0100000_2	0.11001100_2	1.1000000_2	0.10101010_2	1.1100000_2	0.10010010_2
1.0000001_2	0.11111101_2	1.0100001_2	0.11001011_2	1.1000001_2	0.10101001_2	1.1100001_2	0.10010001_2
1.0000010_2	0.11111011_2	1.0100010_2	0.11001010_2	1.1000010_2	0.10101000_2	1.1100010_2	0.10010001_2
1.0000011_2	0.11111001_2	1.0100011_2	0.11001000_2	1.1000011_2	0.10101000_2	1.1100011_2	0.10010000_2
1.0000100_2	0.11110111_2	1.0100100_2	0.11000111_2	1.1000100_2	0.10100111_2	1.1100100_2	0.10001111_2
1.0000101_2	0.11110101_2	1.0100101_2	0.11000110_2	1.1000101_2	0.10100110_2	1.1100101_2	0.10001111_2
1.0000110_2	0.11110100_2	1.0100110_2	0.11000101_2	1.1000110_2	0.10100101_2	1.1100110_2	0.10001110_2
1.0000111_2	0.11110010_2	1.0100111_2	0.11000100_2	1.1000111_2	0.10100100_2	1.1100111_2	0.10001110_2
1.0001000_2	0.11110000_2	1.0101000_2	0.11000010_2	1.1001000_2	0.10100011_2	1.1101000_2	0.10001101_2
1.0001001_2	0.11101110_2	1.0101001_2	0.11000001_2	1.1001001_2	0.10100011_2	1.1101001_2	0.10001100_2
1.0001010_2	0.11101101_2	1.0101010_2	0.11000000_2	1.1001010_2	0.10100010_2	1.1101010_2	0.10001100_2
...	...	...	...	...	...	...	...
1.0010110_2	0.11011010_2	1.0110110_2	0.10110100_2	1.1010110_2	0.10011001_2	1.1110110_2	0.10000101_2
1.0010111_2	0.11011000_2	1.0110111_2	0.10110011_2	1.1010111_2	0.10011000_2	1.1110111_2	0.10000100_2
1.0011000_2	0.11010111_2	1.0111000_2	0.10110010_2	1.1011000_2	0.10010111_2	1.1111000_2	0.10000100_2
1.0011001_2	0.11010101_2	1.0111001_2	0.10110001_2	1.1011001_2	0.10010111_2	1.1111001_2	0.10000011_2
1.0011010_2	0.11010100_2	1.0111010_2	0.10110000_2	1.1011010_2	0.10010110_2	1.1111010_2	0.10000011_2
1.0011011_2	0.11010011_2	1.0111011_2	0.10101111_2	1.1011011_2	0.10010101_2	1.1111011_2	0.10000010_2
1.0011100_2	0.11010001_2	1.0111100_2	0.10101110_2	1.1011100_2	0.10010101_2	1.1111100_2	0.10000010_2
1.0011101_2	0.11010000_2	1.0111101_2	0.10101101_2	1.1011101_2	0.10010100_2	1.1111101_2	0.10000001_2
1.0011110_2	0.11001111_2	1.0111110_2	0.10101100_2	1.1011110_2	0.10010011_2	1.1111110_2	0.10000001_2
1.0011111_2	0.11001101_2	1.0111111_2	0.10101011_2	1.1011111_2	0.10010011_2	1.1111111_2	0.10000000_2

The Formal Model of the Code

```
(defun divide (p d mode)
 (let* ((sd0    (eround (lookup d)                    '(exact 17 8)))
        (dr     (eround d                             '(away 17 32)))
        (sdd0   (eround (* sd0 dr)                    '(away 17 32)))
        (sd1    (eround (* sd0 (comp sdd0 32))        '(trunc 17 32)))
        (sdd1   (eround (* sd1 dr)                    '(away 17 32)))
        (sd2    (eround (* sd1 (comp sdd1 32))        '(trunc 17 32)))
        ...
        (qq2    (eround (+ q2 q3)                     '(sticky 17 64)))
        (qq1    (eround (+ qq2 q1)                    '(sticky 17 64)))
        (divide (round   (+ qq1 q0)                   mode)))
       (or (first-error sd0 dr sdd0 sd1 sdd1 sd2 dh dl p0 ph0 ... divide)
           divide)))
```

ACL2 is Executable

```
ACL2 !>(divide 1 3 '(trunc 15 48 mask-uv))

93824992236885/281474976710656
; 0.010101010101010101010101010101010101010101010101
; run time: 0.317 secs
```

(*We'll see why this is important later!*)

The Theorem

```
(defthm divide-divides
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
           (equal (divide p d mode)
                  (round  (/ p d) mode))))
```

Less formally, if p and d are floating point numbers, $d \neq 0$, and $mode$ is a rounding mode, then the algorithm produces the result of rounding p/d to a floating point number as specified by $mode$.

This work was done in collaboration with Tom Lynch and Matt Kaufmann.

Russinoff's Work

Here is an abstract from a talk recently (April, 1998) given by David Russinoff of AMD:

> Formal design verification at AMD has focused on the elementary arithmetic floating point operations, beginning with the `FDIV` and `FSQRT` instructions of the AMD-K5 processor, and continuing with the `FADD`, `FSUB`, `FMUL`, `FDIV`, and `FSQRT` instructions of the AMD-K7 processor, which is currently under development.
>
> Design-level mathematical models of all of these operations have been rigorously proved to comply with behavioral specifications derived from IEEE Standard 754 and the Intel Pentium Family User's Manual. Every step of each proof (with one minor exception in the case of `FSQRT`) has been formally encoded in the logic of ACL2 and mechanically checked with the ACL2 prover.

In this talk, we shall briefly describe the results of this project:

- a reusable general theory of floating point representation, rounding, and logical operations on bit vectors;
- an automatic translator from AMD's RTL language (essentially a subset of Verilog) to ACL2;
- several design flaws that were exposed by our analysis and ultimately corrected after surviving extensive testing;
- the proofs of correctness of the operations listed above.

(End of Russinoff's Abstract)

Russinoff's floating point books contain over 6000 mechanically proved lemmas.

To corroborate the ACL2 RTL translation, AMD *executed* it on a test suite of 80 million floating point problems and compared the results to their standard RTL simulation. Russinoff's initial model was found to match exactly the behavior of the RTL.

Constant Definition

```
(defconst *divide-table* (compute-divide-table 128))

*divide-table* =
'((1       . 255/256)      ; 1.0000000 0.11111111
  (129/128 . 253/256)      ; 1.0000001 0.11111101
  (65/64   . 251/256)      ; 1.0000010 0.11111011
  ...)
```

Function Definition

"We say a table entry, $\langle k, v\rangle$ mapping key k to value v is ε-*ok* if and only if k and v are rational numbers, $0 < v$, $|kv - 1| < \varepsilon$ and $|(k + 2^{-7})v - 1| < \varepsilon$."

```
(defun table-entry-okp (k v eps)
  (and (rationalp k)
       (rationalp v)
       (< 0 v)
       (< (abs (+ (* k v) -1)) eps)
       (< (abs (+ (* (+ k (expt 2 -7)) v) -1)) eps)))
```

If we think of v as an approximation of the inverse of x for x in the range $k \leq x < k + 2^{-7}$, then the ε-ok condition limits the relative error at the endpoints.

Recursive Function Definition

"We say a table is ε-*ok* if every entry in it is ε-ok."

```
(defun table-okp (table eps)
  (cond ((endp table) t)
        (t (and (table-entry-okp (caar table)
                                 (cdar table)
                                 eps)
                (table-okp (cdr table) eps)))))
```

Executability

```
(table-okp *divide-table* 3/512)
T
; run time: 0.033 secs
```

It is obvious that executability is important to testing formal definitions, e.g., `(divide 1 3 '(trunc 15 48 mask-uv))`.

Executability is important to proof. A key lemma in our proof has the hypothesis `(table-okp table)`.

- The lemma is proved about *any* “ok” table.
- The lemma is applied to `*divide-table*` and its hypothesis is relieved by execution.

Formalizing Floating Point Numbers

Every non-0 rational number x can be uniquely represented in the form $\sigma \times s \times 2^e$ where

- $\sigma \in \{+1, -1\}$,
- s is a rational and $1 \leq s < 2$, and
- e is an integer.

For example, let $x = 1/3 = 0.01\overline{01}_2 = +1 \times 4/3 \times 2^{-2}$.

- $\sigma_x = +1$,
- $\mathrm{s}_x = \mathrm{s}(x) = 1.\overline{01}_2 = 4/3$, and
- $\mathrm{e}_x = \mathrm{e}(x) = -2$

Truncation of Significands

$$\text{truncn}(s, n) = \frac{\lfloor s \times 2^{n-1} \rfloor}{2^{n-1}}.$$

For example,

$$\begin{aligned}
\text{truncn}(4/3, 5) &= \text{truncn}(1.0101\overline{01}_2, 5) \\
&= \lfloor 1.0101\overline{01}_2 \times 2^4 \rfloor / 2^4 \\
&= \lfloor 10101.\overline{01}_2 \rfloor / 2^4 \\
&= 10101_2 / 2^4 \\
&= 1.0101_2 \\
&= 21/16
\end{aligned}$$

Floating Point Numbers

We say x is an $m,, n$ *normalized floating-point number* if and only if

- x is rational,
- $\mathrm{truncn}(\mathrm{s}_x, n) = \mathrm{s}_x$, and
- $2 - 2^{m-1} \leq \mathrm{e}_x \leq 2^{m-1} - 1$.

We say x is an $m,, n$ *floating-point number* if and only if

- x a rational number,
- $\mathrm{truncn}(\mathrm{s}_x, n) = \mathrm{s}_x$, and
- $2 - n - 2^{m-1} \leq \mathrm{e}_x \leq 2^{m-1} - 1$.

Rounding Styles

$$
\begin{aligned}
\mathrm{trunc}(x,n) &= \sigma_x \times \mathrm{truncn}(\mathrm{s}_x, n) \times 2^{\mathrm{e}_x} \\
\mathrm{away}(x,n) &= \sigma_x \times \mathrm{awayn}(\mathrm{s}_x, n) \times 2^{\mathrm{e}_x} \\
\mathrm{sticky}(x,n) &= \sigma_x \times \mathrm{stickyn}(\mathrm{s}_x, n) \times 2^{\mathrm{e}_x} \\
\mathrm{awayn}(s,n) &= \begin{cases} s & \text{if } \mathrm{truncn}(s,n) = s \\ \mathrm{truncn}(s,n) + 2^{1-n} & \text{otherwise} \end{cases} \\
\mathrm{stickyn}(s,n) &= \begin{cases} s & \text{if } \mathrm{truncn}(s,n) = s \\ \mathrm{truncn}(s,n) & \text{if } \mathrm{bitn}(s, 1-n) = 1 \\ \mathrm{awayn}(s,n) & \text{otherwise} \end{cases}
\end{aligned}
$$

x	$\mathrm{trunc}(x,3)$	$\mathrm{away}(x,3)$	$\mathrm{sticky}(x,3)$
10.001_2	10.0_2	10.1_2	10.1_2
11.100_2	11.1_2	11.1_2	11.1_2
11.101_2	11.1_2	100.0_2	11.1_2

(We define three other rounding styles, not discussed here.)

Fundamental Elementary Results

In this section we implicitly assume that x and y are rationals and i is a positive integer.

Lemma
$\mathrm{s}(-x) = \mathrm{s}_x$ and $\mathrm{e}(-x) = \mathrm{e}_x$.

Lemma
For every x there is an ε of the same sign such that $\mathrm{trunc}(x, i) = x - \varepsilon$ and $|\varepsilon| < 2^{\mathrm{e}_x - i + 1}$. By "same sign" here we mean that if $x < 0$ then $\varepsilon \leq 0$ and otherwise $0 \leq \varepsilon$.

Lemma
If $x \neq 0$ then

$$\mathrm{e}(away(x, i)) = \begin{cases} 1 + \mathrm{e}_x & \text{if } 2 < \mathrm{s}_x + 2^{-i+1} \\ \mathrm{e}_x & \text{otherwise} \end{cases}$$

Lemma
If j is an integer and $i \leq j$ then $\mathrm{trunc}(\mathrm{away}(x,i),j) = \mathrm{away}(x,i)$.

Lemma
If m and n are positive integers and $\mathrm{trunc}(x,m) = x$ *and* $\mathrm{trunc}(y,n) = y$ *then* $\mathrm{trunc}(x \times y, m+n) = x \times y$.

Lemma
If s is a rounding style and i and n are positive integers such that $i \leq n$, then then $\mathrm{roundx}(s, \mathrm{sticky}(x, n+2), mode) = \mathrm{roundx}(s, x, i)$.

Lemma (Sticky Plus)
Let x be a non-0 rational that fits in $n > 0$ bits, which is to say $\mathrm{trunc}(x, n) = x$. *Let y be a rational whose exponent is at least two smaller than that of x, $1 + \mathrm{e}_y < \mathrm{e}_x$. Let k be a positive integer such that* $n + \mathrm{e}_y - \mathrm{e}_x < k$.

$$\overbrace{x\,\overbrace{xxxxxxxxxxxxxxxx}^{\mathrm{e}_x}.xxxxxxxxxxx}^{n\ bits}$$

$$\underbrace{y\,\underbrace{yyyyyyyyyyy}_{\mathrm{e}_y}.yyyyyyyyyyyyy\ yyy}_{k\ bits}\dots$$

Then $\mathrm{sticky}(x + y, n) = \mathrm{sticky}(x + \mathrm{sticky}(y, k), n)$.

Key Lemmas

Lemma. *For every non-0 rational d there exists a rational $0 \leq \varepsilon_{sd2} < 2^{-28}$ such that $sd_2 = (1/d)(1 - \varepsilon_{sd2})$.*

Lemma (Digit Separation). *Suppose that p and d are rationals and $d \neq 0$. Let $q'_3 = p_3/d$. Then*

$$\begin{array}{lcccl} q_1 \neq 0 & \rightarrow & \mathrm{e}(q_1) & \leq & \mathrm{e}(q_0) - 23, \\ q_2 \neq 0 & \rightarrow & \mathrm{e}(q_2) & \leq & \mathrm{e}(q_1) - 23, \\ q_3 \neq 0 & \rightarrow & \mathrm{e}(q_3) & < & \mathrm{e}(q_2) - 23, \textit{ and} \\ q'_3 \neq 0 & \rightarrow & \mathrm{e}(q'_3) & < & \mathrm{e}(q_2) - 23. \end{array}$$

Prior Related Work

Miner has formalized part of ANSI/IEEE-854 in PVS. Mechanically checked proofs of a few straightforward lemmas about rounding are shown.

The radix-4 SRT division algorithm has been mechanically verified with OBDD techniques by several groups, including those of Bryant and of Clarke.

Radix-n SRT division was verified using PVS by Reuss, Srivas and Shankar.

However, none of the SRT work discussed above required the formalization of *floating point number* or of *rounding mode*.

A Comparison of Theorems

Theorem. *If* $-8 \times d \leq 3 \times (ps + pc) \leq 8 \times d$ *then*

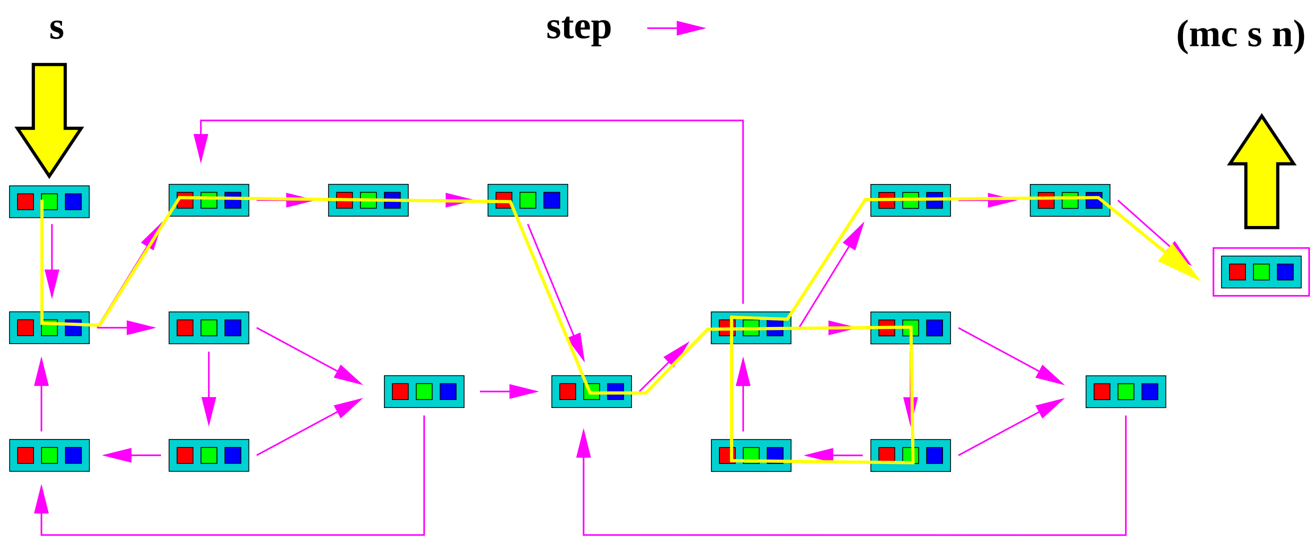
$$-8 \times d \leq 3 \times (ps' + pc') \leq 8 \times d \text{ and } ps' + pc' = 4 \times (ps + pc - q \times d),$$

where ps' *and* pc' *are the results of one iteration through the SRT circuit and "*$+$*" and "*$\times$*" denote certain Boolean operators on 70-bit words.*

Theorem. *If* p *and* d *are floating point numbers,* $d \neq 0$*, and* $mode$ *is a rounding mode, then the algorithm produces the result of rounding* p/d *to a floating point number as specified by* $mode$*.*

Which specification captures what you mean by the "correctness of a divider?"

Modeling State Machines




```
(defun mc (s n)
 (if (zp n)
     s
     (mc (single-step s) (- n 1))))
```

Industrial Microprocessor Models

- Motorola CAP Digital Signal Processor (completed)
  - no hazards $\rightarrow$ pipeline model "=" sequential model
  - no hazards in ROM microcode (established by *execution* of verified hazard condition predicate)
  - two microcode routines verified
  - microcode simulation 3 times faster than Motorola's SPW.
- IBM Austin Research Center (ongoing)
- IBM Somerset (ongoing)
- Rockwell-Collins JEM1 (silicon Java VM) (ongoing).

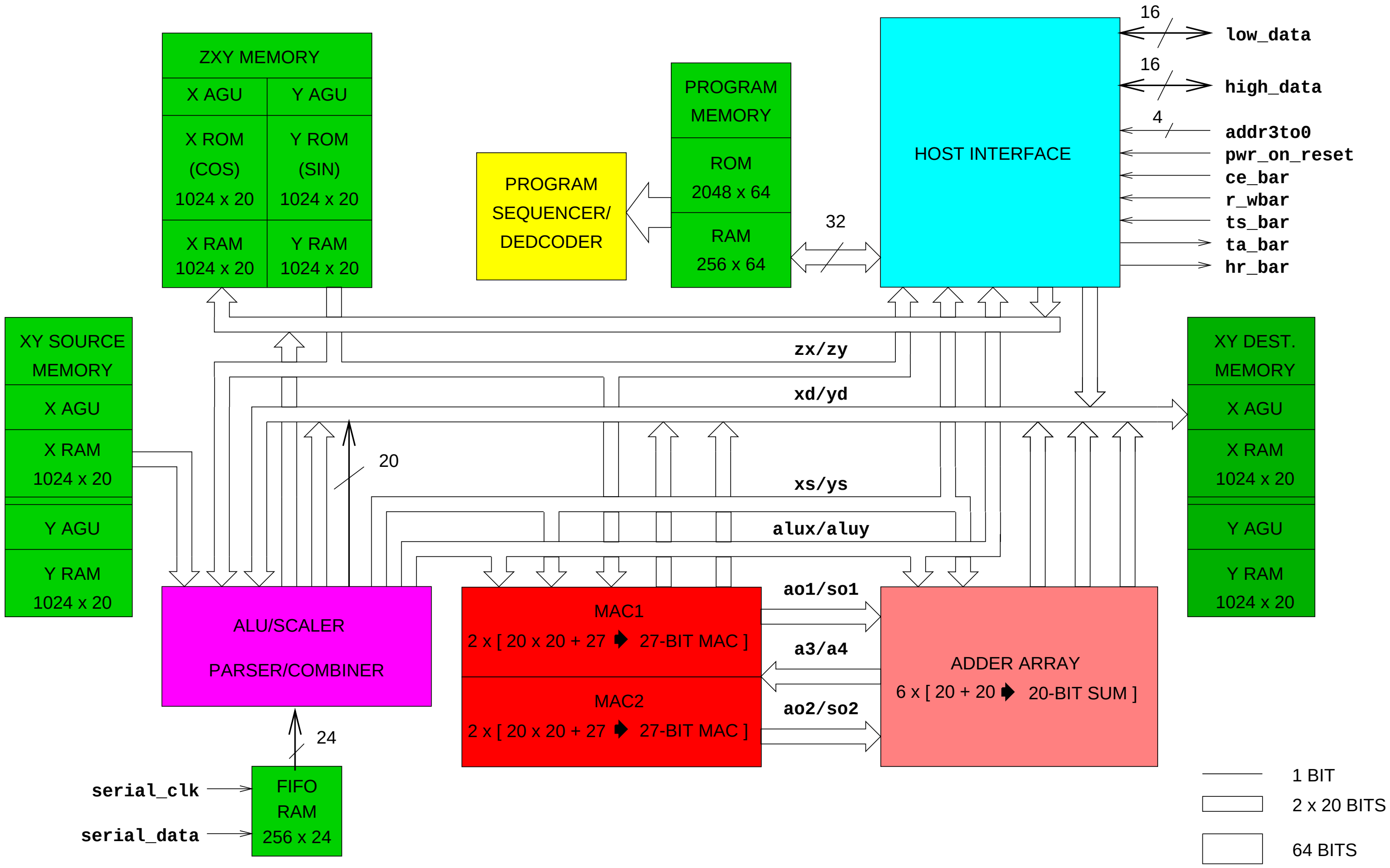
The CAP Project

The Motorola CAP (Complex Arithmetic Processor) is a proprietary digital signal processor (DSP) currently under development.

CAP was proposed by Motorola for the Speakeasy secure communications radio, the joint-service programmable waveform, multiband, multimode radio which will emulate more than 15 existing military radios.

Bishop Brock of CLI developed an ACL2 specification of the architecture, suitable for proving the correctness of DSP algorithms.

CAP Block Diagram



Features of CAP

- separate program and data memories
- 252 programmer-visible data and control registers
- 6 independently addressable data and parameter memories
- data memories are logically partitioned into ‘source’ and ‘destination’ memories; the sense of the memories may be switched under program control
- the arithmetic unit includes 4 multiplier-accumulators and a 6-adder array
- 64-bit instruction word, which in the arithmetic unit is further decoded into a 317-bit, low-level control word

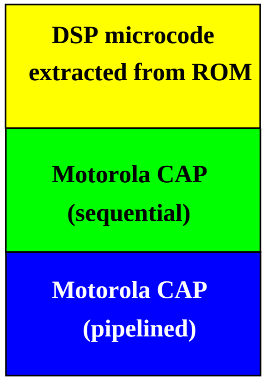
- instruction set includes no-overhead looping constructs; as many as 10 different registers are involved in the determination of the next program counter
- a single instruction can simultaneously modify over 100 registers
- the 3-stage instruction pipeline contains many hazards visible to the programmer.

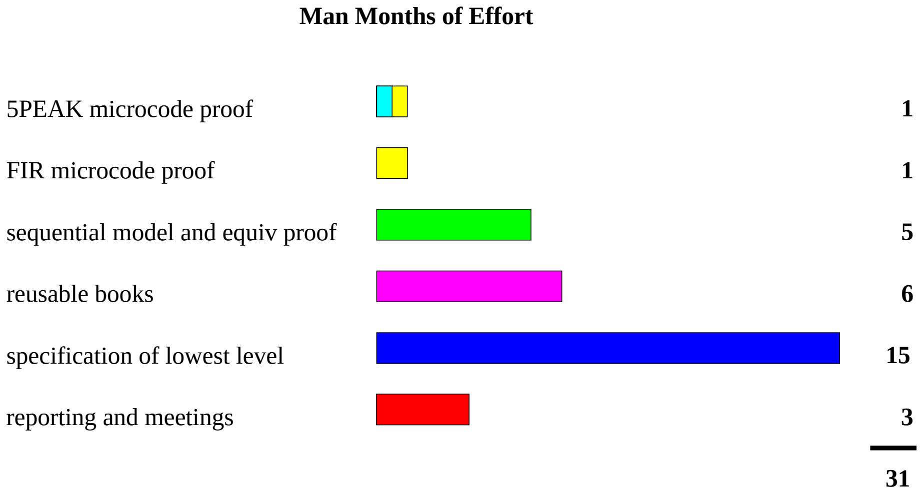
The motivation behind this complexity and unusual design is to allow Motorola engineers to code DSP applications programs on the CAP and have those programs execute with stunning efficiency.

There is no compiler targeting the CAP, perhaps because only a relatively few applications programs are required.

Our goal was to demonstrate that we could model such a complex device formally *as the design was evolving* and prove microcode correct.

CAP Project Tasks





The lowest level model was changed repeatedly to track Motorola's evolving design.

Our model of the CAP is "bit accurate" and "cycle accurate." Furthermore, it is executable. Indeed, it runs several times faster than Motorola's compiled SPW model.

An Example of the Complexity

The inner loop of one verified application is:

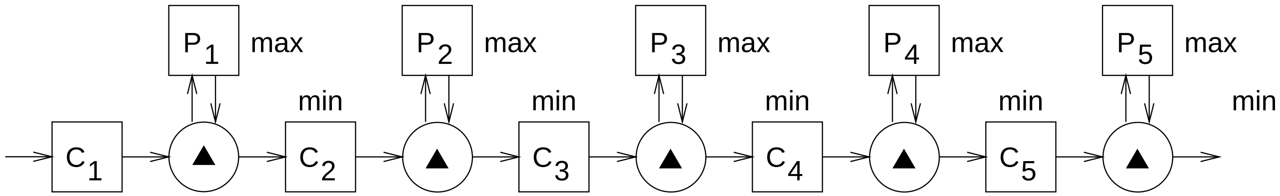
```
  repeat lc2
    mov xy(r1)+,m11,m21
+   mov zxy(r1)+,m13,m12,m23,m22
+   mpy m11,m13,a11 mpy m21,m23,a12
+   acum a11,t11,ao1m acum a12,t12,so1m
```

The behavior of this instruction depends on 9 memories, 3 stacks, and 31 registers. The instruction simultaneously modifies 17 registers.

Brock used many ACL2 features to “program” ACL2 to do “symbolic computation” on CAP states.

A Peak Finding Algorithm

The "5PEAK" program uses the CAP's array of adder/subtracters as a systolic comparator array to find the five highest peaks and their corresponding addresses.



- The array is initialized by setting each C_i and P_i to "negative infinity" (the smallest CAP integer).
- An arbitrary amount of data is streamed through the array (at one cycle/data point).
- Five "positive infinities" are streamed in.
- The last five registers are unloaded as the "five peaks" in the data.

About the 5PEAK Problem

- Motorola engineers wrote the program in CAP assembly language.
- The program actually deals with data/address pairs; it should return the five peaks and their addresses.
- Motorola generated binary microcode with their assembler and loaded it into the CAP program ROM.
- We extracted the binary from the ROM.
- To facilitate proof we invented a general sorting algorithm analogous to the one they coded (also similar to a Carey-Hansen-Thompson VLSI algorithm).
- We proved that under our CAP model the Motorola binary computes the first five elements of the output of our general sort algorithm.

- We proved that our general algorithm produces an ordered permutation of its input.

- But the most interesting lemmas concerned Motorola's use of their "infinities" to initialize and drain the comparator array.

Our Generalization of the Problem We say $\langle addr_1, data_1\rangle$ is *greater than or equal to* $\langle addr_2, data_2\rangle$ iff $data_1 \geq data_2$.

We say a list of pairs is *alternately maximal* if every other element is greater than or equal to each element to its right.

The following function steps the comparator array a, possibly exchanging alternately adjacent elements.

Definition:
step (a)
=
if endp (a) **then nil**
elseif endp (cdr (a)) **then** a
else cons (max-pair (cadr (a), car (a)),
cons (min-pair (cadr (a), car (a)),
step (cddr (a))))

'Sort', below, 'feed's a list l into an initially empty comparator array and then 'drain's it.

Definition:

$$\begin{array}{l}
\mathrm{feed}\,(l,\ a) \\
= \\
\textbf{if}\ \mathrm{endp}\,(l)\ \ \textbf{then}\ \ a \\
\ \textbf{else}\ \ \mathrm{feed}\,(\mathrm{cdr}\,(l), \\
\qquad\qquad\qquad \mathrm{step}\,(\mathrm{cons}\,(\mathrm{car}\,(l),\ a)))
\end{array}$$

Definition:

$$\begin{array}{l}
\mathrm{drain}\,(n,\ a) \\
= \\
\textbf{if}\ \mathrm{zp}\,(n)\ \ \textbf{then}\ \ a \\
\ \textbf{else}\ \ \mathrm{cons}\,(\mathrm{car}\,(a), \\
\qquad\qquad\qquad \mathrm{drain}\,(n - 1,\ \mathrm{step}\,(\mathrm{cdr}\,(a))))
\end{array}$$

Definition:

$$\mathrm{sort}\,(l) = \mathrm{drain}\,(\mathrm{len}\,(l),\ \mathrm{feed}\,(l,\ \textbf{nil}))$$

Key Theorems 'Sort' sorts and "infinities" can be used to initialize and implement 'drain' using only 'feed'.

Theorem: *If l is a list of pairs, then* sort *(l) is a weakly-descending permutation of l.*

Theorem: *Let l be a list of pairs and min be one pair, and suppose every element of l is greater than or equal to min. Let min^* be a list of n repetitions of min. Then* feed *(l, min^*) is the concatenation of* feed *(l,* **nil***) and min^*.*

Theorem: *Let a be an alternately maximal list of pairs. Let max be a pair that is greater than or equal to every element of a. Finally, let max^* be a list of n repetitions of max, where n is an integer, $0 \leq n \leq |a|$. Then* feed *(max^*, a) is the concatenation of max^* and* drain *(n, a).*

Year 2000 Compliance

EDS CIO Services constructs tools to help "remediate" COBOL code to be "Year 2000 Compliant."

Matt Kaufmann has used ACL2 to prove the correctness of some of the rules used by the tools.

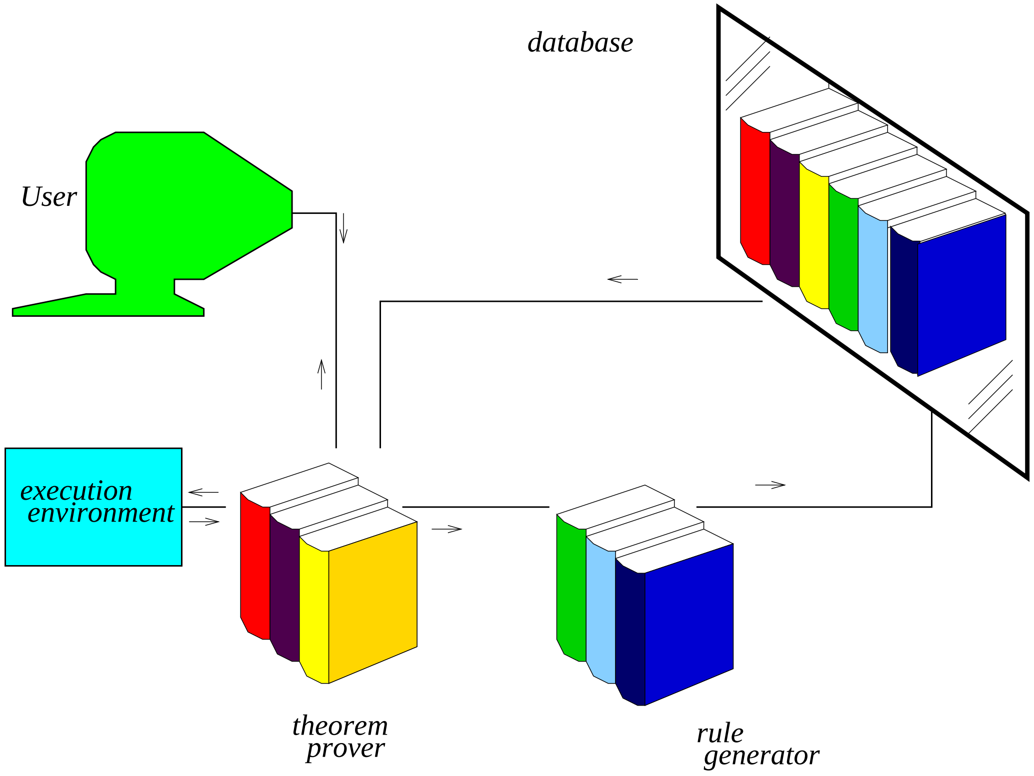
Theorem. Simple Date Comparison

```
(implies (and (datep x)
              (datep y))
         (iff (< (date x) (date y))
              (or (and (< x y)
                       (or (< *window-top* x)
                           (< y *window-bottom*)))
                  (and (< *window-top* x)
                       (< y *window-bottom*)))))
```

Such theorems get much more complicated.

- Dates are strings and are compared lexicographically.
- “Pseudo-dates” must be included, e.g., values representing uninitialized dates, largest and smallest unrepresentable dates, etc.
- Dates come in many “flavors”
  - 2 digit year dates, YY,
  - 5 digit year-day dates YYDDD,
  - 6 digit year-month-day dates YYMMDD.

An Amazing Fact



This opens up exciting possibilities in metatheoretic reasoning. It also helps us practice what we preach.