General Purpose Theorem Proving Methods in the Verification of Digital Hardware and Software

J Strother Moore

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Tx 78703

phone: 512 322-9951
fax: 512 322-0656
http://www.cli.com/
Prelude

“The fact that major theories which involve hundreds of theorems and thousands of deductions about abstractions still cleave as closely to reality as the axioms speaks for a power of mathematics to represent and predict real phenomena with an accuracy which is incredible. Why should long chains of pure reasoning produce such remarkably applicable conclusions? This is the greatest paradox of mathematics.”  Morris Kline
For several millenia, mankind has been using mathematics to describe and predict the behavior of physical systems.

**Proposition 6:** Commensurable magnitudes are in equilibrium at distances reciprocally proportional to the weights.

Archimedes

That is, \( m_1x_1 = m_2x_2 \).
The mathematics necessary to describe even simple systems may be complicated.

\[ m_1 \cdot x_1 + \int_0^{x_1} \rho \cdot x \, dx = \int_0^{x_2} \rho \cdot x \, dx + m_2 \cdot x_2 \]
Manipulating such models may require thousands of steps and rely on hundreds of theorems.

But the final result is always a true statement about the model, e.g.,

‘‘If the beam is this particular ruler (where $\rho$ is 2.16 g/inch) and $m_1$ is 20 g and $m_2$ is 37 g, then the system is in equilibrium when the fulcrum is 7.22 inches from $m_1$.’’
Lessons

• Theorems are not proved about the physical systems.
• Theorems are proved about the mathematical models.
• Models are written down in some notation.
• Models must be corroborated for accuracy.
• No amount of mathematics can guarantee that a physical system will not fail in unexpected (i.e., unmodeled) ways.
• To reject mathematical models because they cannot offer such guarantees is to ignore the most powerful tool mankind has devised for dealing with complex systems.
# A Spectrum of Choices

<table>
<thead>
<tr>
<th>Propositional Calculus</th>
<th>Arithmetic</th>
<th>Quantifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Equality</strong></td>
<td><strong>Mathematical Induction</strong></td>
<td><strong>Set Theory</strong></td>
</tr>
</tbody>
</table>

- **model checking tools**
- **general purpose theorem provers**

**Pro:** nearly automatic

**Pro:** powerful specification language

**Con:** weak specification language

**Con:** requires extensive training to use
Propositional Calculus

Higher Order Logic

Temporal Logic

Set Theory

Real Analysis

Type Theory

Algebraic Manipulation

First-Order Logic with Induction
Outline of this Presentation

- The Boyer-Moore Logic and Theorem Prover
- Some Applications
- Other General-Purpose Theorem Proving Systems
  - HOL
  - PVS
  - Nuprl
  - ACL2
  - many others
- The Future
The Boyer-Moore System (NQTHM)

In 1971, Boyer and I set out to develop a general-purpose theorem prover for a special-purpose logic.

We called it a computational logic because:

- it could be used to formalize computational problems, e.g., data structures, algorithms, programming languages, etc.;
- the value of any ground expression in the logic could be computed; and
- a powerful computational engine could be built to help discover proofs.

http://www.cli.com
In this endeavor we were following in the footsteps of

- John McCarthy, who led us to the view of Lisp as a formal mathematical logic suited for computation in the first two senses above, and

- Woody Bledsoe, who showed us how to write a theorem prover for such a logic.
Definition.
(append x y)
  =
  (if (listp x)
      (cons (car x) (append (cdr x) y))
      y).

Execution.
(append '(1 2 3) (append '(4 5) '(6 7 8)))
  =
  '(1 2 3 4 5 6 7 8).

Theorem.
(equal (append (append a b) c)
       (append a (append b c))).
Five Key Ideas in Our System

There is a trade-off between the expressivity of the logic and the ease with which proofs can be discovered.

Extensibility makes a simple logic useful.

Rewriting is very powerful.

The duality between induction and recursion can be used to guide the search for proofs.

A prover can be programmed via the selection of theorems proved.
A Simple Nqthm Example

(defn reverse (x)
  (if (listp x)
      (append (reverse (cdr x)) (list (car x))) nil))

Example Computation:
(reverse '(A B C D))
= (if (listp '(A B C D))
    (append (reverse (cdr '(A B C D))) ...) ...)
  = (if T
    (append (reverse (cdr '(A B C D))) ...) ...)
  =
= (append (reverse (cdr '(A B C D)))
  (list (car '(A B C D))))

= (append (reverse '(B C D)) '(A))

= (append '(D C B) '(A))

= '(D C B A)
Theorem. Reverse-Reverse
(implies (true-listp x)
  (equal (reverse (reverse x)) x))

Proof.

Give the conjecture the name *1.

Let us appeal to the induction principle. Two inductions are suggested by terms in the conjecture. However, they merge into one likely candidate induction. We will induct according to the following scheme:

(AND (IMPLIES (AND (LISTP X) (p (CDR X)))
  (p X))
  (IMPLIES (NOT (LISTP X)) (p X))).
Linear arithmetic and the lemma CDR-LESSP establish that the measure (COUNT X) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme generates three conjectures:

Case 3.
(IMPLIES (AND (LISTP X)
              (NOT (TRUE-LISTP (CDR X)))
              (TRUE-LISTP X))
              (EQUAL (REVERSE (REVERSE X)) X)),

which simplifies, expanding the function TRUE-LISTP, to:

    T.
Case 2.

(IMPLIES (AND (LISTP X)
   (EQUAL (REVERSE (REVERSE (CDR X)))
   (CDR X))
   (TRUE-LISTP X))
   (EQUAL (REVERSE (REVERSE X)) X)),

which simplifies, opening up the functions TRUE-LISTP and REVERSE, to:

(IMPLIES (AND (LISTP X)
   (EQUAL (REVERSE (REVERSE (CDR X)))
   (CDR X))
   (TRUE-LISTP (CDR X)))
   (EQUAL (REVERSE
      (APPEND (REVERSE (CDR X))
      (LIST (CAR X))))
   X)).
(IMPLIES (AND (LISTP X)
    (EQUAL (REVERSE (REVERSE (CDR X)))
    (CDR X))
    (TRUE-LISTP (CDR X)))
    (EQUAL (REVERSE
        (APPEND (REVERSE (CDR X))
        (LIST (CAR X))))
    X)).

Appealing to the lemma CAR-CDR-ELIM, we now replace X by (CONS A Z) to eliminate (CDR X) and (CAR X). The result is the goal:

(IMPLIES (AND (EQUAL (REVERSE (REVERSE Z))) Z)
    (TRUE-LISTP Z))
    (EQUAL (REVERSE (APPEND (REVERSE Z)
        (LIST A)))
    (CONS A Z))).
(IMPLIES (AND (EQUAL (REVERSE (REVERSE Z)) Z) (TRUE-LISTP Z)) (EQUAL (REVERSE (APPEND (REVERSE Z) (LIST A))) (CONS A Z))).

We use the above equality hypothesis by cross-fertilizing (REVERSE (REVERSE Z)) for Z and throwing away the equality. We thus obtain:

(IMPLIES (TRUE-LISTP Z) (EQUAL (REVERSE (APPEND (REVERSE Z) (LIST A))) (CONS A (REVERSE (REVERSE Z))))),
\[(\text{IMPLIES } \text{(TRUE-LISTP } Z) \quad \text{(EQUAL } \text{(REVERSE } \text{(APPEND } \text{(REVERSE } Z) \quad \text{(LIST A)}) \quad \text{(CONS } A \quad \text{(REVERSE } \text{(REVERSE } Z))))\),
\]

which we generalize by replacing \((\text{REVERSE } Z)\) by \(Y\).

\[(\text{IMPLIES } \text{(TRUE-LISTP } Z) \quad \text{(EQUAL } \text{(REVERSE } \text{(APPEND } Y \quad \text{(LIST A)}) \quad \text{(CONS } A \quad \text{(REVERSE } Y))))\).\]
(IMPLIES (TRUE-LISTP Z)
  (EQUAL (REVERSE (APPEND Y (LIST A)))
    (CONS A (REVERSE Y)))).

Eliminate the irrelevant term. We thus obtain:

(EQUAL (REVERSE (APPEND Y (LIST A)))
  (CONS A (REVERSE Y)));

which we will finally name *1.1.
Case 1.

(IMPLIES (AND (NOT (LISTP X))
              (TRUE-LISTP X))
              (EQUAL (REVERSE (REVERSE X)) X)).

This simplifies, expanding the functions TRUE-LISTP, REVERSE, and EQUAL, to:

T.
So next consider:

\[(\text{EQUAL} \ (\text{REVERSE} \ (\text{APPEND} \ Y \ (\text{LIST} \ V))) \n\quad \text{(CONS} \ V \ (\text{REVERSE} \ Y)))\],

named *1.1 above. Perhaps we can prove it by induction. Two inductions are suggested by terms in the conjecture...

(proof deleted)

That finishes the proof of *1.1, which, in turn, also finishes the proof of *1. Q.E.D.

[ 0.0 0.2 0.2 ]
REVERSE-REVERSE
But if we first prove

**Theorem. Reverse-Append**

\[
\begin{align*}
&\text{(equal (reverse (append x y))} \\
&\quad \text{(append (reverse y)(reverse x))}
\end{align*}
\]

Then **Case 2** above becomes
Case 2.
(IMPLIES (AND (LISTP X)
   (EQUAL (REVERSE (REVERSE (CDR X)))
   (CDR X))
   (TRUE-LISTP X))
   (EQUAL (REVERSE (REVERSE X)) X)),

which simplifies, opening up the functions TRUE-LISTP and REVERSE, and appealing to the lemmas CAR-CONS, CDR-CONS, CONS-CAR-CDR, and REVERSE-APPEND, to:

(IMPLIES (AND (LISTP X)
   (EQUAL (REVERSE (REVERSE (CDR X)))
   (CDR X))
   (TRUE-LISTP (CDR X)))
   (EQUAL (REVERSE
      (APPEND (REVERSE (CDR X))
       (LIST (CAR X))))
       X))

T.
( IMPLIES (AND ... (EQUAL (REVERSE (REVERSE (CDR X)))) (CDR X)) ...

(EQUAL (REVERSE (APPEND (REVERSE (CDR X)) (LIST (CAR X)))) X))

Lemma:
(reverse (append a b))

= (append (reverse b) (reverse a))

Defn:
(APPEND (REVERSE (LIST (CAR X))) (REVERSE (REVERSE (CDR X))))

Defn:
(APPEND (LIST (CAR X)) (CDR X))

Axiom:
X
Syntactic Sugar

Instead of using the standard Lisp-like notation

\[
(\text{IMPLIES} \ (\text{TRUE-LISTP} \ x) \\
(\text{EQUAL} \ (\text{REVERSE} \ (\text{REVERSE} \ x)) \ x))
\]

we will often write

\[
\text{true-listp}(x) \rightarrow \text{reverse}(\text{reverse}(x)) = x.
\]
Some Interesting Theorems Proved

- Church-Rosser Theorem for Lambda Calculus
- Gauss’ Law of Quadratic Reciprocity
- Goedel’s Incompleteness Theorem
- Paris-Harrington Ramsey Theorem.
Its extensibility has allowed our system often to be used as a “‘theorem proving shell’” in which a theorem prover for some particular “‘object system’” is built.

• Step 1. Define the concepts necessary to describe the object system.

• Step 2. Develop and prove a body of facts that make NQTHM “‘behave.’”

• Step 3. Reason in or about the object system with NQTHM.
The objects most commonly studied with Nqthm are “computing machines.”

We often formalize them operationally.

- Define a notion of *state*. In a typical programming language, a state contains some programs, some data, and some control information.

- Define the *step* function which computes the successor of a given state.

- Define the *interpreter* which iteratively applies the step function until some termination condition is met.
Modeling the Motorola 68020

A state may be modeled as a 5-tuple

- \textit{status} - set when an illegal or unformalized instruction is encountered;
- \textit{regs} - a list of register values
- \textit{pc} - a program counter (address) into memory
- \textit{cc} - condition codes
- \textit{mem} - a mapping from memory addresses to values

The mc68020 work was done by Yuan Yu (now at DEC SRC).
The ‘‘next state’’ function may be defined as follows:

**Definition.**

\[
\text{step}(s) = \begin{cases} 
\text{if evenp(pc(s))} \\
\quad \text{then} \\
\quad \quad \text{if pc-word-readp(pc(s), mem(s))} \\
\quad \quad \quad \text{then} \\
\quad \quad \quad \quad \text{execute(current-ins(pc(s), s),} \\
\quad \quad \quad \quad \quad \text{update-pc(2+pc(s) mod } 2^{32}, s)) \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{halt(’pc-outside-rom, s)} \\
\quad \text{else} \\
\quad \quad \text{halt(’pc-at-odd-address, s)} \\
\end{cases}
\]
Definition.
mc68020(s, n)
   =
   if n = 0 ∨ haltp(s)
      then s
      else mc68020(step(s), n-1) endif.
The Model is Executable

\[
\text{iread-dn}(32, 0, \text{mc68020}(s_0, n)) = 14142
\]
The Model Admits Deduction

Theorem.

\[
isqrt\text{-statep}(s, i) \implies \text{iread-dn}(32, 0, \text{mc68020}(s, \text{clock}(i))) = \sqrt{i}
\]

where \(\sqrt{i}\) is the greatest integer whose square is less than or equal to \(i\).
Another Theorem

typedef int word;            /* "word" used for optimal copy speed */
#define wsize    sizeof(word)
#define wmask    (wsize - 1)
/*
 * Copy a block of memory, handling overlap.
 * This is the routine that actually implements
 * (the portable versions of) bcopy, memcpy, and memmove.
 */
void *
memmove(dst0, src0, length)
    void *dst0;
    const void *src0;
    register size_t length;
{
    register char *dst = dst0;
    register const char *src = src0;
    register size_t t;

    if (length == 0 || dst == src)    /* nothing to do */
        goto done;

    /*
     * Macros: loop-t-times; and loop-t-times, t>0
     */
#define TLOOP(s) if (t) TLOOP1(s)
#define TLOOP1(s) do { s; } while (--t)

    ...
if ((unsigned long)dst < (unsigned long)src) {
    /*
     * Copy forward.
     */
    t = (int)src; /* only need low bits */
    if ((t | (int)dst) & wmask) {
        /*
         * Try to align operands. This cannot be done
         * unless the low bits match.
         */
        if ((t ^ (int)dst) & wmask || length < wsize)
            t = length;
        else
            t = wsize - (t & wmask);
        length -= t;
        TLOOP1(*dst++ = *src++);
    }
    /*
     * Copy whole words, then mop up any trailing bytes.
     */
    t = length / wsize;
    TLOOP(*(word *)dst = *(word *)src; src += wsize; dst += wsize);
    t = length & wmask;
    TLOOP(*dst++ = *src++);
} else {
    /*
     * Copy backwards. Otherwise essentially the same.
     * Alignment works as before, except that it takes
     * (t&wmask) bytes to align, not wsize-(t&wmask).
     */
    src += length;
dst += length;
t = (int)src;
if ((t | (int)dst) & wmask) {
    if ((t ^ (int)dst) & wmask || length <= wsize)
        t = length;
    else
        t &= wmask;
    length -= t;
    TLOOP1(*--dst = *--src);
}
t = length / wsize;
TLOOP(src -= wsize; dst -= wsize; *(word *)dst = *(word *)src);
t = length & wmask;
TLOOP(*--dst = *--src);
}
done:
    return (dst0);
}
The first of three pages of MC68020 assembly code from gcc -O.

0x2550 <memmove>:       linkw fp,#0
0x2554 <memmove+4>:     moveml d2-d4,sp@
0x2558 <memmove+8>:     movel fp@(8),d3
0x255c <memmove+12>:    movel fp@(16),d2
0x2560 <memmove+16>:    moveal d3,a1
0x2562 <memmove+18>:    moveal fp@((12),a0
0x2566 <memmove+22>:    beq 0x2604 <memmove+180>
0x256a <memmove+26>:    cmpal d3,a0
0x256c <memmove+28>:    beq 0x2604 <memmove+180>
0x2570 <memmove+32>:    bls 0x25bc <memmove+108>
0x2572 <memmove+34>:    movel a0,d1
0x2574 <memmove+36>:    movel d1,d0
0x2576 <memmove+38>:    orl d3,d0
0x2578 <memmove+40>:    movel #3,d4
0x257a <memmove+42>:    andl d4,d0
0x257c <memmove+44>:    beq 0x25a2 <memmove+82>
0x257e <memmove+46>:    movel d1,d0
0x2580 <memmove+48>:    eorl d3,d0
0x2582 <memmove+50>:    movel #3,d4
0x2584 <memmove+52>:    andl d4,d0
...
The `memmove` machine code:

```
<memmove>:  0x4e56 0x0000 0x48e7 0x3800 0x262e 0x0008 0x242e 0x0010
<memmove+16>: 0x2243 0x206e 0x000c 0x6700 0x009c 0xb1c3 0x6700 0x0096
<memmove+32>: 0x634a 0x2208 0x2001 0x8083 0x7803 0xc084 0x6724 0x2001
<memmove+48>: 0xb780 0x7803 0xc084 0x6606 0x7803 0xb882 0x6504 0x2202
<memmove+64>: 0x6008 0x7003 0xc081 0x7204 0x9280 0x9481 0x12d8 0x5381
<memmove+80>: 0x66fa 0x2202 0xe489 0x6706 0x22d8 0x5381 0x66fa 0x7203
<memmove+96>: 0xc282 0x6750 0x12d8 0x5381 0x66fa 0x6048 0xd1c2 0xd3c2
<memmove+112>: 0x2208 0x2009 0x8081 0x7803 0xc084 0x6720 0x2009 0xb380
<memmove+128>: 0x7803 0xc084 0x6606 0x7804 0xb882 0x6504 0x2202 0x6004
<memmove+144>: 0x7803 0xc284 0x9481 0x1320 0x5381 0x66fa 0x2202 0xe489
<memmove+160>: 0x6706 0x2320 0x5381 0x66fa 0x7203 0xc282 0x6706 0x1320
<memmove+176>: 0x5381 0x66fa 0x2003 0x4cee 0x001c 0xffff 0x4e5e 0x4e75
```
**Theorem.** Memmove-Correctness

\[
\text{memmove-statep}(s, \text{str1}, n, \text{lst1}, \text{str2}, \text{lst2}) \\
\rightarrow \\
\text{normal-terminationp}(\alpha, \ldots) \\
\land \\
\text{read-dn}(32, 0, \alpha) = \text{str1} \\
\land \\
\text{mem-lst}(1, \text{str1}, \text{mem}(\alpha), n, \\
\text{memmove}(\text{str1}, \text{str2}, n, \text{lst1}, \text{lst2})),
\]

where \( \alpha = \text{mc68020}(s, \text{memmove-clock}(\text{str1}, \ldots)) \).

Analogous theorems about 21 of the 22 Berkeley C String Library programs were proved with Nqthm by Yuan Yu. Three bugs were found.
Complexity

Why are these theorems so complicated?

- Because we built a poor model?
- Because we chose a weak, inexpressive logic?
- Because the Motorola 68020 is complicated?

The complexity of these models is inherent in the digital systems modeled.

We believe that an improved logical formalism will have a marginal effect on the size and manageability of any accurate model of the Motorola 68020.

The complexity must be managed, not wished away.
Corroborating the MC68020 Model

As a mathematical model of a physical system, the mc68020 function must be corroborated against reality.

We have run over 30,000 test vectors in the model and on an mc68020 chip (a Sun-3 workstation) and compared the results. The executability of our model was of crucial importance.

To date we have found about a dozen places where the model was incorrect.

The model has been revised each time and all the theorems about it were proved again automatically.

Even an imperfect model can help find bugs in widely-used programs.
The CLI Stack

INPUTS A, B, C;
OUTPUTS SUM, CARRY LEVEL FUNCTION;
DEFINE
TO (SUM1, CARRY1) = H
(SUM, CARRY2) =

die plot produced by LSI Logic, Inc., with conventional CAD tools from CLI's verified NDL

formal models related by mechanically checked mathematical proofs at CLI, Inc.

Micro-Gypsy
Piton assembly language
FM9001 machine code
Formal NDL netlist

fabricated FM9001 device
Combinational Logic

AdNet4 =
'( (N75) XOR (B0 A0))
  (N71) XOR (Cin N75))
  (N69) AND (Cin N75))
  (N68) AND (B0 A0))
  (N52) OR (N69 N68))
  ...

We can formalize the assertion ‘‘This netlist is an adder.’’

- Define the function that computes the output signals as a function of a netlist and the input signals,
  \[ \text{comb-logic}(net, a, b, c). \]

- Define the function, \( n \), that converts a list of bits to a natural number.

- Prove the following theorem:

**Theorem** AdNet4 is an adder:

\[
\text{bit-vector}(a, 4) \land \text{bit-vector}(b, 4) \land \text{bit-vector}(c, 1) \\
\rightarrow \\
\text{n(comb-logic(AdNet4, a, b, c)))} = n(a) + n(b) + n(c).
\]
We usually prove more general theorems, e.g.,

**Theorem** AdNet is an adder:
\[
\text{bit-vector}(a, i) \land \text{bit-vector}(b, i) \land \text{bit-vector}(c, 1) \\
\rightarrow \\
n(\text{comb-logic}(\text{AdNet}(i), a,b,c)) = n(a)+n(b)+n(c).
\]

by induction. We then obtain particular gate graphs by execution, e.g., AdNet(32), and correctness results by instantiation.

We have libraries of verified circuit generators together with the lemmas stating their correctness.

Large circuits need not have large proofs.
Sequential Logic

We formalize sequential logic with a function, seq-logic, which iteratively (recursively) applies combinational logic to a “state.”

**Definition.**

\[
\text{seq-logic}(\text{net, s, n}) = \\
\quad \text{if } n=0 \text{ then } s \\
\quad \text{else seq-logic}(\text{net, comb-logic(\text{net, s}), n-1}) \text{ endif.}
\]

Note: This is only a hint at the full story. Seq-logic and comb-logic are mutually recursive functions because we allow “state-holding” modules in the “combinational logic.”
A small part of a netlist for the fm9001 microprocessor:

Chip-System =
((ALU-CELL (C A B GBN GAN GA PB PAN PA M)
    (P G Z)
    ((N0 (AN) B-NOT (A))
     (N1 (BN) B-NOT (B))
     (P0 (P) P-CELL (A AN B PA PAN PB))
     (G0 (G) G-CELL (A AN BN GA GAN GBN))
     (M0 (MC) B-NAND (C M))
     (Z0 (Z) B-EQUV3 (MC P G)))
NIL)

(G-CELL (A AN BN GA GAN GBN)
    (W-3)
    ((G-2 (W-2) B-NAND (BN GBN))
     (G-1 (W-1) B-NAND (AN GAN))
     (G-0 (W-0) B-NAND (A GA))
    )

NIL)
**Theorem** (sketch).

\[ \text{fm9001-state}(s) \rightarrow \text{fm9001}(s, n) = \text{map-up}(\text{seq-logic}(\text{Chip-System}, \text{map-down}(s), \text{mcycles}(s, n))) \]
Procedure Mult(var...
var K: INT := 0;
loop
  if K le 0
  ...

Implementing Piton on FM9001

One program in a system of Piton programs

subroutine big-add ( a b n )
  push-constant f
  push-local a ; push false onto temp stack
  push-local b ; push a (addr of 1st digit in big number A)
  loop fetch ; fetch first digit of A and push it
    push-local b
    fetch ; push b (addr of 1st digit in big number B)
    add-nat-with-carry ; fetch first digit of B and push it
      ; add the two digits and push sum and carry
      ...
    ...

assemble/link
  ● allocate system resources on fm9001
  ● build symbol tables for Piton programs and data objects
  ● generate relocatable assembly code for each module
  ● assign actual locations of all system data, programs, and data objects
  ● generate absolute binary image

fm9001 state

000011111110000000001000000001
000011111110000100010001100001
01010101111111000000000000000000
00000000000000000000000000000000
0000111111111000001000110000001
...
Piton Implementation is Correct

Operational Piton Semantics:
- defines the semantics of a Piton system operationally
- executes (interprets) any explicitly given Piton system
- allows symbolic execution of Piton systems
- allows proofs of Piton programs

Previously discussed Piton implementation

Previously defined and verified fm9001 machine code interpreter

assemble/link

assemble/link

recovers Piton objects from binary image (given link tables)

display-fm9001-data-segment

k FM9001 steps

n Piton steps

p₀

s₀

pₙ

sₙ
THEOREM: FM9001 Piton is Correct

\[ (\text{proper-p-state}\ p_0) \land (\text{load-addr} \in \mathbb{N}) \land \text{p-loadable}(p_0, \text{load-addr}) \land (\text{p-word-size}(p_0) = 32) \land (p_n = p(p_0, n)) \land (\neg \text{error}(p-\text{psw}(p_n))) \land (t_s = \text{type-specification}(p-\text{data-segment}(p_n))) \rightarrow (\text{p-data-segment}(p_n) = \text{display-fm9001-data-segment}(\text{fm9001}(\text{assemble-link}(p_0, \text{boot-lst}, \text{load-addr}), \text{fm9001-clock}(p_0, n)), t_s, \text{link-tables}(p_0, \text{load-addr}))) \]
The Piton Proof

Theorem 1

Theorem 2

Corollary 1

Theorem 3

Theorem 4

Theorem 5

Theorem 6

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link

assemble/link
The CLI Stack

Procedure Mult(var... var K: INT := 0;
loop
  if K le 0
...

01011101001110101
1101000111010000001010000011000000000000000011110100011101010000001...
An Incomplete Stack

Byzantine Agreement

Clock Synchronization

Asynchronous Communication

Serial Communications Design
Some Nqthm Applications

• Mathematics: Turing completeness, Church-Rosser, Gauss’ Law, Goedel’s Theorem, Paris-Harrington Ramsey Theorem, ...

• Algorithms: fast string searching, majority vote, oral messages, dining philosophers, ...

• Hardware: FM8502, FM9001, oral messages, ...

• Languages: mc68020 machine code, Piton, Ada, Lisp, ...

• Compilers: Piton, Micro-Gypsy, Pure Lisp

• Operating Systems: KIT, Mach kernel specification, EDF scheduling
In Support of General Purpose Approaches

- The mathematical language need not be extraordinarily rich to allow an extremely broad range of applications.
- A general purpose system allows you to view and analyze the same system at many different levels of abstraction.
- A general purpose system allows you to relate the multiple views of a system.
- One tool is sufficient to do hardware and software. This is crucial because hardware design is becoming “just” a programming problem.
Other General Purpose Systems

http://www-formal.stanford.edu/clt/ARS/ars-db.html
http://www.comlab.ox.ac.uk/archive/formal-methods.html

• brief descriptions of ~40 automated reasoning systems
• links to the appropriate sites
• references
http://www-formal.stanford.edu/clt/ARS/ars-db.html

<table>
<thead>
<tr>
<th>3TAP</th>
<th>CLAM</th>
<th>CWB</th>
<th>Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECLiPSe</td>
<td>EHDM</td>
<td>Elf</td>
<td>EVES/NEVER</td>
</tr>
<tr>
<td>GETFOL</td>
<td>ft</td>
<td>HOL</td>
<td>IMPS</td>
</tr>
<tr>
<td>INKA</td>
<td>Isabelle</td>
<td>Jape</td>
<td>KEIM</td>
</tr>
<tr>
<td>KIV</td>
<td>Kripke</td>
<td>LAMBDA</td>
<td>LA</td>
</tr>
<tr>
<td>LeanTAP</td>
<td>LEGO</td>
<td>Merill</td>
<td>Mizar</td>
</tr>
<tr>
<td>NQTHM</td>
<td>NUPRL</td>
<td>OBJ3</td>
<td>Otter</td>
</tr>
<tr>
<td>Oyster</td>
<td>Pc-Nqthm</td>
<td>ProCom</td>
<td>ProofPower</td>
</tr>
<tr>
<td>PROTEIN</td>
<td>PVS</td>
<td>ReDuX</td>
<td>RRL</td>
</tr>
<tr>
<td>SDVS</td>
<td>Setheo</td>
<td>SPIN</td>
<td>Tableaux</td>
</tr>
<tr>
<td>TLA</td>
<td>TPS</td>
<td>UV</td>
<td></td>
</tr>
</tbody>
</table>
HOL

HOL -- an interactive environment for machine-assisted theorem-proving in higher-order logic.

• Brief Description

• Contact:
  hol-support@cl.cam.ac.uk
  John Harrison jrh@cl.cam.ac.uk
  Phillip J. Windley <windley@cs.byu.edu>

• HOL page at BYU (on-line documentation)
  (http://lal.cs.byu.edu/lal/hol-documentation.html)
  HOL page at Oxford
  HOL page at Cambridge
HOL evolved by Mike Gordon (Cambridge) from LCF by replacing Scott’s Logic of Computable Functions with classical higher order logic.

HOL is an ML programming environment for building specialized proof tools.

Characteristics

- system building libraries
- fully expanded proofs
- open system architecture
- large user community and annual users meetings
Sample Session

An extract from an HOL proof by John Harrison of Stone’s theorem that every metrizable space is paracompact:

Definitions:
let PARACOMPACT=new_definition('PARACOMPACT', "PARACOMPACT(top:* topology) =
   !C. OPEN_COVER(top) C ==> 
   ?C'. OPEN_COVER(top) C' /
   C' REFINES C /
   LOCALLY_FINITE(top) C'"');
Main proof:
let STONE = prove_thm('STONE',
  "!m:(*)metric. PARACOMPACT(mtop m)",
  REWRITE_TAC[PARACOMPACT; OPEN_COVER] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_THEN(X_CHOOSE_THEN
  "$<<:(*->bool)->(*->bool)->bool"
    STRIP_ASSUME_TAC) STONE_WO THEN
  Several hundred lines deleted
  GEN_REWRITE_TAC (RAND_CONV o ONCE_DEPTH_CONV) [] [INTER_COMM] THEN
  CONV_TAC(REDEPTH_CONV LEFT_AND_EXISTS_CONV) THEN
  Total number of primitive inferences: 18433

Tactics could be used to reduce this to one command!
Some Papers at the Upcoming HOL Conference

- *Experiments with ZF Set Theory in HOL and Isabelle*, Sten Agerholm and Mike Gordon
- *Floating Point Verification in HOL*, John Harrison
- *Formal Verification of a Pipelined Serial Multiplier*, Jang Dae Kim and Shiu-Kai Chin
- *Formal Verification of Counterflow Pipeline Architecture*, Paul Loewenstein
- *Deep Embedding VHDL*, Ralf Reetz
Two noteworthy current HOL projects at Cambridge:

- verification of an ATM switch  
  (http://www.cl.cam.ac.uk/Research/HVG/atmproof.html)

- theorem proving support for the C programming language  
  (http://www.cl.cam.ac.uk/users/mn200/PhD/)
PVS

PVS (Prototype Verification System) -- a specification verification system based on higher-order logic.

• Brief Description

• Contact:
  John Rushby, N. Shankar, Sam Owre
  {rushby, shankar, owre}@csl.sri.com

• PVS home page
  (http://www.csl.sri.com/pvs.html)
PVS supports classical higher-order logic with

- functions,
- sets,
- records,
- tuples,
- predicate subtypes,
- dependent typing, and
- theories with type and individual parameters.
Proofs are developed interactively by combining high-level inference procedures

- Typechecking
- ground decision procedures
- BDD-based propositional simplification
- model-checking
- rewriting
- induction
sum: THEORY
BEGIN
n: VAR nat
f,g: VAR [nat -> nat]

sum(f,n): RECURSIVE nat =
(IF n = 0
  THEN 0
  ELSE f(n-1) + sum(f, n-1) ENDIF)
MEASURE n

sum_plus: LEMMA
sum((lambda n: f(n)+g(n)), n) = sum(f, n)+ sum(g, n)
END sum
sum_plus:

|--------

{1} (FORALL (f: [nat -> nat], g: [nat -> nat], n: nat):
  sum ((LAMBDA (n: nat): f(n) + g(n)), n)
  = sum(f, n) + sum(g, n))

Rule? (induct-and-rewrite! "n")

sum rewrites sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0)
  to 0
sum rewrites ...
...

Q.E.D.
Some Noteworthy PVS Projects

• Oral Messages algorithms for Byzantine Agreement and Interactive Consistency

• Fischer’s mutual exclusion algorithm and a generalized railroad-crossing example.

• Verification of an N-process generalization of Peterson’s mutual exclusion algorithm.
• The Collins AAMP5 avionics processor. The micro-
architecture and most of the instruction set architecture
(108 out of 209 instructions) of this complex processor
(500,000 transistors, three-stage pipeline, stack architec-
ture, autonomous subunits) were formally specified in
PVS. The microcode of 11 representative instructions was
formally verified. Errors were found.
NUPRL

NUPRL is a proof system for intuitionistic type theory based on Martin Lof type theory, with proof by refinement and extraction of programs from proofs.

http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html

For an interesting example, see


for a NuPrl verification of an algorithm for synthesizing optimum-area boolean circuits.
Designing a New Computational Logic

The problems inspiring Nqthm’s design were small compared to those to which it was eventually applied.

Problems of practical interest
- demand efficient execution
- generate huge systems of definitions
- are produced by multiple users
Some of Nqthm’s Flaws

• prototyping formal models is expensive
• executing formal models is (relatively) slow
• some useful proof techniques are unavailable
• the system is awkward to control
• no provision for multiple users or shared libraries
• implemented in an informally specified language
The New Logic: Acl2

Acl2 is a subset of applicative Common Lisp.

The syntax is Common Lisp, and includes user-defined macros.

Data types

- numbers, e.g., 23, -17, 22/7, #c(3 5),
- characters, e.g, \A, \a, \New-Line,
- strings, e.g., "Arithmetic Overflow",
- symbols (with packages), e.g., STEP, FM9001::STEP, MC68020::STEP,
- conses, ‘((A . 1) (B . 2) (C . 3)).
Approximately 150 Common Lisp function symbols on these data types are axiomatized.

Functions have *guards* characterizing their domains.

Guards are arbitrary propositions in the logic.

Guard checking is undecidable.
To applicative Common Lisp we add

- single-threaded notion of state containing open channels and files,
- fast applicative arrays,
- fast applicative property lists, and
- multiply-valued functions.
Our Claim for Acl2

Suppose

• fn is a function of one argument, \( x \),

• the guard of fn is \( T \) and all guard theorems have been proved, and

• \( fn(x) = T \) has been proved.

Then, in every compliant Common Lisp, every invocation of fn on an object in the Acl2 universe terminates without error (except, possibly, for resource errors) and returns \( T \).
System Architecture

- User
- Execution environment
- Theorem prover
- Rule generator
- Database

- Proposed definitions, conjectures, and advice
- Proofs
- Definitions, theorems, and advice
- Forms and values

- Execution environment
- Theorem prover
- Rule generator
- Database

- Memory
- Gates
- Arith
- Vectors
Definition.
Map-Down(hi-state) = ...

Lemma.
Map-Down(Hi(s,n)) = Lo(...)

Theorem. Hi-is-Correct
Good-Hi-Statep(s) -> Hi(s,n) = Map-Up(Lo(...))
Some Traditional Acl2 Books

- Vectors
- List Processing
- Integers
- Rationals
- Complex Rationals
- Naturals
- Groups
- Rings
- Arithmetic

Complex Rationals
Ongoing Projects

- Hardware Designs
  - Fault Tolerance
  - DSP Applications
    - Programs
      - (like Berkeley C String Library)
  - Motorola DSP
    - (like MC68020)
- VHDL
- Address and Memory Management
- Task Isolation
- VHDL Hardware Designs
- Fault Tolerance
- DSP Applications Programs
- (like Berkeley C String Library)
- Motorola DSP
  - (like MC68020)
- Mach
  - (like KIT)
- Ada
  - (like Piton)
- Nqthm
  - (like nothing I can think of!)
The CAP Project

Mathematics

\[ \sum_{i=1}^{n} c_i \]

Algorithms

ACL2

ACL2 MODEL

ACL2 NETLIST

RTL

SPW MODEL

Hardware

FORMAL CAD LANGUAGE

Physical

CAP ASSEMBLER

CAD TOOLS

MOTOROLA

Mechanical Link

Manual Link

Testing

Verification
Pipeline View

SEQUENCER
PC  PGM

I0 (FETCH)

Z MEM
AGU

XY SRC
AGU

XY DEST
AGU

I1 (DECODE)

ALU

ADDER ARRAY

MAC1,2

BUSES

I2 (EXECUTE)
Non-Pipeline View
Proof of Equivalence

\[
\text{cap-step-predicate-n}(s,n) \rightarrow \\
\text{stall(stall(pipe-n(s,n)))} \sim \text{non-pipe(stall(stall(s)),n)}
\]
(defun cap-step-predicate (cap-state)
  (declare
   (xargs :guard (and (cap-state-p cap-state)))))
(let ((seq/dec (seq/dec cap-state))
      (i-execute (i-execute (v-pipe cap-state)))
      (i-decode (i-decode (v-pipe cap-state)))
      (i-fetch (i-fetch (seq/dec cap-state))))
  (and (sequencer-ok-p seq/dec)
       (no-bp-trap seq/dec)
       (trap-registers-stable-p i-execute)
       (trap-registers-stable-p i-decode)
       (immediate-lc-ok-p i-execute i-fetch)
       (immediates-ok-p i-decode *nop*)
       ...
       (immediate-lc-ok-p i-execute i-fetch)
       (immediate-lc-ok-p i-decode i-fetch)
       (implies
        (immediate-move-p i-decode)
        (not (agu-interference-p
              (i-field :register i-decode)
              i-fetch))))))
The conjuncts specify that:

- The sequencer must be in a state that will execute without error.
- The sequencer will not breakpoint or trap.
- No immediate moves will alter the registers that control breakpoints and traps.
- There must not be any interference between an immediate move to a loop-counter and the current instruction.
- There must be no ‘hard’ interference or any immediate move errors (e.g., move to an unspecified address).
- There is no AGU interference of any kind (hard or soft) between the decode-phase instruction and the fetch-phase instruction.
Libraries Used for CAP

- **BASIC MATH**
  - 120 lemmas
  - 1K

- **QUOTIENT/REMAINDER MATH**
  - 83 lemmas
  - 2K

- **LOGICAL OPS. ON INTEGERS**
  - 88 definitions
  - 2.5K

- **HARDWARE MATH**
  - 216 lemmas
  - 2.5K

- **1-DIMENSIONAL ARRAYS**
  - 120 lemmas
  - 1K

- **RECORD STRUCTURES**
  - 83 lemmas
  - 3.5K

- **LISTS / ALISTS**
  - 88 definitions
  - 3.5K

- **CAP**
  - 199 definitions
  - 174 theorems (guards)
  - 31 types
  - 12K

- **MC68020**
  - 1K
  - 6K

- **PARA**
  - 1K
Open Architecture of Acl2

(in-package "nqthm")

Theorem.
(nlistp x) => (car x) = 0

Definition.
nqthm::car(x)
= if nqthm::listp(x)
  then car(x)
  else 0.

execution environment

rules

theorem prover

rules

rule generator

"Nqthm"

Nqthm books
An Amazing Fact

- User
- Execution Environment
- Database
- Theorem Prover
- Rule Generator

4.5 mb = 100k lines
Practicing What We Preach

- The logic must be efficient
  - The heuristic must handle problems of practical scale
- The system can reason about itself
- Sophisticated metafunctions are possible
- A trusted host is possible
- A soundness proof is possible
The Future

model checking and related "push button" approaches

near term

general purpose methods

hardware design

software design

longer term