An Integration of HOL and ACL2

Michael J.C. Gordon\textsuperscript{1}, Warren A. Hunt, Jr.\textsuperscript{2},
Matt Kaufmann\textsuperscript{2}, James Reynolds\textsuperscript{1}
(alphabetical order)

\textsuperscript{1} Computer Laboratory
William Gates Building, Cambridge CB3 0FD, United Kingdom
E-mail: mjcg@cl.cam.ac.uk, jr291@cam.ac.uk

\textsuperscript{2} Department of Computer Sciences
1 University Station, M/S C0500, Austin TX 78712-0233, USA
E-mail: hunt@cs.utexas.edu, kaufmann@cs.utexas.edu

Abstract. We describe a link between the ACL2 and HOL mechanical proof assistants that enables the strengths of each system to be deployed smoothly within a single formal development. Several application scenarios are being considered: using ACL2’s high performance execution environment by converting HOL models into ACL2; using ACL2’s proof automation to discharge HOL proof obligations; and importing ACL2 definitions and theorems into HOL, to specify and verify properties that cannot easily be stated in the first-order ACL2 logic.

Particular care has been taken to ensure sound translations between the logics supported by HOL and ACL2. The initial ACL2 theory has been defined inside HOL, so that it is possible to prove mechanically that first-order ACL2 functions on S-expressions correspond to higher-order functions operating on a variety of types. The translation between the two systems operates at the level of S-expressions, and has been engineered to allow large models to be converted.

The paper consists of an overview of how the two systems are connected, a summary of the logical HOL theory used to define the initial ACL2 theory, and an example showing how things work in practice.

Keywords: mechanised reasoning, first-order logic, higher-order logic, ACL2, HOL, HOL4, sound translation, proof oracle.

1 Introduction

Higher-order logic (HOL) provides a convenient framework for specification and verification of digital systems. HOL proof assistants (e.g. HOL98 [16,14], HOL4 [13], HOL Light [8], ProofPower [1] and Isabelle/HOL [12]) support HOL, but tools supporting other logics may offer advantages in proof automation. The first-order ACL2 logic with its associated theorem prover [11] is such an example: the ACL2 theorem prover provides superior mechanised support for some kinds of automatic proof (e.g. induction and ground execution of functions).

In what follows we use “HOL” sometimes to refer to higher-order logic and sometimes to proof systems supporting the logic. We use HOL4 when that particular implementation is being referred to. We use “ACL2” to refer both to the system and to the logic it supports.
We have linked ACL2 and HOL4 in a way that enables the strengths of each system to be smoothly deployed within a single formal development. Great care has been taken to ensure that the translations between the higher-order logic of HOL4 and the first-order logic of ACL2 are logically sound.

We have several application scenarios in mind. Initially we are focusing on converting higher-order logic models into first-order logic and then using ACL2 to achieve high performance simulation. An example is given in Section 8.

Another application scenario is importing ACL2 definitions and theorems into HOL and then using a higher-order logic prover such as HOL or Isabelle to prove properties that cannot easily be stated in the first-order ACL2 logic. A concrete instance of this is validating the translation of Cryptol [5] programs to ACL2 currently being undertaken by Galois, Inc. The idea is to start with a Cryptol program, translate it to an ACL2 function definition, import the generated ACL2 definition into HOL and then prove, using the semantics of Cryptol (which is formulated in higher-order logic as it can’t easily be stated in first-order logic) that the translated Cryptol correctly implements the source specification.

The integration infrastructure described here is intended to be application neutral: we avoid biasing design decisions towards any particular application.

In the next two sections we discuss previous work on connecting HOL and Boyer-Moore provers, followed by why we think our work is new and significant. The underlying logical ideas for embedding ACL2 in HOL are then motivated and the theory of ACL2 logic in higher order logic is described. Next we give details, using a simple example, of how HOL is converted to ACL2. The reverse flow is briefly sketched, though we are not focusing on this here. Next comes a simple example illustrating the use of ACL2 to execute HOL, together with runtime data illustrating the kind of performance gains that can be achieved. The paper ends with future work and some conclusions. An appendix describes modelling ACL2 packages, a tricky part of representing the ACL2 logic in HOL.

2 Related work

In 1991, Fink et al. described a proof manager called PM [6] that enabled HOL input to be transformed into “first-order assertions suited to the Boyer-Moore prover”. In 1999 Mark Staples implemented a tool called ACL2PII for linking ACL2 and HOL98 [17]. As far as we know these are the only previous attempts to link HOL to ACL2. ACL2PII was used by Susanto and Melham [18, 19].

Both PM and ACL2PII provided ways of translating between higher-order logic and first-order logic. When translating from untyped Boyer-Moore logic to typed higher-order logic it can be hard to figure out which types to assign. Staples points out that the ACL2 S-expression NIL might need to be translated to $\text{F}$ (boolean type), or $\text{[]}$ (list type) or $\text{NONE}$ (option type), depending on context. The ACL2PII user has to set up “translation specifications” that are pattern-matching rewrite rules to perform the ACL2-to-HOL translation. These are encoded in ML and are thus not supported by any formal validation.

3 See http://www.cl.cam.ac.uk/users/mjcg/hol2acl2/examples for more details.
3 Scientific contribution

Previous links between HOL and Boyer-Moore/ACL2 systems have been open to the criticism that the translation of formulae between the systems may be unsound. Our contribution is to design and implement an approach that provides extremely high assurance that the corresponding HOL and ACL2 formulae have corresponding semantics.

The key idea is to bridge the gap between HOL and ACL2 by defining a theory, SEXP, that consists of an S-expression datatype in HOL together with definitions of ACL2 primitives operating on that datatype.

The correspondence between HOL and SEXP is done with proof in HOL4, so is highly assured. We are developing automated tools to mechanise these proofs, as we illustrate in Section 8.

The correspondence between SEXP and ACL2 is not performed by formal manipulation in a logic, because HOL4 and ACL2 are different systems written in different languages (Standard ML and Common Lisp, respectively). However, we reduce risk by limiting the exchange of data between ML and Lisp to S-expressions. We also intend to prove translations of ACL2 axioms in the SEXP theory (simple axioms have already been proved – see the discussion of the axiom car-cdr-elim in Section 5.2). Finally, we have done some “round-trip” testing, taking ACL2 terms, translating them to SEXP, and then translating them back to ACL2, and checking that the results agree with the original ACL2 terms.

In summary, our approach provides much higher assurance than that provided by PM or ACL2PII because we perform proof-based formal translation between the first-order formulæ of ACL2 and the higher-order logic formulæ of HOL4. The logically tricky parts of the translations are done within a formal framework, namely translation to SEXP within HOL; so we have a high assurance of soundness. Thus the meta-language scripts of PM or ACL2PII are replaced by deductions in the HOL4 system together with a relatively straightforward link between SEXP and ACL2.

4 ACL2: axiomatic theory or interpreter?

Consider the ACL2 axiom ASSOCIATIVITY-OF-* given in the ACL2 source file axioms.lisp: (EQUAL (* (* X Y) Z) (* X (* Y Z))). This can be viewed from two perspectives: (i) as an S-expression in ACL2’s version of Lisp; (ii) as a formula of first-order logic.

Under the first view (i), the axiom is valid because if X, Y and Z are replaced by any S-expressions, then the resulting instance of the axiom will evaluate to
‘true’, i.e., T in Common Lisp. Under the second view, the formula is an axiom that defines what it means for evaluation to be correct: it is a partial semantics of Lisp evaluation. Thus, in order to build a formal model of the ACL2 logic we are faced with deciding whether to take Lisp evaluation or the ACL2 axioms as ‘golden’ – i.e., as the primary specification.

If the first view were adopted, we could try to build a formal model of ACL2’s Lisp evaluation in HOL, so that the ACL2 axioms can be proved consistent with Lisp semantics by proving, e.g. \((\text{EQUAL} (\ast (\ast X Y) Z) (\ast X (\ast Y Z)))\) always evaluates to T. However, the model of Lisp evaluation in HOL would need to be validated with respect to some reference evaluator and it is not clear what this reference should be, since there is no ACL2 definition of S-expression evaluation.

We have decided to adopt the second view (ii), namely that the axioms in the ACL2 source file \texttt{axioms.lisp} define the logic [10], rather than some ‘golden’ evaluator. If there are discrepancies between this and the actual behaviour of ACL2 evaluation (and as far as we know there are none), then our view is that it would be a bug in the evaluator, not in the ACL2 axioms.

Our approach is to define S-expressions in higher-order logic (by defining a type \texttt{sexp}) and then to specify HOL functions operating on this type that correspond to the ACL2 functions axiomatised in \texttt{axioms.lisp} (\texttt{cons}, \texttt{car}, \texttt{cdr}, etc.) and which satisfy the axioms. The key property we must ensure is that for any formula provable in ACL2, its translation is provable in the SEXP theory in HOL. This property guarantees that we can use ACL2 as a trusted oracle.

5 \textbf{SEXP: a theory of the ACL2 logic in HOL}

We define a HOL theory, called SEXP, which includes a type \texttt{sexp} representing S-expressions in higher-order logic and constants corresponding to the ACL2 primitive functions that satisfy the ACL2 logic axioms.

5.1 \textbf{S-expressions in HOL}

The type \texttt{sexp} accurately represents ACL2 S-expressions. It is a recursively defined datatype composed of four kinds of atoms (symbols, strings, characters and complex rational numbers) and pairs of S-expressions.

HOL and ACL2 each have their own notions of characters, strings and numbers. Fortunately the match between characters and strings in HOL and ACL2 is exact. In ACL2, numbers are specified axiomatically in \texttt{axioms.lisp}, which contain axioms like the associativity and commutativity of addition and multiplication. ACL2 complex rational numbers consist of two rationals: a real part and an imaginary part. Rational numbers in HOL are defined as a quotient type [9] using a rational package developed by Jens Brandt [3] and consist of two integers: the numerator and denominator. Thus an ACL2 number can be represented by four integers (real part numerator, real part denominator, imaginary part numerator, imaginary part denominator). The first-order axiomatisation of numbers in ACL2 admits non-standard interpretations, but the higher-order HOL definition of numbers does not – it constrains numbers to be standard.
Thus, there may be properties of numbers in the SEXP layer that cannot be proved in ACL2. We do not view this as a problem as we already know that there are things that can be proved in HOL but not in ACL2.

Not all symbols in the HOL datatype `sexp` correspond to valid ACL2 symbols due to ACL2’s rules for ‘interning’ symbols in packages. This is handled by having an explicit definition of the package structure in HOL and by making the definition of `symbolp` return `nil` on symbols which are not symbols according to this structure. Symbols in ACL2 consist of a package name and a symbol name separated by “::”, but normally only the symbol name is input and output, the package name being implicit via the current package. It simplifies the representation of ACL2 inside HOL if we use fully expanded ACL2 names as the names of the corresponding HOL constants. For convenience when working with HOL we have implemented a mechanism for overloading parser-friendly names onto ACL2 names. For example, `mult`, `add` and `unary_minus` are the parser-friendly HOL names overloaded onto `ACL2::BINARY-` (multiplication), `ACL2::BINARY+` (addition) and `ACL2::UNARY--` (unary minus), respectively. These particular names are used in the example in Section 6. The modelling of the ACL2 package system in HOL, including how HOL and ACL2 names are managed, is described in more detail in the appendix.

5.2 ACL2 primitives in HOL

Certain ACL2 symbols represent primitive notions in the initial theory. Examples are `t`, `nil`, `car`, `cdr`, `cons`, `consp` and `if`. There are 33 such primitive symbols:

- `t nil 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 string symbol-name symbol-package-name symbolp` 

The ACL2 logic constrains the interpretation of these by axioms listed in the file `axioms.lisp`. For example, one of the axioms, called `car-cdr-elim`, is:

```
(defaxiom car-cdr-elim
  (implies (consp x) (equal (cons (car x) (cdr x)) x)))
```

This uses an auxiliary function `implies` defined in terms of the primitives by:

```
(defun implies (p q) (if p (if q t nil) t))
```

We have defined all the ACL2 primitives, and all the auxiliary functions they use, as constants or functions in the HOL theory `SEXP`. The constants `t` and `nil` are names of particular symbols, and the function `cons` is a primitive constructor of the datatype `sexp`. The functions `car` and `cdr` map `sexp` pairs to their first and second components, respectively, and all other `sexp` values (i.e. strings, characters and numbers) to `nil`. The function `consp` maps pairs to `t` and all other `sexp`-values to `nil`. The function `equal` is defined in HOL by:

```
\forall x y. equal x y = if x = y then t else nil
```

where equality (=) and conditional (if-then-else) are those of the HOL logic, and the ACL2 conditional `acl2_if` is defined in `SEXP` by:

```
\forall x y z. acl2_if x y z = if x = nil then z else y
```
Unfortunately, the HOL parser makes it inconvenient to use “if” for the ACL2 conditional in the theory \texttt{SEXP}, because it is a HOL keyword – thus we use “\texttt{acl2\_if}” instead. The auxiliary function \texttt{implies} used in the ACL2 axiom \texttt{car-cdr-elim} is defined in \texttt{SEXP} by:

\[
\forall p, q. \quad \texttt{implies} p q = \texttt{acl2\_if} p (\texttt{acl2\_if} q t \texttt{nil}) t
\]

To ensure that our definitions of the ACL2 primitives are sound, we have proved many of the axioms in \texttt{axioms.lisp} (and we intend to prove them all eventually). ACL2 formulae are S-expressions, but HOL formulae are terms of type \texttt{bool}. When an ACL2 term \texttt{p} is used as a formula it means that \texttt{p} is not \texttt{nil}, thus we define the HOL formula \( \models \texttt{p} \) by:

\[
\forall p. \quad \models p \iff (p \neq \texttt{nil})
\]

to mean that \( p \) is true. The axiom \texttt{car-cdr-elim} is then verified in the HOL model of ACL2 by proving:

\[
\forall x. \quad \models \texttt{implies} (\texttt{consp} x) (\texttt{equal} (\texttt{cons} (\texttt{car} x) (\texttt{cdr} x)) x)
\]

The first-order logic of ACL2 is not typed, but the higher-order logic of HOL is. All constants defined in HOL must have a fixed type. For example, the HOL types of the functions in the theory \texttt{SEXP} described above are:

<table>
<thead>
<tr>
<th>HOL constants</th>
<th>HOL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{t}, \texttt{nil}</td>
<td>\texttt{sexp}</td>
</tr>
<tr>
<td>( \models )</td>
<td>\texttt{sexp} \rightarrow \texttt{bool}</td>
</tr>
<tr>
<td>\texttt{car}, \texttt{cdr}</td>
<td>\texttt{sexp} \rightarrow \texttt{sexp}</td>
</tr>
<tr>
<td>\texttt{cons}, \texttt{equal}, \texttt{implies}</td>
<td>\texttt{sexp} \rightarrow \texttt{sexp} \rightarrow \texttt{sexp}</td>
</tr>
<tr>
<td>\texttt{acl2_if}</td>
<td>\texttt{sexp} \rightarrow \texttt{sexp} \rightarrow \texttt{sexp} \rightarrow \texttt{sexp}</td>
</tr>
</tbody>
</table>

### 6 Encoding HOL developments into ACL2

We have implemented a set of tools that can take a sequence of definitions in higher-order logic in the HOL4 system and create a sequence of definitions in \texttt{SEXP}, together with a proof that the encoding of the definitions is correct. The coding and decoding of HOL functions is performed recursively using the encodings and decodings of sub-functions, starting from a Library of encodings and decodings of primitive functions (e.g. arithmetic and list-processing).

On some examples the tool is automatic, but currently its output sometimes needs additional hand proof in HOL (represented in the diagram in Section 3 by the box entitled “Manually adjusted first-order ACL2 logic in HOL”). We expect the tool eventually to be fully automatic.

In this section we show the flow from higher-order logic to ACL2 with a simple example: exponentiation on natural numbers. In Section 8 we have another example, together with figures illustrating the performance benefits of using ACL2 for simulation. Exponentiation is defined recursively in HOL by a conjunction of equations, one for the basis and one for the inductive step (in what follows we are defining an infix \texttt{**} using HOL’s built in multiplication \texttt{*} and the successor function \texttt{SUC} on natural numbers).

\[
(\forall m.\ m \times 0 = 1) \land (\forall m.\ m \times SUC n = m \times n + (m \times n))
\]

The first step is to use the encoding tools to generate a definition of a constant \texttt{acl2\_EXP} that represents \texttt{**} as a first-order operation on S-expressions in HOL.
\textit{An Integration of HOL and ACL2}  

\[ \forall m \, n. \ (\text{acl2\_EXP}\ m \ n = \text{acl2\_if} \ (\text{zp} \ n) \ (\text{nat} 1) \ (\text{mult} m \ (\text{acl2\_EXP} m \ (\text{add} n \ (\text{unary\_minus} (\text{nat} 1)))))) \]

together with the theorem:

\[ \vdash \forall m \, n. \ m \ **\ n = \text{sexp\_to\_nat} \ (\text{acl2\_EXP} (\text{nat} m) (\text{nat} n)) \]

The generated definition of \text{acl2\_EXP} uses a HOL version of the ACL2 predicate \text{zp} (tests if an S-expression is an integer). The functions \text{nat} and \text{sexp\_to\_nat} implement the encoding of HOL numbers as S-expressions: \text{nat} maps a HOL number (of type \text{num}) to an S-expression (type \text{sexp}), and \text{sexp\_to\_nat} is the inverse. These functions appearing in the definition of \text{acl2\_EXP} are generated using the library of pre-proved encoding theorems.

To transfer the definition to ACL2, an ML function prints the ACL2-style definition of \text{acl2\_EXP} to a file in the following syntax:

\[(\text{COMMON-LISP::DEFUN} \text{ACL2::acl2\_EXP} (\text{ACL2::m} \text{ACL2::n}) \]
\[ (\text{COMMON-LISP::IF} \]
\[ (\text{ACL2::ZP} \text{ACL2::n}) \]
\[ 1/1 \]
\[ (\text{ACL2::BINARY\_\*} \text{ACL2::m}) \]
\[ (\text{ACL2::acl2\_EXP} \text{ACL2::m} (\text{ACL2::BINARY\_\*} \text{ACL2::m} (\text{ACL2::BINARY\_\-}\text{ACL2::n} (\text{ACL2::UNARY\_\-} 1/1))))) \]

Full ACL2 names are generated, and the default package name (ACL2) is used for all symbols. We have designed the system to support a general hierarchy of packages in HOL, but currently anticipate that when moving developments from HOL to ACL2 we will stay within package ACL2. If the \text{DEFUN} above is printed using ACL2 then one sees:

\[(\text{DEFUN} \text{ACL2\_EXP} (M N) \]
\[ (\text{IF} (\text{ZP} N) 1 (\text{BINARY\_\*} M (\text{ACL2\_EXP} M (\text{BINARY\_\-} N (\text{UNARY\_\-} 1))))) \]

The transfer of definitions of functions on S-expressions in HOL to definitions in ACL2 is not audited by any proof (represented by the dotted arrow labelled “\text{trusted code translating ML and LISP S-expressions}” in the diagram in Section 3). It is vital that the translation code be trustworthy. We have attempted to achieve this by keeping it straightforward.

\section{Encoding ACL2 developments into HOL}

We have implemented tools for importing ACL2 developments into HOL. The first step is to use ACL2 to translate to ACL2 internal form, for example expanding all macros (see Section 9) and expanding LETs to LAMBdas. A function written in ACL2 then prints a file containing the expanded ACL2 terms in a form that can be read into HOL.

The system allows users either to trust ACL2 or to create goals in HOL to validate ACL2 axioms or theorems. Theorems that are created in the SEXP theory by trusting ACL2 are ‘tagged’ with their source, so users can always distinguish theorems proved entirely inside HOL from those proved using ACL2 as an oracle. A particular case of this is when ACL2 definitions are imported into HOL. ACL2 will only admit a definition if it can prove that it is sound. HOL has a similar discipline for definitions. Trusting ACL2 definitions means trusting that they are sound in HOL. We believe that any definition admitted by
ACL2 could also be soundly admitted in HOL (because HOL provides induction
support at least as strong as the \( \epsilon_0 \)-induction used by ACL2).

As a check, we have imported several hundred ACL2 axioms, definitions and
theorems from \texttt{axioms.lisp} and from an ACL2 model of Y86 (a very small X86-
like machine [4]) into HOL, converted them into HOL theorems and definitions,
then exported them back to ACL2, and successfully checked that the results are
correct. This substantially adds to our confidence that our tools are sound. Note
that this “round trip” test just helps validate the conversions between ML and
Lisp, it doesn’t prove that the axioms follow by proof in HOL in the \texttt{SEXP}
thephyory (doing that is work in progress).

8 Example

The example in this section shows the kind of performance that can be attained
by using ACL2 to execute HOL specifications. It is chosen to be small enough
so that we can give a complete description here.

We consider a simple memory model that can interpret ‘read’ and ‘write’
instructions of the form \((b, x, y)\) where \(b\) is a Boolean (T for a write, F for a
read), \(x\) is an address (a natural number) and \(y\) a value to write (ignored by
reads). The following two functions construct instructions:

\begin{verbatim}
Define 'read x = (F, x, 0)
Define 'write (x,y) = (T, x, y)
\end{verbatim}

The state of a memory is represented by a list \([(a1,v1);\ldots;(an,vn)]\)
where \(a1,\ldots,an\) are the addresses holding non-zero values, and the value stored
at \(ai\) is \(vi\) \((1 \leq i \leq n)\) – thus an empty list \([\]\) represents a memory in which all
addresses hold 0.

The function \texttt{read_step} takes an address and a memory state and returns
the value stored at the address. This is defined in HOL as follows (the double
colon “::” is HOL’s infixed list-cons operation – don’t confuse it with ACL2’s
use to separate package names from symbol names in symbols):

\begin{verbatim}
Define '(read_step addr [] = 0)
  \wedge
  (read_step addr ((addr,v)::alist) =
   if addr = addr' then v else read_step addr alist)
\end{verbatim}

The function \texttt{write_step} takes an address, a value and a memory state and
returns an updated memory state with the value written to the address.
The following function \texttt{run} takes a list of instructions, a memory state
and an accumulator consisting of a list of previously read values (in reverse
order). It then executes the instructions, adding any values read to the front of
the accumulator.

\begin{verbatim}
Define '(write_step addr v [] = [(addr,v)])
  \wedge
  (write_step addr v ((addr',v')::alist) =
   if addr = addr'
    then (addr,v)::alist
    else (addr',v')::(write_step addr v alist))
\end{verbatim}
An Integration of HOL and ACL2

Define

\[(\text{run [\[] \text{alist reversed_values_read} = \text{reversed_values_read})\]
\]
\[
(\text{run ((T,addr,v)::instrs) \text{alist reversed_values_read} =}
\]
\[
\text{run instrs (write_step addr v \text{alist}) reversed_values_read})\]
\]
\[
(\text{run ((F,addr,v)::instrs) \text{alist reversed_values_read} =}
\]
\[
\text{run instrs \text{alist ((read_step addr \text{alist}):reversed_values_read))}\]
\]

We now define functions to generate some sequences of instructions for testing. We will use memories with addresses less than max_addr, which we define to be 100.

Define ‘max_addr = 100’

We also define two constants that indicate how much to increment the address and value when generating the next instruction in a test.

Define ‘write_increment = 13’
Define ‘read_increment = 17’

and we define a function for fixing (by ‘wrapping-around’) the address if we try to generate one bigger than the limit (in the definition below a is an address and when fix_address is called b will be max_addr).

Define

\['\text{fix_address a b = if } a \geq b \text{ and } (0 < b) \text{ then fix_address (a - b)} \text{ b else a}’\]

The function make_instrs takes an initial value for generating reads and write, and a flag to determine whether to generate a read or write next (the flag alternates, so reads and writes alternate in the test programs generated).

Define

\['\text{make instrs read_start write_start flag 0 acc = acc)}\]
\[
(\text{make instrs read_start write_start flag (SUC n) acc =}
\]
\[
\text{if flag then make instrs}
\]
\[
\text{read_start}
\]
\[
(\text{fix_address (write_start + write_increment) max_addr})
\]
\[
(\text{‘flag})
\]
\[
\text{n}
\]
\[
(\text{write(write_start,SUC n)::acc})
\]
\[
\text{else make instrs}
\]
\[
(\text{fix_address (read_start + read_increment) max_addr})
\]
\[
\text{write_start}
\]
\[
(‘\text{flag})
\]
\[
\text{n}
\]
\[
(\text{read read_start::acc}})\]

The little HOL development above is converted, mostly automatically, into the following ACL2 development (shown with all package names on symbols removed by ACL2):

(defun acl2_read_step (addr naddr)
  (if (consp naddr)
      (if (equal addr (car (car naddr)))
          (cdr (car naddr))
          (acl2_read_step addr (cdr naddr)))
    0))
\begin{verbatim}
(defun acl2_write_step (addr v naddr)
  (if (consp naddr)
    (if (equal addr (car (car naddr)))
      (cons (cons addr v) (cdr naddr))
    (cons (cons (car (car naddr)) (cdr (car naddr)))
      (acl2_write_step addr v (cdr naddr))))
  (cons (cons addr v) nil)))
(defun acl2_run (addr alist reversed_values_read)
  (if (consp addr)
    (if (car (car addr))
      (acl2_run (cdr addr)
        (acl2_write_step (car (cdr (car addr)))
          (cdr (cdr (car addr)))
          alist)
        reversed_values_read)
    (acl2_run (cdr addr)
      alist
      (cons (acl2_read_step (car (cdr (car addr))) alist)
        reversed_values_read))
  reversed_values_read))
(defun acl2_write_increment () 13)
(defun acl2_read_increment () 17)
(defun acl2_max_addr () 100)
(defun acl2_read (x) (cons nil (cons x 0)))
(defun acl2_write (x y) (cons t (cons x y)))
(defun acl2_fix_address (a b)
  (if (if (natp a)
    (if (natp b) (if (not (< a b)) (< 0 b) nil) nil)
    (acl2_fix_address (nfix (+ a (- b))) b)
    a))
(defun acl2_make_instrs (read_start write_start flag n acc)
  (if (if (natp n) (not (equal n 0)) nil)
    (if flag
      (acl2_make_instrs read_start write_start flag
        (+ n (- 1))
        (cons (acl2_write write_start (+ (+ n (- 1)) 1))
          acc))
      (acl2_make_instrs
        (acl2_fix_address (+ read_start (acl2_read_increment))
          (acl2_max_addr))
        write_start (if flag nil t)
        (+ n (- 1))
        (cons (acl2_read (acl2_read start) acc))
        acc))
  acc))

As a benchmark we create a program with a million instructions (by in-
voxing \texttt{make_instrs}) and then run the the program. Specifically we eval-
uate: \texttt{run (make_instrs 0 10 F 1000000 \[] \[] \[] in HOL and ML, and the}
translation to S-expressions in ACL2. The results shown below should be taken
as illustrative only, since they involve three different systems (Moscow ML, ML-
ton [20] and ACL2), and it is not clear if exactly the same things are being timed:
the ML compilers report \texttt{"gc"}, \texttt{"sys"} and \texttt{"usr"} times (figures given below are
\end{verbatim}
An Integration of HOL and ACL2

Evaluation by proof in HOL (EVAL) takes too long to measure. Translating to ML and using the standard Moscow ML interpreter is a little faster than using the ACL2 read-eval-print loop on the compiled functions above and about the same as running them in Common Lisp\(^4\). The fastest execution is with a manually verified translation to single-threaded code \([2]\). The MLton ML compiler is the most optimising ML compiler we know, but it is more than an order of magnitude slower than single-threaded ACL2. Execution in ACL2 is within a theorem prover and so the results are based on formal definitions in the logic, unlike those from the ML compilers.

9 Future Work

We have proved several of the ACL2 axioms from the HOL definitions of the ACL2 primitives, and our plan is to prove all of them.

So far we have concentrated on using ACL2 to execute HOL models. The biggest example done so far is the floating point evaluation function that forms part of an ARM co-processor \([15]\). This executed about 300 times faster in ACL2 than using HOL's native execution facility (EVAL).

We are planning to build a high assurance ACL2 simulation platform for running a complete model of an ARM processor that already exists in HOL \([7]\). This will be linked to the floating point co-processor. Using this platform, it is hoped that we can validate the HOL ARM model against ARM's internal simulator and against actual ARM hardware.

We want to investigate translating HOL goals, together with the definitions they depend on, into SEXP. Once the problem has been reformulated into SEXP, we export it to a file of ACL2 S-expressions, in the form of an ACL2 book that may be processed by ACL2. An ACL2 user then creates an ACL2 book that includes the one that was generated and perhaps additional intermediate definitions and lemmas. After ACL2 has been led to validate the extended book, it will be trivial to certify the generated book, at which point the original SEXP goal can be declared to have been solved and an ACL2-tagged SEXP theorem achieving the goal is created. For this to be sound, we need to be sure that the original goal is a theorem of HOL, but we are confident this is true because the theorems about S-expressions provable in the HOL theory SEXP are a superset of the theorems provable in ACL2. Note that in ACL2 there is an implicit assumption

\[\text{gc+sys+usr times}, \] whilst ACL2/LISP reports “real”, “run-gbc” and “gbc” times (figures given below are real times). All timings were done on the same machine.

<table>
<thead>
<tr>
<th>HOL EVAL</th>
<th>Moscow ML</th>
<th>MLton</th>
<th>ACL2</th>
<th>Common Lisp</th>
<th>ACL2+stobj</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞?</td>
<td>5.8</td>
<td>1.26</td>
<td>8.16</td>
<td>5.6</td>
<td>0.06</td>
</tr>
</tbody>
</table>

\(^4\) The “Common Lisp” time comes from inserting hand-generated guards into the ACL2 functions and evaluating with those functions in the ACL2 loop; but those functions immediately call their compiled Common Lisp counterparts.
that all instances of induction up to the ordinal $\varepsilon_0$ are available. The HOL logic is sufficiently strong that these instances are provable in SEXP.

Our current treatment of macros may not work for importing a large ACL2 development, say the JVM, into SEXP because elimination of macros can cause significant expansion in code size and loss of mnemonic abstractions. By defining appropriate auxiliary HOL definitions or ML functions, it may be possible to provide better support in SEXP for importing macros, but for current work, expanding them seems adequate. Macros require more thought and a case study to identify technical issues.

10 Conclusion

Connecting ACL2 and HOL is not new, nor is using such a connection for fast execution: both Staples and Susanto (see previously cited papers) used ACL2 as an oracle for executing processors specified in HOL. Our contribution is a new methodology for linking the two systems that minimises the amount and complexity of trusted code. The translation of higher-order logic into an embedded ACL2 logic is new, and enables complex encoding scripts of previous approaches to be replaced by proof-audited formal translation. No simplifying assumptions about the ACL2 logic have been made – the entire logic is modelled with very high accuracy. We believe that if you trust ACL2 and HOL4 then you should be able to trust shared developments using the systems linked according to our approach, which is designed to scale robustly to large developments.

11 Acknowledgements

We have had discussions in person or by email with many people about this work, including Bob Boyer, John Matthews, Pete Manolios, J Moore and Mark Shields. Konrad Slind has made major contributions to the ideas and also to our tools. In particular, he helped us implement acl2Define.

References


Appendix A: Modelling Packages

The ACL2 logic includes a package system [10] that provides name spaces. Here we give a brief summary of the modelling of that system in HOL.

Symbols in ACL2 have a package name and a symbol name. The package name of basic Lisp primitives is often "COMMON-LISP", whilst the package name
of other ACL2 primitives is "ACL2" (distinct symbols may have the same symbol name if their package names differ). A development in ACL2 may introduce a hierarchy of packages, but there is always a current package in which one is working (the default being the package named "ACL2"). If a symbol is present in the current package, then the symbol is typically input and output to ACL2 without printing its package name. For example, symbol car is typically printed as CAR; ACL2, following COMMON LISP, is case-insensitive. Its package name is "COMMON-LISP", and this symbol is imported into the "ACL2" package, which justifies printing it without its package name. But if the current package is a package that does not import this symbol, then this symbol will be printed as COMMON-LISP::CAR. Logically, every symbol has a package name, so in the HOL theory SEXP we must represent package names.

Although the HOL logic allows constants to have arbitrary strings as names, the parser\(^5\) requires names to be simple (a letter followed by a sequence of letters, numbers, prime characters or underscores). In particular, the HOL parser doesn’t allow names to contain hyphens or colons, both of which occur frequently in ACL2 names. In order to define SEXP versions of ACL2 constants and functions with the same names we provide a tool acl2Define to define a constant with an arbitrary string as its name. Simple examples of its use are

\[
\begin{align*}
\text{acl2Define } "\text{COMMON-LISP::CAR}" & \quad \left( \text{car } (\text{cons } \_ \_ ) = x \right) \land \left( \text{car } \_ = \text{nil} \right) \\
\text{acl2Define } "\text{COMMON-LISP::CONSP}" & \quad \left( \text{consp } (\text{cons } x \_ y) = \text{t} \right) \land \left( \text{consp } \_ \_ = \text{nil} \right) \\
\text{acl2Define } "\text{COMMON-LISP::CDR}" & \quad \left( \text{cdr } (\text{cons } \_ y) = y \right) \land \left( \text{cdr } \_ \_ = \text{nil} \right) \\
\text{acl2Define } "\text{COMMON-LISP::EQUAL}" & \quad \text{equal } x y = \text{if } x = y \text{ then } \text{t} \text{ else } \text{nil}
\end{align*}
\]

The first of these definitions defines a constant called COMMON-LISP::CAR (the ML string supplied as the first argument to acl2Define) by giving a conjunction of equations (the second argument to acl2Define) that looks like it’s defining a constant car. In fact, acl2Define substitutes the string COMMON-LISP::CAR for car in the equations before making the definition in HOL4. After the definition is made acl2Define declares car as an abbreviation for the constant just defined (i.e. COMMON-LISP::EQUAL) using HOL4’s overloading facility. Each ACL2 function or constant is always given an ACL2-style name of the form packageName::symbolName together with a user-specified parser friendly HOL name which is overloaded onto the ACL2 name.

The ACL2 package system is represented in HOL with a function BASIC_INTERNAL which takes a symbol name and a package name and returns an S-expression. An ACL2 theory associates each package name with a list of imported symbols. For example, consider the ACL2 form (defpkg "FOO" '(A B)), where A and B are in the "ACL2" package. This defines an ACL2 package named "FOO" that imports symbols A and B, represented in HOL as sym "ACL2" "A" and sym "ACL2" "B" (where sym:string->string->sexp is the HOL constructor for symbols).

\(^5\) HOL terms can be created by parsing a concrete syntax string or with term constructor functions. Some terms can only be constructed using constructor functions.
Let us turn now to the definition of \textit{BASIC\_INTERN}. If \texttt{pkg\_name} is the name of a known package and \texttt{symbol\_name} is the name of a symbol imported into that package from some other package, named \texttt{old\_pkg}, then:

\[
\text{BASIC\_INTERN} \texttt{symbol\_name pkg\_name} = (\texttt{sym old\_pkg symbol\_name})
\]

For example, given the example above, \texttt{BASIC\_INTERN "A" "FOO"} evaluates to \texttt{sym "ACL2" "A"}. Otherwise, if \texttt{pkg\_name} is the name of a known ACL2 package, then:

\[
\text{BASIC\_INTERN} \texttt{symbol\_name pkg\_name} = (\texttt{sym pkg\_name symbol\_name})
\]

Finally, if \texttt{pkg\_name} is not the name of a known ACL2 package, we return an arbitrary value.

An ACL2 data structure represented via a constant defined in HOL is used to help with the definition of \textit{BASIC\_INTERN}. The constant \texttt{ACL2\_PACKAGE\_ALIST} contains a list of triples:

\[
\texttt{(symbol\_name , known\_pkg\_name , actual\_pkg\_name)}
\]

The initial value, imported from ACL2, contains 2717 triples:

\[
\texttt{|- ACL2\_PACKAGE\_ALIST} = \\begin{array}{l}
(("ALLOW-OTHER-KEYS","ACL2","COMMON-LISP")); \\
("*PRINT-MISER-WIDTH*","ACL2","COMMON-LISP")); \\
("&AUX","ACL2","COMMON-LISP")); \\
\ldots \\
("ZEROP","ACL2-USER","COMMON-LISP"); \\
("A","FOO","ACL2"); \\
("B","FOO","ACL2"); \\
("C","BAR","FOO"); \\
("D","BAR","ACL2"); \\
("NIL","BAR","COMMON-LISP") : thm
\end{array}
\]

The idea is that when \texttt{symbol\_name} is interned into \texttt{known\_pkg\_name}, the resulting symbol's package name is \texttt{actual\_pkg\_name}. That is, the symbol with name \texttt{symbol\_name} and package-name \texttt{actual\_pkg\_name} is imported into package \texttt{known\_pkg\_name}. If we define \texttt{LOOKUP y [(x1,y1,z1);...;(xn,yn,zn)] x} to return \texttt{zi} if \texttt{x=x1} and \texttt{y=y1}, and to return \texttt{y} otherwise, then \textit{BASIC\_INTERN} is defined by:

\[
\text{BASIC\_INTERN} \texttt{sym\_name pkg\_name} = \\
\texttt{sym sym\_name (LOOKUP pkg\_name ACL2\_PACKAGE\_ALIST sym\_name)}
\]

We then define the notion of an ACL2 symbol by:

\[
\text{acl2Define "COMMON-LISP::SYMBOLP"} \\
\texttt{`symbolp (sym p n) = if BASIC\_INTERN n p = sym p n then t else nil`}
\]

The function \texttt{symbolp} tests if a value of type \texttt{sexp} constructed in HOL using the constructor \texttt{sym} represents a valid symbol in the package structure defined by \texttt{ACL2\_PACKAGE\_ALIST}. The initial value of this constant is obtained by importing the corresponding data-structure from ACL2 using our importing tools. The current package is represented in HOL with an ML reference variable (initialised to the string "ACL2").