An Integration of HOL and ACL2

Michael J.C. Gordon, James Reynolds
University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
mjcg@cl.cam.ac.uk, jr291@cam.ac.uk

Warren A. Hunt, Jr., Matt Kaufmann
Department of Computer Sciences
1 University Station, M/S C0500
Austin TX 78712-0233, USA
hunt@cs.utexas.edu, kaufmann@cs.utexas.edu

Abstract

A link between the ACL2 and HOL4 proof assistants is described. This allows each system to be deployed smoothly within a single formal development. Several applications are being considered: using ACL2’s execution environment for simulating HOL models; using ACL2’s proof automation to discharge HOL proof obligations; and using HOL to specify and verify properties of ACL2 functions that cannot easily be stated in the first-order ACL2 logic.

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 is intended to handle large hardware and software models.

1. Introduction

Higher-Order Logic (HOL) and First-Order Logic (FOL) are both used to model hardware and software. Separate verification communities have evolved based on each kind of logic (e.g. a verification group in one major processor company uses higher-order logic, whilst the corresponding group in a competitor company uses first-order logic). There are projects in progress that use models in both logics (e.g. the Cryptol/AAMP7 project at Rockwell Collins and Galois, Inc., which is discussed briefly later). In this paper we describe a method of linking the HOL4 [21] and ACL2 [16] proof assistants in a way that enables the strengths of each system to be smoothly deployed within a single formal development and the resulting verifications to be evaluated to very high levels of assurance. Our work links two particular systems, but the approach is intended to be portable: we are investigating linking Isabelle/HOL [20] and ACL2.

The key idea is a HOL theory, SEXP, that bridges the gap between the HOL4 and ACL2 logics. HOL4 developments are mapped to SEXP developments and the relationship verified by proof. SEXP developments closely correspond to ACL2 developments and can be converted to them by simple reading and writing of files containing S-expressions.

Our initial motivation was to use ACL2 for high performance simulation of HOL models (see Section 7). A different kind of application is to use higher-order logic to specify properties that cannot easily be stated in the first-order ACL2 logic. An instance of this is validating the translation of Cryptol [4] programs to AAMP7 [26] binary code whose semantics are defined in ACL2, as currently being undertaken by Galois, Inc. and Rockwell Collins [10, 23]. 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, that the translated Cryptol program correctly implements the source specification. Another example, which we are working on in collaboration with Joe Hurd, is to verify that an ACL2 function implementing a probabilistic primality test satisfies a specification based on concepts from measure theory [13]. ACL2 executes this algorithm very fast, but it is hard to express its specification in the ACL2 logic.

In the next section we discuss previous work on connecting HOL and Boyer-Moore provers. The underlying logical ideas for embedding ACL2 in HOL are then motivated and the theory of the ACL2 logic in higher order logic is described. We then give details, using simple examples, of how we convert between HOL and ACL2. 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.

We use “HOL” both to refer to higher-order logic and to proof systems supporting the logic. We use “HOL4” when that particular implementation is intended. We use “ACL2” for both the system and the logic it supports.
2. Related work

In 1991, Fink et al. described a proof manager 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 [28]. As far as we know these are the only previous attempts to link HOL to ACL2. ACL2PII was used by Susanto and Melham [29, 30].

Both PM and ACL2PII translate 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 `F` (boolean type), or `[ ]` (list type) or `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.

The 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 high assurance that the corresponding HOL and ACL2 formulae have equivalent semantics. Our approach provides higher assurance than that provided by PM or ACL2PII because we perform proof-based formal translation between the higher-order logic formulae of HOL4 and the first-order formulae of ACL2. The logically tricky parts of the translations are done within a formal framework, namely translation to SEXP within HOL. Thus the meta-language scripts of PM or ACL2PII are replaced by deductions in the HOL4 system together with a clean and semantically simple link between SEXP and ACL2.

There has been lots of work on connecting together other proof assistants. Felt and Howe [5] import HOL90 theories into Nuprl. The theory justifying this is sophisticated (the two logics are fairly different) and the link is only one way. More recently, Mason and Talcott linked the Maude rewriting system to PVS [18] using an architecture based on the Actor model of computation. Both of these linkings use complex meta-theory to justify the soundness of transferring formulae between logics. In contrast, our approach mechanically checks the semantically tricky parts of the linkage (higher-order logic to SEXP). The actual data transfer between HOL4 and ACL2 uses terms (S-expressions) with identical structure and meanings in each system.

Recently, motivated by the Flyspeck project [31], tools have been implemented [19] to move Isabelle/HOL developments into HOL Light [11] and HOL Light developments into Isabelle/HOL [22]. HOL light is a proof assistant for exactly the same logic as the one supported by HOL4, but it has slightly different proof infrastructure and is implemented in O’Caml rather than Standard ML. Isabelle/HOL [20] is a proof assistant for a significantly more complex and expressive version of higher order logic than the one supported by HOL4 and HOL Light (e.g. it has type classes). Obua and Skalberg’s paper provides a framework for importing HOL4 and HOL Light proofs into the Isabelle/HOL logic and then replaying them inside the Isabelle/HOL system. No trusting of the source system (HOL4 or HOL Light) is needed. A proof recording mechanism generates proof scripts that can be replayed. This direction is logically easy as the logic being imported is a subset of the Isabelle/HOL logic. The paper by McLaughlin describes a transfer of proofs going the other way (Isabelle/HOL to HOL Light) and shows how to generate ML code for Isabelle/HOL type classes than can replay class instance proofs inside HOL Light, using ML functors to elaborate type class instances. The work in these two papers bridges a smaller semantic gap than that from HOL to ACL2, but accomplishes more, namely the translation of proofs as well as formulae. HOL4, HOL Light and Isabelle/HOL are all LCF-style tactic-based systems, so share the same proof scripting methodology. The proof development methodologies of HOL and ACL2 are not at all the same, so harder to link. Developing tools to mechanise the replaying of HOL proofs in ACL2, and vice versa is a topic for future research.

The linking of proof assistants to automatic tools, like decision procedures, automatic theorem provers and model checkers, has been studied extensively. One approach is to embed the logic of the automatic tool in the logic of the proof assistant. This may be easy as the former logic might just be a subset of the latter logic. For example, SAT solvers operate on propositional logic, which is a subset of first order logic or higher order logic. However, in some cases the automatic tool has a specialised logic. Examples of this are model checkers, which decide properties of Kripke structures (the models) expressed in temporal logic. One approach to linking these is to build a theory representing the formulae of the specialised logic in the proof assistant logic, which is usually some kind of higher order logic. The languages used for model checking (Kripke structure notations for models and temporal logic for properties) can easily be defined in higher order logic. Problems solvable by model checking can then be converted (e.g. by rewriting) into formulae in the embedded checkable language, and then exported to external model checkers, which are used as trusted oracles. An early pioneering example linked PVS to a symbolic mu-calculus checker [24] and more recently SMV has been linked to HOL4 [32]. This approach to linking model checkers to proof assistants is similar to the way we are linking HOL4 and ACL2. However, the details and general flavour are different since the logics of HOL4 and ACL2
are both general purpose specification languages. It is easy to define Kripke structures and temporal logic operators in higher order logic, but it turns out to be quite complex and tricky to embed the entire ACL2 logic in HOL.

3. ACL2: axiomatic theory or interpreter?

Consider the ACL2 axiom \textsc{Associativity-of-\ast} occurring in the ACL2 source file \texttt{axioms.lisp}:

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

This can be viewed as an S-expression in ACL2’s version of Lisp, or as a formula of first-order logic.

Under the first view the axiom is valid because if \( X, Y \) and \( \ast \) 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, for example, proving:

\[
(\text{equal} \ (\ast \ (\ast \ X \ Y) \ Z) \ (\ast \ X \ (\ast \ Y \ Z))) \text{ always evaluates to } T.
\]

However, the model of Lisp evaluation in HOL would need to be validated against 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, namely that the axioms in the ACL2 source file \texttt{axioms.lisp} define the logic [14], 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 HOL type \texttt{sexp} and then to specify HOL functions operating on this type that correspond to the ACL2 functions axiomatised in \texttt{axioms.lisp} (cons, car, cdr, etc.). 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 for HOL. The property follows from a standard theory interpretation argument: the axioms of SEXP are direct translations of axioms of ACL2, and the rules of inference in HOL are powerful enough to model the first-order and induction rules of inference of ACL2.

4. 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. The type \texttt{sexp} is a recursively defined datatype composed of four kinds of atoms (symbols, strings, characters and complex rational numbers) and pairs of S-expressions. Further details about SEXP can be found in a companion paper [9].

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 [12] using a rational package developed by Jens Brandt [2] 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 theory 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 \texttt{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 symbols 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::\texttt{BINARY-\ast} (multiplication), ACL2::\texttt{BINARY-\ast} (addition) and ACL2::\texttt{UNARY--} (unary minus), respectively. These particular names are used in the example in Section 5.

Certain ACL2 symbols represent primitive notions in the initial theory. Examples are \texttt{t}, \texttt{nil}, \texttt{car}, \texttt{cdr}, \texttt{cons}, \texttt{consp} and \texttt{if}. There are 33 such primitive symbols:
The ACL2 logic constrains the interpretation of these by about eighty axioms listed in the file axioms.lisp. For example, one of the axioms, called car-cdr-elim, is:

\[
\begin{align*}
& \text{(defaxiom car-cdr-elim)} \\
& \quad \text{(implies (consp x))} \\
& \quad \text{(equal (cons (car x) (cdr x)) x))}
\end{align*}
\]

This uses an auxiliary function \text{implies} defined in terms of the primitives by:

\[
\begin{align*}
& \text{(defun implies (p q) (if p (if q t nil) t))}
\end{align*}
\]

We have defined all the ACL2 primitives, and all the auxiliary functions they use, as constants or functions in the HOL theory. Unfortunately, the HOL parser makes it inconvenient where equality (\text{=} ) and the conditional are those of the HOL logic. Unfortunately, the HOL parser makes it inconvenient to use “if” for the ACL2 conditional on S-expressions in the theory SEXP, because it is a HOL keyword (as illustrated above), so we use “ite” (for “if-then-else”) instead:

\[
\begin{align*}
& \forall x y. \ \text{ite x y z} = \text{if x = y then z else y}
\end{align*}
\]

The auxiliary function \text{implies} used in the ACL2 axiom \text{car-cdr-elim} is defined in SEXP by:

\[
\begin{align*}
& \exists p q. \ \text{implies p q = ite p (ite q t nil) t}
\end{align*}
\]

ACL2 formulae are S-expressions, but HOL formulae are terms of type \text{bool}. When an ACL2 term \text{p} is used as a formula it means that \text{p} is not \text{nil}, thus we define the HOL formula \text{\models p} by:

\[
\begin{align*}
& \forall p. \ \text{\models p} = \neg (p = \text{nil})
\end{align*}
\]

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

\[
\begin{align*}
& \forall x. \ \text{\models (consp x)} \\
& \quad \text{(equal (cons (car x) (cdr x)) x)}
\end{align*}
\]

To ensure that our definitions of the ACL2 primitives are sound, we have proved many of the axioms in axioms.lisp (we plan to prove them all eventually).

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 SEXP described above are:

**HOL constants**

| t, nil | sexp |
| car, cdr | sexp<--bool |
| cons, equal, implies | sexp<--sexp<--sexp |
| ite | sexp<--sexp<--sexp<--sexp |

**5. Encoding HOL developments into ACL2**

We have implemented a set of tools that can take a sequence of definitions in HOL and create a sequence of definitions in the theory SEXP, together with a proof that the encoding of the definitions is correct. The coding and decoding of HOL functions is performed recursively, using ideas from ‘polymorphism’ [27], by composing the encodings and decodings of sub-functions, starting from a library of encodings and decodings of primitive functions. (e.g. booleans, arithmetic, list-processing, et).

There is an initial set of functions for encoding/decoding primitive HOL types (e.g. booleans, arithmetic, list-processing etc.) to S-expression. HOL allows new types to be defined recursively, and we have tools that automatically generate encodings to S-expressions from type definitions. For example, consider a HOL type definition:

\[
\text{HOL datatype}
\]

\[
\text{rose_tree = Branch of (} \alpha \times \text{rose_tree)} \text{ list} \]

This defines a new polymorphic unary type operator \text{rose_tree}, where, for arbitrary type \text{\alpha}, the values of type \text{(} \alpha \text{)rose_tree are the empty tree} \text{Branch[]} \text{and non-empty trees} \text{Branch[} (a_1, t_1); \ldots; (a_n, t_n) \text{], where} \text{a_i has type} \text{\alpha and} \text{t_i has type} \text{(} \alpha \text{)rose_tree (} 1 \leq i \leq n \text{).}

From this HOL type definition, our encoder generates an encoder function for \text{rose_tree}, parameterised on an encoder for the type parameter \text{\alpha}, i.e. a HOL constant:

\[
\text{encode_rose_tree:} (\alpha \rightarrow \text{sexp}) \rightarrow (\alpha \rightarrow \text{sexp)}
\]

If \text{List:} \text{(sexp)} \rightarrow \text{sexp} encodes HOL lists of S-expressions as ACL2 lists and \text{nat:} \text{num} \rightarrow \text{sexp} is a predefined encoder of HOL natural numbers as S-expressions, then instantiating \text{\alpha} to \text{num}:

\[
\text{encode_rose_tree nat}
\]

\[
\begin{align*}
& (\text{Branch} \\
& \quad \{(0, \text{Branch[]}); (1, \text{Branch[} (2, \text{Branch[}])])\}) \\
& = \text{List[Branch[0]; Branch[1; Branch[2]]]}
\end{align*}
\]

Each HOL type whose values can be encoded as S-expressions also has an automatically generated recogniser, which is a HOL function of type \text{sexp<--)sexp} (the S-expression returned will be \text{t} or \text{nil}). For example, \text{cons, natp} are recognisers for encoded dotted-pairs and encoded HOL natural numbers, respectively. If a type is parameterised, then its recogniser will be a function that takes recognisers for the type parameters as arguments. For example, the recogniser for S-expressions
encoding HOL lists of type \((\alpha)\text{list} \Rightarrow \text{listp of type } \langle \alpha \rangle)\). The recogniser for Cartesian product types \(\alpha \times \beta \Rightarrow \text{a (curried) function that takes recognisers for } \alpha \text{ and } \beta \text{ as arguments. The definitions of } \text{listp} \text{ and } \text{pairp} \text{ are given below, using an auxiliary function } \text{andl} \text{ that is similar to the ACL2 macro and (its definition uses HOL’s inx list-cons operator} \Rightarrow \rangle.

\[
\begin{align*}
(\text{andl} \text{[} & \text{nil} \text{]} = \text{t}) \land (\text{andl} \text{[} s \text{]} = \text{s}) \land \\
(\text{andl} \text{[} x::y::l \text{]} = \text{ite} x \text{ (andl} \text{[} y::l \text{]} \text{) nil})
\end{align*}
\]

\[
\begin{align*}
\text{listp} p x =
\begin{cases}
\text{t} & \text{if } (\text{equal} x \text{ nil})\\
\text{andl} \text{[} \text{cons} x \text{; p(car} x \text{); listp} p \text{ (cdr} x \text{)]}
\end{cases}
\end{align*}
\]

pairp f g x =
\[
\begin{align*}
\text{andl} \text{[} \text{cons} x \text{; f(car} x \text{); f(cdr} x \text{)]}
\end{align*}
\]

Whenever an encoder for a user-defined HOL type is generated a recogniser is also created.

\[
\begin{align*}
\text{rose_tree} f s =
\text{listp} \text{ (pairp} f \text{ (rose_tree} f \text{))} s
\end{align*}
\]

Our encoding tools handle polymorphic type operators, but only ground instances of polymorphic functions can be represented as functions on sexp and exported to ACL2. A ‘flattening’ process generates first-order functions in HOL suitable for exporting to ACL2.

\[
\begin{align*}
\text{nat_rose_tree} s =
\text{ite} \text{ (equal} s \text{ nil) t} \\
\text{ (andl} \text{[} \text{cons} s \text{; cons} (\text{car} s) \text{); natp} \text{ (car} (\text{car} s) \text{));} \\
\text{ nat_rose_tree} \text{ (cdr} (\text{car} s) \text{));} \\
\text{ nat_rose_tree} \text{ (cdr} s) \text{)}
\end{align*}
\]

Functions defined in HOL are automatically translated to first-order functions on S-expressions. For example, the counting function \(\text{count} \) defined by:

\[
\begin{align*}
(\text{count} \text{ (Branch} [\text{nil}] \text{)} = \text{0}) \land \\
(\text{count} \text{ (Branch} [(x,hd)::tl] \text{)} = 1 \text{ + count} \text{ hd} \text{ + count} \text{ (Branch} tl \text{)})
\end{align*}
\]

is automatically translated to the S-expression function:

\[
\begin{align*}
\text{acl2} \text{ _count} tl =
\text{ite} \text{ (nat_rose_tree} tl \text{)} \\
\text{ (ite} \text{ (cons} tl \text{) (add} \\
\text{ (add} \\
\text{ (nat} 1 \text{) (acl2} \text{ _count} \text{ (cdr} (\text{car} tl) \text{))})} \\
\text{ (acl2} \text{ _count} \text{ (cdr} tl \text{))} \\
\text{ (int} 0 \text{)}) \\
\text{ (int} 0 \text{)}
\end{align*}
\]

and a correctness theorem automatically proved:

\[
\begin{align*}
\forall tl. \text{ count} tl = \text{sexp_to_nat} \text{ (acl2} \text{ _count} \text{ (encode_rose_tree} \text{ (nat} tl \text{))})
\end{align*}
\]

If a HOL function operates only on datatypes that are in the initial library of encodings to S-expressions, then no type encoding functions need to be generated. For example, the infixed exponentiation operator \(**\) is defined recursively in HOL by a conjunction of equations, one for the basis and one for the inductive step. Built-in natural number multiplication \((\ast)\) and successor \((\text{SUC})\) functions are used.

\[
\begin{align*}
(\forall m. m ** 0 = 1) \\
(\forall n. m ** \text{SUC} n = m * (m ** n))
\end{align*}
\]

The encoder generates a definition of a constant \(\text{acl2} _\text{exp}\) that represents \(**\) as a first-order operation on S-expressions in HOL.

\[
\begin{align*}
\text{acl2} _\text{exp} m n =
\text{ite} \text{ (andl} \text{[} \text{natp} n \text{; natp} m \text{]) (ite} \text{ (equal} n \text{ (int} 0 \text{)) (int} 1 \text{) (mult} m \\
\text{ (acl2} \text{ _exp} m \\
\text{ (nfix} \text{ (add} n \text{ (unary_min} \text{e (nat} 1 \text{))))))) (int} 0 \text{)}
\end{align*}
\]

together with the correctness-validating theorem:

\[
\begin{align*}
\forall mn. m ** n = \text{sexp_to_nat} \text{ (acl2} _\text{exp} \text{ (nat} m \text{) (nat} n \text{))}
\end{align*}
\]

Note that the translator generated a definition of the function \(\text{acl2} _\text{exp}\) that returns the S-expression corresponding to zero \((\text{int} 0)\) when either of its arguments are not numbers. Similarly, the predefined SEXP encodings of addition and multiplication return \text{int} 0 on non-number arguments. This ensures that axioms like \text{ASSOCIATIVITY-OF-**} are true in the HOL model.

6. Moving between ACL2 and HOL

The link between the HOL4 and ACL2 systems is implemented by printing and reading files of S-expressions. To send an S-expression from HOL4 to ACL2 an ML function recursively prints the S-expression to a file suitable for input by ACL2. The reverse direction is similar: ACL2 recursively writes an S-expression to a file that can be input to ML. Before S-expressions are moved between systems they are elaborated into a simple form. For example, HOL terms are simplified by rewriting away auxiliary function like \text{andl} and \text{let}-terms are eliminated by \(\beta\)-reductions. ACL2 definitions and formulae have all macros expanded and \text{let}-terms replaced by equivalent ACL2 lambda. This elaboration is done securely (i.e. the elaborated forms are provably equivalent to the unlebelerated sources).

There are a few low-level details that need care, including the treatment of character, string and number literals. Modelling of ACL2 packages is quite complicated, reflecting the inherent complexity of ACL2 namespace management: see the companion paper [9].
The conversion of HOL definitions to S-expressions is straightforward. An equation \( f \ x_1 \cdots x_n = e \) is converted to \( \text{(defun } f \ ( x_1 \cdots \ x_n ) \ \hat{e} \text{)} \), where \( \hat{e} \) is the translation of \( e \). A HOL theorem \( \vdash ( \equiv \ e ) \) corresponds to \( \text{(defthm } name \ \hat{e} \text{)} \), where \( name \) is generated from the name of the theorem in HOL.

The transfer of S-expressions between the HOL4 and ACL2 systems is not validated by any proof. It is thus vital that the translation code be trustworthy. We have attempted to achieve this by keeping it straightforward. To test our code, we have imported all the ACL2 axioms, definitions and theorems from axioms.lisp and also a complete ACL2 model of Y86 (a simple X86-like machine [3]) into HOL. converted them to HOL theories and definitions, then exported them back to ACL2, and successfully checked that the results are correct. This substantially adds to our confidence that our code for printing and reading S-expressions is 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 SEXP theory (doing this is work in progress).

Definitions and theorems imported from ACL2 can either be trusted, or proof obligations can be established to validate them in HOL. Theorems created in the SEXP theory by trusting ACL2 are ‘tagged’ with their source, so one can always distinguish theorems proved entirely inside HOL from those proved using ACL2 as an oracle. ACL2 will only admit a definition if it is proved that it is consistent (indeed, conservative) to add it [15]. HOL has a similar discipline. 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).

7. Performance

In this section we briefly describe a simple example that shows the kind of performance that can be attained by using ACL2 to execute HOL specifications.

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 state of a memory is represented in HOL by a list \([(a_1,v_1);\ldots;(a_n,v_n)]\) where \(a_1,\ldots,a_n\) are the addresses holding non-zero values, and the value stored at \(a_i\) is \(v_i\) \((1 \leq i \leq n)\) – thus an empty list \([\ ]\) represents a memory in which all addresses hold 0.

The function \(\text{read}\_\text{step}\) takes an address and a memory state and returns the value stored at the address. The function \(\text{write}\_\text{step}\) takes an address, a value and a memory state and returns an updated memory state with the value written to the address.

A tail-recursive function \(\text{run}\) 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. A function \(\text{make}\_\text{instrs}\) generates text programs consisting of sequences of reads and writes starting from a given initial state.

As a benchmark we created a program with a million instructions. The results shown below should be taken as illustrative only, as they involve three different systems (Moscow ML, MLton [33], and ACL2 built on Gnu Common Lisp), and it is not clear if the same things are being timed: the ML compilers report “gc”, “sys” and “usr” times (figures given below are \(gc+sys+usr\) times), whilst ACL2/LISP reports “real”, “run-gbc” and “gbc” times (figures given below are \(\text{real} \) times). Timings were done on the same machine.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HOL</th>
<th>Moscow ML</th>
<th>MLton</th>
<th>ACL2</th>
<th>Common Lisp</th>
<th>ACL2 + stdlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\infty)?</td>
<td>5.8</td>
<td>1.26</td>
<td>8.16</td>
<td>5.6</td>
<td>0.06</td>
<td></td>
</tr>
</tbody>
</table>

Evaluation by proof in HOL (EVAL) runs out of memory after 10 hours. 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 and about the same as running them in Common Lisp\(^1\). The fastest execution is with a manually verified translation to single-threaded code [1]. 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.

8. Future work and conclusions

We plan to prove all the axioms of the ACL2 logic in HOL. This will verify that we have a sound model of ACL2.

So far we have concentrated on using ACL2 to execute HOL models. The biggest example we have done is a hand translation of the floating point evaluation function that forms part of an ARM co-processor [25]. This executed about 300 times faster in interpreted ACL2 than with HOL’s native execution facility (EVAL). Automatic encoding of this is in progress. We plan to use ACL2 to run an existing HOL model of an ARM processor [7]. This will be linked to the floating point co-processor. Using ACL2, it is hoped that we can validate the HOL ARM model against ARM’s internal simulator and against actual ARM hardware.

\(^1\)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.
We want to investigate translating HOL goals, and 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 as an ACL2 book that may be processed by ACL2. An ACL2 user then creates an ACL2 book that includes the contents of 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 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, like the functions `List` and `andl` mentioned in Section 5, it may be possible to provide better support for importing macros, but for current work, expanding them seems adequate. Macros require more thought and experimental studies to identify technical issues.

No simplifying assumptions about the ACL2 logic have been made. 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.

9. Acknowledgements

We have had discussions with many people about this work, including Bob Boyer, Joe Hurd, John Matthews, Pete Manolios, J Moore and Mark Shields. Konrad Slind made major contributions to our ideas and helped us implement tools for defining functions in the SEXP theory.

Warren Hunt and Matt Kaufmann are supported by DARPA and the National Science Foundation under Grant No. CNS-0429591. James Reynolds is supported by an EPSRC Studentship.

References


[22] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Furbach and Shankar [8].


8