Embedding ACL2 in HOL

Mike Gordon, Warren A. Hunt, Jr., Matt Kaufmann, James Reynolds
Embedding ACL2 in HOL

Mike Gordon, Warren A. Hunt, Jr., Matt Kaufmann, James Reynolds

- Higher-order logic
  - proof in HOL4
- First-order ACL2 logic in HOL
  - trusted code translating ML and LISP S-expressions
- ACL2 input file
  - proof in ACL2
- Optimised ACL2 specification
HOL and ACL2

- Higher order logic (HOL) can express pretty much anything
  - traditional textbook semantics
    - denotational semantics needs higher order functions
    - operational semantics needs inductive relations
  - arbitrary mathematics
    - classical analysis (e.g. measure theory)
    - infinite stream processing (e.g. Cryptol semantics)

- ACL2 is a programming language and a theorem prover
  - ACL2 logic terms = Common Lisp programs
  - theorem prover for first order logic (FOL) + induction
  - high assurance + fast execution + strong proof automation

- Some projects committed to HOL, others to ACL2
  - Cambridge ARM project committed to HOL
  - Rockwell-Collins AAMP7 committed to ACL2
  - Galois SHADE project uses both HOL and ACL2

Gordon, Hunt, Kaufmann, Reynolds
Embedding ACL2 in HOL (ACL206 Workshop, Seattle)
Motivating examples for linking HOL and ACL2

- ACL2 as a HOL simulation engine
  - translate HOL specifications into first-order ACL2
  - export ACL2-in-HOL to ACL2 system
  - run on ground data using ACL2 stobj-execution

- Validate the Galois Connections Cryptol-to-ACL2 compiler
  - Cryptol semantics easier in HOL than in ACL2
  - Galois SHADE tool translates Cryptol to AAMP7 via ACL2
  - validate SHADE compilation of $D$ by HOL proof of
    $\vdash \text{CryptolSemantics}(D) \equiv \text{Acl2ToHol}(\text{SHADE}(D))$

- Use HOL measure theory to validate ACL2 primality test
  - Miller-Rabin test easy to code in ACL2, but hard to specify
  - HOL has a library supporting measure theory (Hurd)
  - validate ACL2 checker against HOL measure theory spec
This talk is background, motivation and simple overview
- workshop proceedings contain technical details
- emphasises low level logical issues

Companion paper to be presented at FMCAD 2006
- more comprehensive
- emphasises automatic encoding/decoding tools in HOL

Code and examples in SourceForge repository for HOL4

http://hol.cvs.sourceforge.net/hol/hol98/examples/acl2/
Previous work

- PM (Proof Manager) by Fink, Archer and Yang (UC Davis)
  - low emphasis on logical issues
  - main effort on unified UI for various provers

- ACL2PII by Staples
  - uses Prosper Integration Interface (PII)
  - more emphasis on logic issues than PM
  - tricky translation from HOL to FOL by ML scripts
  - used by Susanto to run his unverified ARM model
Requirements of current work

- Believable soundness story
  - earlier attempt not accepted by ACL2 community

- Handle big examples robustly
  - run software on Fox’s verified ARM6 model

- Ease of use
  - value can be realised with only minimal knowledge of ACL2

- Compatible with Isabelle/HOL
  - Galois (Matthews) uses Isabelle/HOL for Cryptol semantics
Our approach: HOL theory SEXP of ACL2 logic

- Machine verified translation between higher order logic and first order SEXP theory

- Clean translations between HOL/SEXP and ACL2
  - ML tool writes HOL/SEXP to ACL2 input files
  - LISP tool writes ACL2 to HOL/SEXP input files

- Machine verified translation between expanded ACL2 and conventional style ACL2
ACL2: programming language or logic?

\[(\text{EQUAL} \ (\ast \ (\ast \ X \ Y) \ Z) \ (\ast \ X \ (\ast \ Y \ Z)))\]

[\text{ASSOCIATIVITY-OF-\ast \ from ACL2 file axioms.lisp}]

1. An S-expression in Lisp?

   valid because if \(X, Y\) and \(Z\) are replaced by any S-expressions, then the resulting instance of the axiom will evaluate to \(t\) in Lisp

2. A formula of first order logic?

   defines what it means for evaluation to be correct: it is a partial semantics of Lisp evaluation

Second approach adopted:

\text{axioms.lisp defines the ACL2 logic}

differences between this and Lisp behaviour (when there are no guard violations) viewed as bugs in Lisp, not in the ACL2 axioms.
First, a datatype of S-expressions in higher order logic

\[
\begin{align*}
\text{type}_\text{abbrev} & ("\text{packagename"}, "\text{:string"}) \\
\text{type}_\text{abbrev} & ("\text{name"}, "\text{:string"}) \\
\text{Hol_datatype} & \\
\text{`sexp} = \text{ACL2_SYMBOL} & \quad \text{of packagename} = \to \text{name} \\
| & \text{ACL2_STRING} \quad \text{of string} \\
| & \text{ACL2_CHARACTER} \quad \text{of char} \\
| & \text{ACL2_NUMBER} \quad \text{of complex_rational} \\
| & \text{ACL2_PAIR} \quad \text{of sexp} = \to \text{sexp}'
\end{align*}
\]

Similar to Staples’ ML definition, but inside the HOL logic

\text{complex_rational} \quad \text{built from rationals} \quad \text{(Jens Brandt)}
Overloading used to manage ACL2 names

- `acl2Define "acl2Name" 'holName . . .`  
- Constant `acl2Name` defined, then overloaded on `holName`
- Full ACL2 names simplify SEXP $\leftrightarrow$ ACL2 correspondence

Simple examples: overload `sym` on `ACL2_SYMBOL`, then:

- `acl2Define "COMMON-LISP::NIL" 'nil = sym "COMMON-LISP" "NIL"`'
- `acl2Define "COMMON-LISP::T" 't = sym "COMMON-LISP" "T"`'
- `acl2Define "COMMON-LISP::EQUAL" 'equal x y = if x = y then t else nil`
More examples: overload \texttt{cons} on \texttt{ACL2\_PAIR}, then:

\begin{verbatim}
 acl2Define "COMMON-LISP::CAR"
 '(car(cons x _) = x) \land (car _ = nil)'

 acl2Define "COMMON-LISP::CDR"
 '(cdr(cons _ y) = y) \land (cdr _ = nil)'

 acl2Define "COMMON-LISP::IF"
 'ite x y z = if x = nil then z else y'
\end{verbatim}

31 ACL2 primitives in \texttt{axioms.lisp}:
\begin{verbatim}
 acl2-numberp bad-atom<=  binary-* binary+-  unary-- unary-/  <  car  cdr  char-code  characterp  code-char  complex  complex-rationalp  coerce  cons  consp  denominator  equal  if  imagpart  integerp  intern-in-package-of-symbol  numerator  pkg-witness  rationalp  realpart  stringp  symbol-name  symbol-package-name  symbolp
\end{verbatim}

All these ACL2 primitives have been defined in HOL

Some tricky to get right (e.g. \texttt{symbolp} – see paper)!
S-expression \( p \) corresponds to formula \( \neg (p = \text{nil}) \)
- so define: \( (\models p) = \neg (p = \text{nil}) \)

Note that \( 1 \) is a theorem of ACL2: \( \vdash \models 1 \)

Some ACL2 axioms are trivial to prove
- \( \vdash \forall x\ y. \models \text{equal} (\text{car} (\text{cons} x\ y))\ x \)
- \( \vdash \forall x\ y. \models \text{equal} (\text{cdr} (\text{cons} x\ y))\ y \)

Others are harder
- may just be hard (e.g. validity of \( \varepsilon_0 \)-induction)
- or have lots of fiddly details

78 axioms: we are slowly working through their proofs ...
Coding HOL values as S-expressions

Universe of HOL types

sexp
A simple HOL type definition:

\begin{verbatim}
Hol_datatype `colour = R | B`
\end{verbatim}

The following theorems are generated automatically:

\begin{verbatim}
⊢ encode_colour t = case t of R -> nat 0 || B -> nat 1

⊢ decode_colour x = 
  if x = nat 0 then R else if x = nat 1 then B else ARB

⊢ colourp x = ite (equal (nat 0) x) t (equal (nat 1) x)

⊢ decode_colour(encode_colour x) = x

⊢ (|= colourp x) ==> (encode_colour(decode_colour x)=x)

⊢ |= f(case a of R -> C0 || B -> C1) = 
  ite (equal(encode_colour a)(nat 0)) (f C0) (f C1)
\end{verbatim}

Can handle recursive datatypes (e.g. red-black trees)
From a HOL function definition:

\[ (\text{flip\_colour} \; R = B) \land (\text{flip\_colour} \; B = R) \]

The following are generated automatically:

- definition of encoding function
  \[ \triangledown \text{acl2\_flip\_colour} \; a = \]
  \[ \text{ite} \; (\text{colourp} \; a) \]
  \[ (\text{ite} \; (\text{equal} \; a \; (\text{nat} \; 0)) \; (\text{nat} \; 1) \; (\text{nat} \; 0)) \]
  \[ (\text{nat} \; 1) \]

- recogniser theorem
  \[ \triangledown \vdash \text{colourp}(\text{acl2\_flip\_colour} \; a) \]

- correctness theorem
  \[ \triangledown \text{encode\_colour}(\text{flip\_colour} \; a) = \text{acl2\_flip\_colour}(\text{encode\_colour} \; a) \]

Can handle recursively defined functions
ACL2 is faster and/or more secure than ML
- computation has higher assurance than ML
- can execute industrial scale models

ACL2 combines a programming language with a logic
- maybe uniquely has this property

HOL can express things hard to express in ACL2
- e.g. the definition of a measurable set

Using ACL2 with HOL enlarges ‘circle of trust’
- but can attach \texttt{ACL2} tag to HOL theorems

Extra trusted code minimised
- HOL, ACL2 assumed trusted
- clean translations \texttt{SEXP-in-HOL} $\leftrightarrow$ \texttt{SEXP-in-ACL2}
Proving ACL2 axioms in HOL4 revealed bugs!

In HOL 4:
- performance issues for strings and parsing bugs for characters
- ask Mike for more details

In ACL2:
- logical (“*1*”) code for primitive function pkg-witness had wrong default value
- ask Matt for more details
The future

- Finish off proving the ACL2 axioms in HOL
  - more than half the axioms are already proved

- ACL2 execution of HOL model of ARM FP coprocessor
  - hand translation done (Reynolds), next do it automatically

- ACL2 execution of HOL model of ARM processor
  - main effort will be deriving ACL2 version of Fox HOL model

- Apply HOL measure theory to ACL2 Miller-Rabin test
  - relate Hurd’s proofs with ACL2 model

- Explore Galois Inc’s SHADE validation example
  - Cryptol semantics in higher order logic rather complex
Questions?
Example axioms proved (1)

("closure_defaxiom",
|- |= andl
  [acl2_numberp (add x y); acl2_numberp (mult x y);
   acl2_numberp (unary_minus x); acl2_numberp (reciprocal x)])

("associativity_of_plus_defaxiom",
|- |= equal (add (add x y) z) (add x (add y z)))

("commutativity_of_plus_defaxiom", |- |= equal (add x y) (add y x))

("unicity_of_0_defaxiom", |- |= equal (add (nat 0) x) (fix x))

("inverse_of_plus_defaxiom", |- |= equal (add x (unary_minus x)) (nat 0))

("associativity_of_star_defaxiom",
|- |= equal (mult (mult x y) z) (mult x (mult y z)))

("commutativity_of_star_defaxiom", |- |= equal (mult x y) (mult y x))

("unicity_of_1_defaxiom", |- |= equal (mult (nat 1) x) (fix x))

("inverse_of_star_defaxiom",
|- |= implies (andl [acl2_numberp x; not (equal x (nat 0))])
  (equal (mult x (reciprocal x)) (nat 1)))

("integer_0_defaxiom", |- |= integerp (nat 0))

("integer_1_defaxiom", |- |= integerp (nat 1))

("car_cons_defaxiom", |- |= equal (car (cons x y)) x)

("cdr_cons_defaxiom", |- |= equal (cdr (cons x y)) y)

("cons_equal_defaxiom",
|- |= equal (equal (cons x1 y1) (cons x2 y2))
  (andl [equal x1 x2; equal y1 y2]))

("booleanp_characterp_defaxiom", |- |= booleanp (characterp x))

("characterp_page_defaxiom", |- |= characterp (chr \'\f\'))

("characterp_tab_defaxiom", |- |= characterp (chr \'\t\'))

("characterp_rubout_defaxiom", |- |= characterp (chr \'\127\'))

("coerce_inverse_1_defaxiom",
|- |= implies (character_listp x)
  (equal (coerce (coerce x (csym "STRING")) (csym "LIST")) x))
Example axioms proved (2)

("coerce_inverse_2_defaxiom",
 |- |- implies (stringp x)
   (equal (coerce (coerce x (csym "LIST")) (csym "STRING")) x))
("character_listp_list_to_sexp",
 |- !l. |- character_listp (list_to_sexp chr l))
("character_listp_coerce_defaxiom",
 |- |- character_listp (coerce acl2_str (csym "LIST")))
("lower_case_p_char_downcase_defaxiom",
 |- |- implies (andl [upper_case_p x; characterp x])
   (lower_case_p (char_downcase x)))
("stringp_symbol_package_name_defaxiom",
 |- |- stringp (symbol_package_name x))
("symbolp_intern_in_package_of_symbol_defaxiom",
 |- |- symbolp (intern_in_package_of_symbol x y))
("symbolp_pkg_witness_defaxiom", |- |- symbolp (pkg_witness x))
("completion_of_plus_defaxiom",
 |- |- equal (add x y)
   (itel
    [(acl2_numberp x,ite (acl2_numberp y) (add x y) x);
     (acl2_numberp y,y)] (nat 0))))
("completion_of_car_defaxiom",
 |- |- equal (car x) (andl [consp x; car x]))
("completion_of_cdr_defaxiom",
 |- |- equal (cdr x) (andl [consp x; cdr x]))
("completion_of_char_code_defaxiom",
 |- |- equal (char_code x) (ite (characterp x) (char_code x) (nat 0))))
("completion_of_denominator_defaxiom",
 |- |- equal (denominator x) (ite (rationalp x) (denominator x) (nat 1))))
("completion_of_imagpart_defaxiom",
 |- |- equal (imagpart x) (ite (acl2_numberp x) (imagpart x) (nat 0))))
Example axioms proved (3)

("completion_of_intern_in_package_of_symbol_defaxiom",
 |- |e = equal (intern_in_package_of_symbol x y)
    (andl [stringp x; symbolp y; intern_in_package_of_symbol x y]))
("completion_of_numerator_defaxiom",
 |- |e = equal (numerator x) (ite (rationalp x) (numerator x) (nat 0)))
("completion_of_realpart_defaxiom",
 |- |e = equal (realpart x) (ite (acl2_numberp x) (realpart x) (nat 0)))
("completion_of_symbol_name_defaxiom",
 |- |e = equal (symbol_name x) (ite (symbolp x) (symbol_name x) (str "")))
("completion_of_symbol_package_name_defaxiom",
 |- |e = equal (symbol_package_name x)
    (ite (symbolp x) (symbol_package_name x) (str "")))
("booleanp_bad_atom_less_equal_defaxiom",
 |- |e = ite (equal (bad_atom_less_equal x y) t)
    (equal (bad_atom_less_equal x y) t)
    (equal (bad_atom_less_equal x y) nil))
("bad_atom_less_equal_antisymmetric_defaxiom",
 |- |e = implies
    (andl
     [bad_atom x; bad_atom y; bad_atom_less_equal x y; bad_atom_less_equal y x]) (equal x y))
("bad_atom_less_equal_transitive_defaxiom",
 |- |e = implies
    (andl
     [bad_atom_less_equal x y; bad_atom_less_equal y z; bad_atom x; bad_atom y; bad_atom z])
    (bad_atom_less_equal x z))
("bad_atom_less_equal_total_defaxiom",
 |- |e = implies (andl [bad_atom x; bad_atom y])
    (ite (bad_atom_less_equal x y) (bad_atom_less_equal x y)
    (bad_atom_less_equal x y)))