An Embedding of the ACL2 Logic in HOL

Michael J.C. Gordon, Cambridge University, mjcg@cl.cam.ac.uk
Warren A. Hunt, Jr., University of Texas at Austin, hunt@cs.utexas.edu
Matt Kaufmann, University of Texas at Austin, kaufmann@cs.utexas.edu
James Reynolds, Cambridge University, jr291@cam.ac.uk

ABSTRACT

We describe an embedding of the ACL2 logic into higher-order logic. An implementation of this embedding allows ACL2 to be used as an oracle for higher-order logic provers.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computational logic, mechanical theorem proving*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*

General Terms

Languages, Security, Theory, Verification

Keywords

Verification, formal methods, logic, ACL2, HOL, HOL4, first-order logic, higher-order logic, sound translation, proof oracle

1. INTRODUCTION

We describe an embedding of the ACL2 logic [6, 5] into higher-order logic (HOL). The basis for our translation is a HOL theory, `SEXP`, which consists of an S-expression data type, `sexp`, together with translations of ACL2 primitives that operate on `sexp`. Specifically, `SEXP` is built in the following three steps:

1. hand-define the `sexp` data type;
2. hand-define translations of the built-in undefined functions; (`car`, binary-*, and so on);
3. automatically translate built-in defined functions from ACL2 source file `axioms.lisp`.

We also discuss translation of user-supplied ACL2 definitions and theorems into HOL. The key logical property is that theorems of any extension of ACL2's built-in *ground-zero* theory are translated to theorems of a corresponding extension of HOL's `SEXP` theory. The key requirement guaranteeing this property is that ACL2 axioms are translated to formulas of `SEXP` that are provable in HOL. Then ACL2 theorems will translate to HOL theorems because ACL2's first-order rules of inference correspond to valid HOL rules and HOL provides induction support that we believe to be at least as strong as ACL2's ϵ_0-induction (though we have not yet proved this).

We want to be able to use ACL2 to assist in HOL proof developments, as described in the companion paper [3]. The main idea is to bridge the gap between HOL and ACL2 by way of `SEXP`, using a two-step process. The first step is to take a "pure" HOL development (not in `SEXP`) and create a parallel `SEXP` development. For example, an append function on HOL lists would have a corresponding append function on `SEXP` "lists", i.e., null-terminated linear trees built from the `sexp cons` function, representing ACL2's `true-listp` objects. The aforementioned companion paper describes progress in this direction, using the HOL4 [8] implementation of higher-order logic. We are optimistic that conversion of HOL functions to `SEXP` functions will generally require only minimal user intervention. The second step is to translate the resulting `SEXP` development into ACL2, fully automatically. That step is conceptually straightforward once we understand the connection between `SEXP` and ACL2, which is the topic of the present paper.

Section 2 defines the S-expression data type, `sexp`, in HOL. Then in Section 3 we give some highlights of the definitions of ACL2 primitives, with full details deferred to an appendix. Section 4 discusses translation of ACL2 events into `SEXP`. After discussing some issues in Section 5, we make some concluding remarks in Section 6.

Interaction described in this paper is with the HOL4 system. But our results should apply to any implementation of classical higher-order logic.

2. S-EXPRESSIONS IN HOL

Communication with the HOL4 system is through Standard ML (SML), which provides a metalanguage for programming infrastructure, issuing commands to make definitions, and directing the proof process. Terms, types and theorems of higher order logic are distinct types `term`, `hol_type` and `thm` of SML. A typechecker ensures values of type `thm` can only be created by applying inference rules to instances of axioms or definitions. This is the main idea of 'LCF-style' theorem provers [2].

ACL2 '06 Seattle, Washington USA

The following brief introduction to HOL syntax should be adequate in support of reading this paper. All functions are unary ("curried"), so for example `if x y z` is equivalent to `(((if x) y) z)`: `(if x)` is a function taking two arguments that returns the first (`y`) if `x` is true and returns the second (`z`) if `x` is false. The symbol `!` is read "for all"; for example, `!x y. P x y` is read as "forall all `x` and `y`, `P x y`".

Package names and and symbol names will both be represented in HOL by strings (a predefined type in the logic). The following two ML commands define `packagename` and `name` to be abbreviations for the `string` type.

```
type_abbrev("packagename", ``:string``);
type_abbrev("name",        ``:string``);
```

We may now adapt code from Mark Staples [9] to define S-expressions in HOL. Note that although conceptually, the symbol and pair constructors each take two arguments, it is convenient technically to make each take one argument, yielding a function that expects the other argument — so called *currying* of higher-order logic.

```
Hol_datatype
 `sexp = ACL2_SYMBOL    of packagename => name
       | ACL2_STRING    of string
       | ACL2_CHARACTER of char
       | ACL2_NUMBER    of complex_rational
       | ACL2_PAIR      of sexp => sexp`;
```

Evaluating this SML expression defines a new HOL type, `sexp`, representing S-expressions. It is the disjoint union of symbols (as pairs of strings), strings, characters, complex rationals, and pairs of S-expressions (conses). The `complex_rational` type is defined in terms of pairs of rational numbers [1], and hence corresponds to the complex rational numbers as included in ACL2. Fortunately, HOL characters and strings correspond to those of ACL2. So the only tricky part here is symbols.

Notice that the `sexp` constructor `ACL2_SYMBOL` defined above creates a HOL term of type `sexp` whenever it is given a package name and a symbol name. However, some such objects do not correspond to ACL2 symbols. For example, the value of `(ACL2_SYMBOL "ACL2" "NIL")` does not correspond to an ACL2 symbol, because the package name of ACL2's `NIL` symbol is `"COMMON-LISP"`, not `"ACL2"`:

```
ACL2 !>(symbol-package-name 'nil)
"COMMON-LISP"
ACL2 !>
```

We choose to treat such "bad symbols" as ACL2 *bad atoms*, in the sense that the translation of `symbolp` to HOL will fail on such atoms. We elaborate in the next section. Fortunately, the ACL2 logic makes no requirement that every object be a symbol, a string, a character, a number, or a cons pair.

It is convenient to introduce short names for the `sexp` constructors. For example, the following allows us to write `num` in place of `ACL2_NUMBER`. These overloading commands allow `cons` and the others to behave like constructors, so they can be used in patterns in definitions.

```
declare_names ("ACL2_PAIR",      "cons");
declare_names ("ACL2_SYMBOL",    "sym");
declare_names ("ACL2_NUMBER",    "num");
declare_names ("ACL2_STRING",    "str");
declare_names ("ACL2_CHARACTER", "chr");
```

The metalanguage function `declare_names` is part of the infrastructure that we have programmed in SML to support our HOL-ACL2 link.

3. DEFINING ACL2 PRIMITIVES IN HOL

In this section we describe definitions of ACL2 primitive functions in `SEXP`. Details are provided in the Appendix.

But let us start by defining ACL2 constants `nil` and `t` in the HOL theory `SEXP`. First consider `nil`. This symbol is in the `"COMMON-LISP"` package, so we define a constant named `COMMON-LISP::NIL` in HOL. However, this name is cumbersome; furthermore, "`::`", "`-`", and some other characters are not handled by the HOL4 parser. Therefore, we also provide a HOL-friendly name that is overloaded onto the ACL2 name. Thus, the following definition overloads name `"COMMON-LISP::NIL"` with the HOL-friendly name `"nil"`, defined to be a call of the `sym` constructor on package name `"COMMON-LISP"` and name `"NIL"`.

```
acl2Define "COMMON-LISP::NIL"
 `nil = sym "COMMON-LISP" "NIL"`;
```

where the metalanguage function `acl2Define` is another part of the infrastructure that we have programmed to support our HOL-ACL2 link. It invokes HOL4's built-in definitional mechanism to define a new constant named `COMMON-LISP::NIL` in the `SEXP` theory satisfying the equation:

```
COMMON-LISP::NIL = ACL2_SYMBOL "COMMON-LISP" "NIL"
```

and then uses `declare_names` to create `nil` as the HOL-friendly name for this constant.

The definition of `t` is similar.

```
acl2Define "COMMON-LISP::T"
 `t = sym "COMMON-LISP" "T"`;
```

Let us turn now to the definition of ACL2 primitive functions. The ACL2 source code defines a constant `*primitive--formals-and-guards*`, whose value is an association list whose keys are the built-in ACL2 functions that do not have explicit definitions in the logic:

```
(defconst *primitive-formals-and-guards*
  '((acl2-numberp (x) 't)
    (bad-atom<= (x y) (if (bad-atom x)
                          (bad-atom y)
                        'nil))
    ...))
```

For example, `acl2-numberp` has formal parameter list `(x)` and a guard of `t`.[1] We will be ignoring the guards, which are logically irrelevant.

We need to provide a corresponding definition for each of these primitives in `SEXP`. Let us start with the definition of `acl2-numberp`. This symbol is in the `"ACL2"` package, so we define the function named `ACL2::ACL2-NUMBERP` in HOL.

```
acl2Define "ACL2::ACL2-NUMBERP"
 `(acl2_numberp(num x) = t) /\
  (acl2_numberp _      = nil)`;
```

[1]The guard is given in internal (translated) form: in this case, `'t` rather than `t`.

This definition says: "define a new function, ACL2::ACL2-NUMBERP (with alternate name acl2_numberp), returning t on any object constructed by num, and returning nil on any other object."

A full list of such definitions may be found in the Appendix. Here, we explain few of the more interesting ones.

The following definition of ACL2's addition function takes into account the behavior of this function on non-numbers.

```
acl2Define "ACL2::BINARY-+"
 `(add (num x) (num y) = num(x+y)) /\
  (add (num x) _       = num x)    /\
  (add _       (num y) = num y)    /\
  (add _       _       = int 0)`;
```

Consider the following axiom, copied from ACL2 source file axioms.lisp.

```
(defaxiom completion-of-+
  (equal (+ x y)
         (if (acl2-numberp x)
             (if (acl2-numberp y)
                 (+ x y)
               x)
           (if (acl2-numberp y)
               y
             0)))
  :rule-classes nil)
```

The care taken in the definition of add above allows us to use HOL4 to prove a translation of this axiom to SEXP.

```
 |- !x y.
     equal
      (add x y)
      (ite
        (acl2_numberp x)
        (ite (acl2_numberp y) (add x y) x)
        (ite (acl2_numberp y) y (int 0)))
     = t
```

Our current automated translation actually produces the following instead, also easily proved (see the Appendix for the definition of cpx).

```
  |- ~(equal
        (add ACL2::X ACL2::Y)
        (ite (acl2_numberp ACL2::X)
             (ite (acl2_numberp ACL2::Y)
                  (add ACL2::X ACL2::Y)
                  ACL2::X)
             (ite (acl2_numberp ACL2::Y)
                  ACL2::Y
                  (cpx 0 1 0 1)))
       =
       nil)
```

The following definition takes advantage of the fact that we have already defined nil.

```
acl2Define "COMMON-LISP::IF"
 `ite x (y:sexp) (z:sexp) =
    if x = nil then z else y`;
```

(The type decorations ":sexp" stop the HOL typechecker from making the constant COMMON-LISP::IF polymorphic; such polymorphism is harmless, but isn't useful here.)

Perhaps the trickiest part of the translation is the handling of symbols and packages. We need to make sure that SEXP faithfully represents ACL2's notions of the package name and symbol name of a symbol.

The ACL2 package system is represented in HOL with a function BASIC_INTERN, 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 an B, represented in HOL as sym "ACL2" "A" and sym "ACL2" "B".

Let us turn now to the definition of BASIC_INTERN. If pkg_name is the name of a known package and symbol_name is the name of a symbol imported into that package from some other package, named old_pkg, then:

```
  BASIC_INTERN symbol_name pkg_name =
    (sym old_pkg symbol_name)
```

E.g., BASIC_INTERN "A" "FOO" equals sym "ACL2" "A" under the definition of package "FOO" given above. Otherwise, if pkg_name is the name of a known ACL2 package, then:

```
  BASIC_INTERN symbol_name pkg_name =
    (sym pkg_name symbol_name)
```

Finally, if pkg_name is not the name of a known ACL2 package, we return an arbitrary value.

An ACL2 data structure, (known-package-alist state), is represented via a HOL constant ACL2_PACKAGE_ALIST. This constant, which helps with the definition of BASIC_INTERN, contains a list of triples

```
  (symbol-name , known-pkg-name , actual-pkg-name)
```

The idea is that when symbol-name is interned into known--pkg-name, the resulting symbol's package name is actual--pkg-name. That is, the symbol with name symbol-name and package-name actual-pkg-name is imported into package known-pkg-name.

A given ACL2 development will define ACL2_PACKAGE_ALIST for the collection of packages defined in that development. Its value for the initial (*ground-zero*) ACL2 theory contains over 2700 triples:

```
 |- ACL2_PACKAGE_ALIST =
     [("&ALLOW-OTHER-KEYS","ACL2","COMMON-LISP");
      ("*PRINT-MISER-WIDTH*","ACL2","COMMON-LISP");
      ("&AUX","ACL2","COMMON-LISP");
      .
      .
      .] : thm
```

If we define

```
    LOOKUP y [(x1,y1,z1);...;(xn,yn,zn)] x
```

to return zi if x=xi and y=yi, and to return y otherwise, then BASIC_INTERN is defined by:

```
BASIC_INTERN sym_name pkg_name =
 sym sym_name (LOOKUP pkg_name
                      ACL2_PACKAGE_ALIST
                      sym_name)
```

We then define the notion of an ACL2 symbol as follows, test whether an `sexp` constructed in HOL using the constructor `sym` represents a valid symbol in the package structure defined by `ACL2_PACKAGE_ALIST`.

```
acl2Define "COMMON-LISP::SYMBOLP"
 `(symbolp (sym p n) =
    if (BASIC_INTERN n p = sym p n)
       /\ ~(p = "")
    then t else nil)
  /\
  (symbolp _ = nil)`;
```

4. TRANSLATING ACL2 DEFINITIONS AND THEOREMS TO HOL

The preceding section explains how the ACL2 primitives are defined directly in HOL's SEXP theory. However, our embedding also demands the ability to translate ACL2 definitions, axioms, and theorems into HOL.

The translation of ACL2 definitions relies on the translation of ACL2 expressions, which has already been illustrated in the preceding section. Definitions, then, are handled in a straightforward manner. (Note that the translation, which is still evolving, converts ACL2 function names to lower case while replacing "_" with "-", at least in most common cases.) Consider the following ACL2 definition.

```
(defun d5 (x)
  (if (consp x)
      (d1 x)
    (if (symbolp x)
        (d4 'xyz)
      (d2 x))))
```

The corresponding HOL definition results in the following defining theorem (again, note that "!" is HOL's "forall" symbol).

```
|- !X.
   d5 X =
   ite (consp X)
       (d1 X)
       (ite (symbolp X)
            (d4 (sym "ACL2" "XYZ"))
            (d2 X))
```

The following example illustrates our careful handling of quoted constants. It also illustrates our translation of primitives, such as translation of `binary-+` to the HOL function `add` defined in the preceding section. The ACL2 definition

```
(defun foo (x y)
  (cons (binary-+ y x)
        '(a (b car) . c)))
```

generates a HOL definition that yields this defining theorem:

```
|- !X Y.
   foo X Y =
   cons (add Y X)
     (cons (sym "ACL2" "A")
        (cons
           (cons (sym "ACL2" "B")
              (cons (sym "COMMON-LISP" "CAR")
                 (sym "COMMON-LISP" "NIL")))
           (sym "ACL2" "C")))
```

Theorems and axioms are translated using the same mechanism as definitions (since definitions in HOL are essentially conjunctions of equations).

5. DISCUSSION AND FUTURE WORK

We have provided a connection between SEXP (a HOL theory) and ACL2, in order to factor the gap between HOL and ACL2. Part of the connection is a bridge between HOL and SEXP, which is provided through formal proof within HOL. The rest of the connection bridges a much smaller semantic gap, namely between SEXP and ACL2, which however does not have such a convenient opportunity for formalization. Thus, we make this latter connection through untrusted tools, for which testing is therefore critical. We have performed some preliminary "round-trip" tests, converting ACL2 code to SEXP and back again, that increase our confidence in the correctness of our code.

But we would also like to be confident of the key logical requirement: ACL2 axioms translate to theorems of SEXP. Thus, we have translated to HOL all `defaxiom` events in ACL2 source file `axioms.lisp` and have made significant progress towards proving those translations in HOL4. We fully expect that our definitions of the primitives and translations of functions defined in that file will make it straightforward (if tedious) to complete this exercise, which will increase confidence in the correctness of our embedding. We also intend to complete the task of showing that HOL is powerful enough to prove the necessary instances of induction.

Although we have preliminary tools for connecting HOL4 and ACL2, we are still thinking about how to create a user-friendly environment for working in both systems. For example, we imagine that `local` events will not be imported, but we have not yet implemented this idea. An essential part of our plan is that theorems and recursive definitions may be imported from ACL2 into HOL, but they will be given an ACL2 *tag*, in support of the HOL philosophy that all theorems must be given formal proofs. Thus, tagged theorems are treated as axioms from that perspective, but if we have done our job right, we can believe that these ACL2-tagged "axioms" are indeed theorems.

We intend to translate `encapsulate` events to SEXP as follows. Consider an `encapsulate` event that introduces functions `f1`, ..., `fk` that satisfy formula φ. HOL provides a mechanism for introducing corresponding functions satisfying the translation φ' of φ to SEXP, but with the obligation to prove that such functions exist satisfying φ'. This proof obligation can be marked as a theorem with an ACL2 tag, since φ has been proved in ACL2 for appropriate functions (locally defined within the `encapsulate`). Note that the same idea can be used to translate recursive (and mutually recursive) functions into SEXP, where ACL2-tagged theorems can avoid the need to prove termination in HOL.

This work supports the use of ACL2 as an oracle for HOL, because of the key property that ACL2 theorems are to be translated to theorems of SEXP. But can this work support the use of HOL as an oracle for ACL2? Investigation has begun on supporting a general mechanism for hooking external tools with ACL2, the main idea being that an external tool should implement a first-order theory. John Matthews [7] has observed that if the external tool supports a higher-order logic, then we may be able to restrict to the set of first-order consequences to get the requisite first-order theory.

6. CONCLUSION

We have shown how to connect HOL and ACL2 by defining a theory in HOL, SEXP, that is in some sense "isomorphic" to ACL2. More accurately, SEXP can be viewed as a model of ACL2. Yet more accurately, our embedding corresponds to the classical notion of theory embedding: for any theorem provable in ACL2, its translation to SEXP is provable in HOL.

The companion paper [3] describes the application of this connection to encode HOL developments into ACL2. The encoding is factored into a translation from appropriate HOL developments into SEXP, which requires proof, and a translation from SEXP to ACL2. The former may may employ some automation in both the translation and the proof. The latter is fully automatic, justified by the theory laid out in the present paper.

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`. We also thank the referees for useful expository suggestions.

This material is based upon work supported (for Warren A. Hunt, Jr. and Matt Kaufmann) by DARPA and the National Science Foundation under Grant No. CNS-0429591. James Reynolds is supported by an EPSRC Studentship.

Appendix: Defining ACL2 primitives in HOL

The following ML file, slightly abbreviated here, is taken from the public distribution hosted at SourceForge at `http://cvs.sourceforge.net/viewcvs.py/*checkout*/hol/hol98/examples/acl2/ml/sexpScript.sml`.

```
val equal_def =
 acl2Define "COMMON-LISP::EQUAL"
  `equal (x:sexp) (y:sexp) = if x = y then t else nil`;

val stringp_def =
 acl2Define "COMMON-LISP::STRINGP"
  `(stringp(str x) = t) /\ (stringp _ = nil)`;

val characterp_def =
 acl2Define "COMMON-LISP::CHARACTERP"
  `(characterp(chr x) = t) /\ (characterp _ = nil)`;

(*****************************************************************************)
(* Construct a fraction then a rational from numerator and denominator      *)
(*****************************************************************************)
val rat_def = Define `rat n d = abs_rat(abs_frac(n,d))`;

(*****************************************************************************)
(* Construct a complex from four integers: an/ad + (bn/bd)i.                 *)
(*****************************************************************************)
val cpx_def =
 Define `cpx an ad bn bd = num(com (rat an ad) (rat bn bd))`;

(*****************************************************************************)
(*  Construct a complex from an integer: n |--> n/1  + (0/1)i.               *)
(*****************************************************************************)
val int_def = Define `int n = cpx n 1 0 1`;

(*****************************************************************************)
(*  Construct a complex from a natural number: n |--> int n.                 *)
(*****************************************************************************)
val nat_def = Define `nat n = int(& n)`;

val acl2_numberp_def =
 acl2Define "ACL2::ACL2-NUMBERP"
  `(acl2_numberp(num x) = t) /\ (acl2_numberp _ = nil)`;

val add_def =
 acl2Define "ACL2::BINARY-+"
  `(add (num x) (num y) = num(x+y)) /\
   (add (num x) _       = num x)    /\
   (add _       (num y) = num y)    /\
   (add _       _       = int 0)`;

val mult_def =
 acl2Define "ACL2::BINARY-*"
  `(mult (num x) (num y) = num(x*y)) /\
   (mult _       _       = int 0)`;

val less_def =
 acl2Define "COMMON-LISP::<"
  `(less (num(com xr xi)) (num(com yr yi)) =
     if xr < yr
      then t
      else (if xr = yr then (if xi < yi then t else nil) else nil))
   /\
   (less _ (num(com yr yi)) =
     if rat_0 < yr
      then t
      else (if rat_0 = yr then (if rat_0 < yi then t else nil) else nil))
   /\
   (less (num(com xr xi)) _ =
     if xr < rat_0
      then t
      else (if xr = rat_0 then (if xi < rat_0 then t else nil) else nil))
   /\
   (less _ _ =  nil)`;

val unary_minus_def =
 acl2Define "ACL2::UNARY--"
  `(unary_minus(num x) = num(COMPLEX_SUB com_0 x)) /\
   (unary_minus _      = int 0)`;

val reciprocal_def =
 acl2Define "ACL2::UNARY-/"
  `(reciprocal (num x) =
     if x = com_0 then int 0 else num(COMPLEX_RECIPROCAL x))
   /\
   (reciprocal _ = int 0)`;

val consp_def =
 acl2Define "COMMON-LISP::CONSP"
  `(consp(cons x y) = t) /\ (consp _ = nil)`;

val car_def =
 acl2Define "COMMON-LISP::CAR"
  `(car(cons x _) = x) /\ (car _ = nil)`;

val cdr_def =
 acl2Define "COMMON-LISP::CDR"
  `(cdr(cons _ y) = y) /\ (cdr _ = nil)`;

val realpart_def =
 acl2Define "COMMON-LISP::REALPART"
  `(realpart(num(com a b)) = num(com a rat_0)) /\
   (realpart _             = int 0)`;

val imagpart_def =
 acl2Define "COMMON-LISP::IMAGPART"
  `(imagpart(num(com a b)) = num(com b rat_0)) /\
   (imagpart _             = int 0)`;

val rationalp_def =
 acl2Define "COMMON-LISP::RATIONALP"
  `(rationalp(num(com a b)) = if b = rat_0 then t else nil) /\
   (rationalp _             = nil)`;

val complex_rationalp_def =
 acl2Define "ACL2::COMPLEX-RATIONALP"
  `(complex_rationalp(num(com a b)) = if b = rat_0 then nil else t)
   /\
   (complex_rationalp _ = nil)`;

val complex_def =
 acl2Define "COMMON-LISP::COMPLEX"
  `(complex (num(com xr xi)) (num(com yr yi)) =
     num(com (if (xi = rat_0) then xr else rat_0)
             (if (yi = rat_0) then yr else rat_0)))
   /\
   (complex (num(com xr xi)) _ =
     num(com (if (xi = rat_0) then xr else rat_0) rat_0))
   /\
   (complex _ (num(com yr yi)) =
     num(com rat_0 (if (yi = rat_0) then yr else rat_0)))
   /\
   (complex _ _ = int 0)`;

val integerp_def =
 acl2Define "COMMON-LISP::INTEGERP"
  `(integerp(num n) = if IS_INT n then t else nil) /\
   (integerp _      = nil)`;

val numerator_def =
 acl2Define "COMMON-LISP::NUMERATOR"
  `(numerator(num(com a b)) =
     if b = rat_0 then int(reduced_nmr a) else int 0)
   /\
   (numerator _ = int 0)`;

val denominator_def =
 acl2Define "COMMON-LISP::DENOMINATOR"
  `(denominator(num(com a b)) =
     if b = rat_0 then int(reduced_dnm a) else int 1)
   /\
   (denominator _ = int 1)`;

val char_code_def =
 acl2Define "COMMON-LISP::CHAR-CODE"
  `(char_code(chr c) = int (&(ORD c))) /\
   (char_code _      = int 0)`;

val code_char_def =
 acl2Define "COMMON-LISP::CODE-CHAR"
  `(code_char(num(com a b)) =
     if IS_INT(com a b) /\ (0 <= reduced_nmr a) /\ (reduced_nmr a < 256)
      then chr(CHR (Num(reduced_nmr a)))
      else chr(CHR 0))
```

```
   /\
   (code_char _ = chr(CHR 0))`;

val ite_def =
 acl2Define "COMMON-LISP::IF"
  `ite x (y:sexp) (z:sexp) = if x = nil then z else y`;

(*****************************************************************************)
(* If f : 'a -> sexp then list_to_sexp f : num list : 'a list -> sexp.       *)
(*                                                                           *)
(* For example:                                                              *)
(*                                                                           *)
(* |- list_to_sexp num [1; 2; 3] =                                           *)
(*     cons (num 1) (cons (num 2) (cons (num 3) (sym "COMMON-LISP" "NIL")))  *)
(*****************************************************************************)
val list_to_sexp_def =
 Define
  `(list_to_sexp f [] = nil) /\
   (list_to_sexp f (x::l) = cons (f x) (list_to_sexp f l))`;

(*****************************************************************************)
(* coerce                                                                    *)
(*                                                                           *)
(* ; First, we need to translate this ACL2 definition:                       *)
(*                                                                           *)
(* (defun make-character-list (x)                                            *)
(*   (cond ((atom x) nil)                                                    *)
(*         ((characterp (car x))                                             *)
(*          (cons (car x) (make-character-list (cdr x))))                    *)
(*         (t                                                                *)
(*          (cons (code-char 0) (make-character-list (cdr x))))))            *)
(*                                                                           *)
(* ; We also require HOL functions coerce_string_to_list and                 *)
(* ; coerce_list_to_string (not written here) that coerce a string (e.g.,    *)
(* ; "abc") to an SEXP list (e.g., cons 'a' (cons 'b' (cons 'c' nil)))       *)
(* ; and vice-versa, respectively.                                           *)
(*                                                                           *)
(*****************************************************************************)

(*****************************************************************************)
(* (defun make-character-list (x)                                            *)
(*   (cond ((atom x) nil)                                                    *)
(*         ((characterp (car x))                                             *)
(*          (cons (car x) (make-character-list (cdr x))))                    *)
(*         (t                                                                *)
(*          (cons (code-char 0) (make-character-list (cdr x))))))            *)
(*****************************************************************************)
val make_character_list_def =
 Define
  `(make_character_list(cons (chr c) y) =
     (cons (chr c) (make_character_list y)))
   /\
   (make_character_list(cons x y) =
     (cons (code_char(int 0)) (make_character_list y)))
   /\
   (make_character_list _ = nil)`;

(*****************************************************************************)
(* "abc" |--> (cons (chr #"a") (cons (chr #"b") (cons (chr #"c") nil)))      *)
(*                                                                           *)
(* list_to_sexp maps a function down a HOL list and then conses up an        *)
(* s-expression from the resulting list. For example:                       *)
(*                                                                           *)
(*   list_to_sexp chr [a; b; c] =                                            *)
(*    cons (chr a) (cons (chr b) (cons (chr c) (sym "COMMON-LISP" "NIL")))   *)
(*                                                                           *)
(* EXPLODE explodes a HOL string into a HOL list of characters.              *)
(*****************************************************************************)
val coerce_string_to_list_def =
 Define
  `coerce_string_to_list s = list_to_sexp chr (EXPLODE s)`;

(*****************************************************************************)
(* (cons (chr #"a") (cons (chr #"b") (cons (chr #"c") nil))) |--> "abc"      *)
(*                                                                           *)
(* STRING : char->string->string  is HOL's string-cons function.             *)
(*****************************************************************************)
val coerce_list_to_string_def =
 Define
  `(coerce_list_to_string(cons (chr c) y) =
     STRING c (coerce_list_to_string y))
   /\
   (coerce_list_to_string _ = "")`;

val coerce_def =
 acl2Define "COMMON-LISP::COERCE"
  `(coerce (str s) y =
     if y = sym "COMMON-LISP" "LIST"
      then coerce_string_to_list s
      else str "")
   /\
   (coerce (cons a x) y =
     if y = sym "COMMON-LISP" "LIST"
      then nil
      else str(coerce_list_to_string(make_character_list(cons a x))))
   /\
   (coerce _ y = if y = sym "COMMON-LISP" "LIST" then nil else str "")`;

(*****************************************************************************)
(* The following function represents an ACL2 package system, but is not     *)
(* itself an ACL2 primitive; rather, it is used in the translation (see      *)
(* for example intern-in-package-of-symbol).                                 *)
(*                                                                           *)
(*   BASIC_INTERN : string -> string -> SEXP                                 *)
(*                                                                           *)
(* An ACL2 data structure is available to help with the definition of        *)
(* BASIC_INTERN.  For example, after evaluation of (defpkg "FOO" '(a         *)
(* b)), the form (known-package-alist state) evaluates to the following      *)
(* (which I have abbreviated, omitting irrelevant or not-useful info).       *)
(* Note that each package is associated with a list of imported              *)
(* symbols.  For example, "FOO" imports two symbols, represented in HOL      *)
(* as (sym "ACL2" "A") and (sym "ACL2" "B").                                 *)
(*                                                                           *)
(* (("FOO" (A B) ...)                                                        *)
(*  ("ACL2-USER" (& *ACL2-EXPORTS* ...))                                     *)
(*  ("ACL2-PC" NIL ...)                                                      *)
(*  ("ACL2-INPUT-CHANNEL" NIL NIL NIL)                                       *)
(*  ("ACL2-OUTPUT-CHANNEL" NIL NIL NIL)                                      *)
(*  ("ACL2" (&ALLOW-OTHER-KEYS *PRINT-MISER-WIDTH* ...) NIL NIL)             *)
(*  ("COMMON-LISP" NIL NIL NIL)                                              *)
(*  ("KEYWORD" NIL NIL NIL))                                                 *)
(*                                                                           *)
(* Let us turn now to the definition of BASIC_INTERN.                        *)
(*                                                                           *)
(* If pkg_name is the name of a known package and symbol_name is the         *)
(* name of a symbol imported into that package from some other package,      *)
(* named old_pkg, then:                                                      *)
(*                                                                           *)
(*   BASIC_INTERN symbol_name pkg_name = (sym old_pkg symbol_name)           *)
(*                                                                           *)
(* For example, given the package system shown above,                        *)
(* BASIC_INTERN "A" "FOO" = (sym "ACL2" "A").                                *)
(*                                                                           *)
(* Otherwise, if pkg_name is the name of a known package (from the ACL2      *)
(* data structure as shown above), then:                                     *)
(*                                                                           *)
(*   BASIC_INTERN symbol_name pkg_name = (sym pkg_name symbol_name)          *)
(*                                                                           *)
(* Finally, if pkg_name is not the name of a known package, we return        *)
(* an arbitrary value.                                                       *)
(*****************************************************************************)

(*****************************************************************************)
(* ACL2_PACKAGE_ALIST contains a list of triples                             *)
(*                                                                           *)
(*   (symbol-name , known-pkg-name , actual-pkg-name)                        *)
(*                                                                           *)
(* The idea is that when symbol-name is interned into known-pkg-name, the    *)
(* resulting symbol's package name is actual-pkg-name.  That is, the         *)
(* symbol with name symbol-name and package-name actual-pkg-name is          *)
(* imported into package known-pkg-name.                                     *)
(*****************************************************************************)

(*****************************************************************************)
(*     - LOOKUP y [(x1,y1,z1);...;(xn,yn,zn)] x  =  zi if x=xi and y=yi      *)
(*                                                   (scan from left)        *)
(*     - LOOKUP y [(x1,y1,z1);...;(xn,yn,zn)] x  =  y  if (x=/=xi or y=/=yi) *)
(*                                                   for any i               *)
(*****************************************************************************)
val LOOKUP_def =
 Define
  `(LOOKUP y [] _ = y)
   /\
   (LOOKUP y ((x1,y1,z1)::a) x =
     if (x=x1) /\ (y=y1) then z1 else LOOKUP y a x)`;

val BASIC_INTERN_def =
 Define
  `BASIC_INTERN sym_name pkg_name =
    sym (LOOKUP pkg_name ACL2_PACKAGE_ALIST sym_name) sym_name`;

val symbolp_def =
 acl2Define "COMMON-LISP::SYMBOLP"
  `(symbolp (sym p n) =
     if (BASIC_INTERN n p = sym p n) /\ ~(p = "") then t else nil)
   /\
   (symbolp _ = nil)`;

(*****************************************************************************)
(* bad-atom<=                                                                *)
(*                                                                           *)
(* ; For us, bad atoms are objects that look like symbols but whose          *)
(* ; combination of package name and symbol name are impossible for the      *)
(* ; given package system.                                                   *)
(*                                                                           *)
(*****************************************************************************)

(*****************************************************************************)
(* Pick a well-founded relation on s-expressions                             *)
(*****************************************************************************)
val SEXP_WF_LESS_def =
 Define `SEXP_WF_LESS = @R:sexp->sexp->bool. WF R`;

(*****************************************************************************)
(* ACL2_BAD_ATOM_LESS x y iff x is less then y in the well-founded relation  *)
(*****************************************************************************)
val bad_atom_less_def =
 acl2Define "ACL2::BAD-ATOM<="
  `bad_atom_less x y = if SEXP_WF_LESS x y then t else nil`;

val symbol_name_def =
 acl2Define "COMMON-LISP::SYMBOL-NAME"
  `(symbol_name (sym p n) = ite (symbolp (sym p n)) (str n) (str ""))
   /\
   (symbol_name _ = (str ""))`;

val symbol_package_name_def =
 acl2Define "ACL2::SYMBOL-PACKAGE-NAME"
  `(symbol_package_name (sym p n) =
     ite (symbolp (sym p n)) (str p) (str ""))
   /\
   (symbol_package_name _ = (str ""))`;

(*****************************************************************************)
(* pkg-witness                                                               *)
(*                                                                           *)
(* Note that ACL2 refuses to parse (pkg-witness pkg) unless pkg is an        *)
(* explicit string naming a package already known to ACL2.                   *)
(*****************************************************************************)
val pkg_witness_def =
```

```
 acl2Define "ACL2::PKG-WITNESS"
  `pkg_witness (str x) =
   let s = BASIC_INTERN "PKG-WITNESS" x in ite (symbolp s) s nil`;

val intern_in_package_of_symbol_def =
 acl2Define "ACL2::INTERN-IN-PACKAGE-OF-SYMBOL"
  `(intern_in_package_of_symbol (str x) (sym p n) =
    ite (symbolp (sym p n)) (BASIC_INTERN x p) nil)
   /\
   (intern_in_package_of_symbol _ _ = nil)`;
```

7. REFERENCES

[1] Jens Brandt. HOL module `rational`. Website of Reactive Systems Group of the Department of Computer Science at the University of Kaiserslautern. `http://rsg.informatik.uni-kl.de/tools/hol-modules/rational/`.

[2] M. J. C. Gordon and R. Milner and C. P. Wadsworth. Edinburgh LCF: A Mechanised Logic of Computation. Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[3] Michael J. C. Gordon, Warren A. Hunt, Jr., Matt Kaufmann, and James Reynolds. An integration of HOL and ACL2. In Aarti Gupta and Panagiotis Manolios, *Proceedings of Formal Methods in Computer Aided Design (FMCAD), 2006*, to appear.

[4] Matt Kaufmann and J Strother Moore. The ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2/`.

[5] Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. See URL `http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html\#Foundations`.

[6] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. Also available via URL `http://www.cs.utexas.edu/users/moore/publications/acl2-books/OrderingInformation.html`.

[7] John Matthews. Personal communication (email), Feb. 4, 2006.

[8] Michael Norrish and Konrad Slind (project administrators). The HOL4 System. SourceForge website. `http://hol.sourceforge.net/`.

[9] Mark Staples. Linking ACL2 and HOL. Technical Report UCAM-CL-TR-476, University of Cambridge Computer Laboratory, November 1999. `http://www.cl.cam.ac.uk/users/mjcg/hol2acl2/papers/staples99linking.ps`.