Connecting HOL to ACL2 (Version 2.4)

Michael J.C. Gordon[1], Warren A. Hunt, Jr.[2] and Matt Kaufmann[2]

[1] Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
E-mail: Mike.Gordon@cl.cam.ac.uk

[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 semantically coherent link between HOL, the higher-order logic supported by the HOL4 proof assistant, and the first order logic supported by ACL2. The key idea is to define a theory of s-expressions in HOL which contains definitions of the ACL2 primitive constants and functions. This theory is an intermediate layer that serves as a stepping stone between the logics of the HOL4 and ACL2 systems. It provides the target for translating explicit values, definitions and theorems between the systems.

1 Introduction

Higher order logic (henceforth HOL) provides a convenient framework for specification and verification of digital systems. HOL proof assistants (e.g. HOL98, HOL4, HOL Light, ProofPower and Isabelle/HOL) support HOL, but there are examples in which tools supporting other logics have substantially better capabilities. We consider one such example here: the ACL2 theorem prover. ACL2 can provide superior mechanisation of some kinds of automatic proof (e.g. simple inductions), and the ground execution of specifications (which is highly optimised via single threaded objects).

Our goal is to link ACL2 and HOL4 in a way that is both logically sound and enables the strengths of each system to be smoothly deployed within a single formal development. We have several general application scenarios in mind.

One such scenario is converting HOL models into ACL2 and then executing them. A specific instance is to translate the HOL-based ARM specification due to Anthony Fox into an ACL2 specification, and then use ACL2 execution to achieve high performance animation of Fox's specification. This should enable us to validate the ARM specification using ACL2's execution environment through co-simulation with ARM's internal simulators and actual ARM implementations.

Another application scenario is importing ACL2 definitions and theorems into HOL and then using a higher-order logic prover such as HOL4 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 programs to ACL2. 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 previous scenario used HOL4 to prove higher-order properties of ACL2 specifications. Conversely, one might want to use ACL2 to prove properties about HOL specifications. The first scenario was an instance of this, where proof was just execution, but we think that certain HOL goals might be easily solved by ACL2 general automation, so that an 'ACL2 tactic' in HOL4 could be a powerful tool. Such a tactic could take a HOL goal G and convert it into a corresponding ACL2 formula G_{ACL2}. Any definitions of constants used in G would need to be converted into ACL2 definitions and imported into the system. Using this imported context of definitions, ACL2 is then used to prove G_{ACL2}, and if this succeeds then G_{ACL2} could be asserted as a theorem of HOL (tagged with the information that it had been proved in ACL2).

In what follows we use "HOL" in contexts where we don't want to distinguish between proof assistants. We also use it sometimes to refer to higher order logic and sometimes to 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 hope this slightly sloppy usage is clear in practice.

2 Related work

In 1991, Fink, Archer and Yang described a proof manager called PM that enabled HOL input to be transformed into "first-order assertions suited to the Boyer-Moore prover"[1]. In 1999 Mark Staples implemented a tool called ACL2PII for linking ACL2 and HOL98. As far as we know these are the only previous attempts to link HOL to ACL2. ACL2PII was used by Kong Woei Susanto at Glasgow for his PhD research.

Both PM and ACL2PII provided ways of translating between higher order logic and first order logic. One of the tricky things about translating from untyped Boyer-Moore logic to typed higher-order logic is figuring out which types to assign. An example Staples gives is that, depending on the context, the ACL2 s-expression `NIL` might need to be translated to `F` (falsity of HOL type bool), or `[]` (the empty list) or `NONE` (the empty option). The ACL2PII user has to set

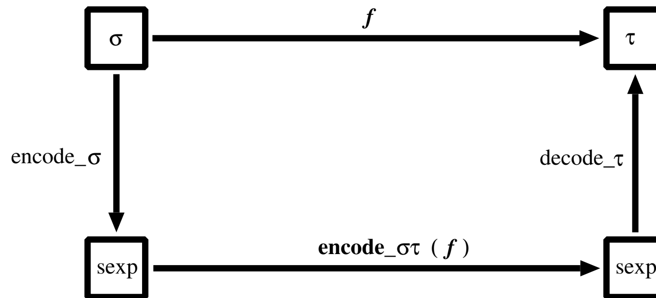
[1] George Fink, Myla Archer, Lie Yang. *PM: A Proof Manager for HOL and Other provers*, Proceedings of the 1991 International Workshop on the HOL theorem proving system and its applications, IEEE Computer Society Press, 1991.

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.

Our goal is to design and implement an approach to linking HOL4 and ACL2 that provides extremely high assurance that corresponding HOL and ACL2 formulas have corresponding semantics. This assurance will be higher than that provided by PM or ACL2PII because our approach is based upon a suitably verified translation of formulas between ACL2 and HOL. In particular, we define a formal model of ACL2's intended universe in HOL, and we mechanically prove that this model satisfies the ACL2 axioms, which can be shown to justify our translation of formulas between ACL2 and HOL. Thus scripts of PM or ACL2PII are replaced by deductions in the HOL4 system.

The idea of our approach is that a HOL theory, called `SEXP`, defines a type `sexp` representing s-expressions in higher order logic and also defines constants corresponding to the ACL2 primitive functions (`car`, `cdr`, `cons` etc.) to conform to the ACL2 logic axioms.

HOL datatypes are then encoded as s-expressions, and HOL functions are encoded as first-order functions on s-expressions. To encode HOL types σ and τ one defines HOL functions $\texttt{encode_}\sigma : \sigma \rightarrow \texttt{sexp}$ and $\texttt{decode_}\tau : \texttt{sexp} \rightarrow \tau$, respectively. Then if $f : \sigma \rightarrow \tau$ is a HOL function we attempt to encode it as a function $\texttt{encode_}\sigma\tau(f) : \texttt{sexp} \rightarrow \texttt{sexp}$, such that the diagram below commutes:



Defining type encoding and decoding functions for primitive types (booleans, numbers, strings etc.) is expected to be a one-off and relatively straightforward piece of infrastructure building. It is hoped to develop tools that can automatically create encoding for a subset of user-defined datatypes (e.g. enumerated datatypes and records). Creating an encoding of a function, f say, is in general harder, but given encodings of the functions that are used to define f, then we hope to be able to automatically generate an encoding of f. A possible approach is to use ideas from polytypism[2]. The problem is similar to the 'boolification' needed for encoding HOL types and functions as the boolean values and functions necessary for model checking.

[2] Konrad Slind, Joe Hurd. *Formally Verified Encryption of High-Level Datatypes: An Application of Polytypism.* http://www.cs.utah.edu/ slind/papers/encrypt.html

3 Foundational issues

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:

1. as an s-expression in ACL2's version of Lisp;
2. as a formula of first order logic.

Under the first view, 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, then we could try to build a formal model of ACL2's Lisp evaluation, so that the ACL2 axioms can be proved consistent with Lisp semantics. However, this would need to be validated with respect to some reference evaluator and it is not clear what this reference should be, since there really isn't an "ACL2 definition of EVAL".

We have decided to adopt the second view, namely that the axioms in the ACL2 source file `axioms.lisp` define the logic, rather than some 'golden' evaluator. If there are discrepancies between this and the actual behavior 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 the set of s-expressions in higher-order logic (by defining a type `sexp`) and then to specify HOL functions operating on this type that correspond to the ACL2 functions axiomatized in `axioms.lisp` (`cons`, `car`, `cdr`, `consp`, `if`, 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.

3.1 SEXP: s-expressions in HOL

The type `sexp` of s-expressions is defined in the HOL theory SEXP, which provides a stepping stone between higher order HOL and first order ACL2. The theory SEXP also contains definitions of constants corresponding to the ACL2 primitive constants and functions. The type `sexp` is a recursively defined datatype composed of four kinds of atoms (symbols, strings, characters and complex rational numbers) and pairs, whose components are s-expressions.

The actual definition of the type `sexp` in HOL is shown below (comments are enclosed between `(*` and `*)`).

```
(******************************************************************************)
(* Introduce mnemonic abbreviations to indicate use of type ``:string``      *)
(******************************************************************************)
type_abbrev(``packagename'', ``:string``);
type_abbrev(``name'',        ``:string``);
```

These type abbreviations provide mnemonic names for occurrences of the type `string`. The ML function `Hol_datatype` used below introduces an Nqthm-style 'shell principle' and simultaneously introduces a new type `sexp` and constructors:

```
(******************************************************************************)
(* ACL2 S-expressions defined as a HOL datatype.                              *)
(* Definition below adapted from Mark Staples' code.                          *)
(******************************************************************************)
Hol_datatype
 `sexp = ACL2_SYMBOL    of packagename => name     (* only curried for style *)
       | ACL2_STRING    of string
       | ACL2_CHARACTER of char
       | ACL2_NUMBER    of complex_rational
       | ACL2_PAIR      of sexp => sexp`           (* only curried for style *)
```

The constructors `ACL2_SYMBOL` and `ACL2_PAIR` are curried, but this is just a style choice (it means we can write `ACL2_PAIR x y` instead of `ACL2_PAIR(x,y)`). The HOL type `char` only has 256 values, so exactly matches ACL2's characters. The type `complex_rational` is defined using the rational package (`ratLib`) developed by Jens Brandt of the Department of Computer Science at the University of Kaiserslautern.

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 package structure and making the definition of `symbolp` return `nil` on HOL symbols which are not symbols according to this structure.

For readability and conciseness , short abbreviations for the contructors of the datatype `sxp` are defined.[3]

```
sym  : string -> string -> sexp  (abbreviates ACL2_SYMBOL)
str  : string -> sexp            (abbreviates ACL2_STRING)
chr  : char -> sexp              (abbreviates ACL2_CHARACTER)
num  : complex_rational -> sexp  (abbreviates ACL2_NUMBER)
cons : sexp -> sexp -> sexp      (abbreviates ACL2_PAIR)
```

Numbers in ACL2 are complex rationals, which consist of two rationals: the real and imaginary parts. Rationals 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). We have defined a constant `cpx` of type `int->int->int->int->sexp` such that a complex rational `(a/b) + (p/q)`i is represented by the HOL term `cpx a b p q`. We have also defined additional constructors `nat:num->sexp` and `int:int->sexp` to convert HOL natural numbers and integers to complex-rationals. They are defined by:

[3] The `sexp` constructor names starting with `ACL2_` date from an earlier implementation and may be eliminated in favour of the short names in future versions.

```
int n = cpx n 1 0 1
nat n = int(& n)
```

The operator `&` injects a natural number (i.e. value of type `num`) to the corresponding integer (value of type `int`). The term `& n` is equivalent to `int_of_num n`.

3.2 Defining the ACL2 primitives

We aim to define the meaning of each primitive ACL2 function in HOL so that the appropriate axioms in `axioms.lisp` are satisfied.

Each ACL2 function or constant that we define is given a name of the form `package::name`. Since `::` is not allowed in identifiers by the HOL4 lexical analyser (and there are also many other character sequences that HOL4 can't parse occurring in ACL2 names) we also provide a HOL4 friendly name which is overloaded onto the ACL2 name. For example the ACL2 name `ACL2::BAD-ATOM<=` is given the HOL4 friendly name `bad_atom_less`.

The HOL logic allows constants to have arbitrary strings as names, but the ML tools for inputting terms via the parser requires the names to be simple (a letter followed by a sequence of letters, numbers, prime characters (`'`) or underscores (`_`). In order to define `SEXP` versions of ACL2 constants and functions we have defined a tool `acl2Define` that can define a constant with an arbitrary string as its name. An example of its use is:

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

Here we define a constant called `COMMON-LISP::NIL` (supplied as an ML string as the first argument to `acl2Define`) by giving an equation (the second argument to `acl2Define`) that looks like it's defining a constant `nil`. In fact, `acl2Define` substitutes the string for the left hand side of the equation before performing a definition event in HOL4. After the definition is made, then `acl2Define` declares the left hand side of the originally supplied equation (`nil`) as an abbreviation for the constant just defined (`COMMON-LISP::NIL`).

The first order theory of ACL2 axiomatizes (in `axioms.lisp`) the Lisp atoms `nil` and `t` (defined analogously to `nil`) together with 31 ACL2 primitives:

```
acl2-numberp bad-atom<= binary-* binary-+ unary-- unary-/ < car cdr
char-code characterp code-char complex complex-rationalp coerce cons
consp denominator equal if imagpart integerp
intern-in-package-of-symbol numerator pkg-witness rationalp realpart
stringp symbol-name symbol-package-name symbolp
```

The `SEXP` definitions of these are given in Appendix 2.

In `axioms.lisp` it is specified that when a term `p` is used as a formula it means `(not (equal p nil))`. From the definition of `not` we see that the only values `not` can return are `nil` or `t`. Thus we can define an ACL2 term `p` to be true if `(not (equal p nil))` is equal to `t` in HOL. As a sanity check it is easy to prove that `(not(equal p nil) = t)` is equivalent to `¬(p = nil)`, where the infixed equality symbol (`=`) is HOL's built-in equality.

4 Passing objects between HOL and ACL2

We imagine there being three main classes of objects, all represented as s-expressions, that we wish to pass back and forth between HOL and ACL2: explicit data, definitions, and theorems.

One scenario we envision is translating HOL goals, together with the definitions they depend on, into SEXP. This may require manual proof, but we hope to develop some mechanical support (see diagram and discussion on page 3). Once the problem has been reformulated into SEXP, we export it to a file of ACL2 s-expressions so they may be processed by ACL2. An ACL2 user then creates an ACL2 *book* that includes the exported s-expressions and perhaps additional intermediate definitions and lemmas. After ACL2 has *certified* this book, the original SEXP goal can be declared to have been solved, at which time an ACL2-tagged SEXP theorem achieving the goal is created. For this to be sound, we need to be sure that the original goal could have been proved in 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 ε_0 are available. The HOL logic is sufficiently strong that these instances are just provable in SEXP.

Another scenario is to transfer an ACL2 development, say JVM, into SEXP. This raises the issue of how to manage macros. One approach would be to expand them before moving to SEXP, but this could cause a blowup and loss of mnemonic abstractions. It may be possible to provide some support in SEXP for importing macros either by defining appropriate auxilliary HOL definitions or ML functions which are then executed and anti-quoted. Alternatively, a trusted but ugly translation could be performed by first eliminating all macros on the ACL2 side. Ultimately one can imagine checking that the two translations are equivalent, so that one can use the pretty version (without eliminating macros on the ACL2 side first); perhaps they are even read into identical internal HOL terms. Macros require more thought and a case study to identify technical issues.

4.1 Data values in HOL and ACL2

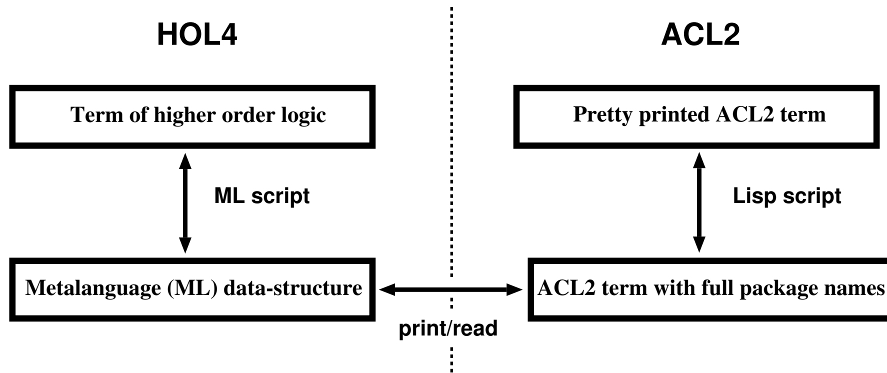
HOL and ACL2 each have their own notions of characters, strings and numbers. Fortunately the match between characters and strings in HOL and ACL2 is perfect. ACL2, numbers are specified axiomatically in `axioms.lisp`, which contain axioms like the associativity and commutativity of addition and multiplication. HOL numbers are defined using a construction based on classical methods.

The HOL theory SEXP contains definitions of the ACL2 versions of arithmetical functions using the native HOL versions. The intention is that all the axioms of the ACL2 logic in `axioms.lisp` can be proved from these definitions. This approach forces numbers to conform to the standard model (which is the only model satisfying the HOL definition of numbers, but cannot be specified in first-order ACL2). 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: we already know that there are things that can be proved in HOL but not in ACL2.

4.2 Tranferring values between HOL and ACL2

The transfer of s-expressions between the HOL4 and ACL2 systems should be as simple as possible, since this code needs to be trusted. A logic term in the `SEXP` subset of HOL is transferred to the ACL2 system by first converting it to a data-structure in ML, then printing this to a file containing ACL2 input, then reading the file into ACL2 and pretty-printing it back out. Going from ACL2 to HOL4 is essentially the reverse: an ACL2 script creates a file that when read into HOL builds an ML data-structure, which is then converted to a term of higher order logic by an ML program. The following diagram illustrates this.



The reason for having a special ML data-structure for representing ACL2 terms is to maximise simplicity. The printing scripts in ML and ACL2 are simpler if they do not have to deal with some of the intricacies of HOL logic terms. The main intricacy is in the treatment of atoms, particularly symbols and numbers.

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"`. 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 `"ACL2"`). We transfer s-expressions between HOL4 and ACL2 using fully expanded symbols that always have an explicit package name.

Although numbers in higher order logic are 'mathematical' (i.e. infinite precision), numbers in the ML programming language are finite precision and implementation dependant (typically 32-bit). We thus use strings to represent numbers in ML (via decimal notation) and hence to transfer them between the two systems. The ML definition of the data-type `mlsexp` representing ACL2 s-expressions is thus:

```
datatype mlsexp =
   mlsym  of string * string
 | mlstr  of string
 | mlchr  of char
 | mlnum  of string * string * string * string
 | mlpair of mlsexp * mlsexp
```

We have implemented an ML function `term_to_mlsexp` to convert terms of HOL logical type `sexp` to ML data values of ML type `mlsexp`. For example:

```
- val mlsexp1 = term_to_mlsexp ``(cons (nat 3) nil)``;
> val mlsexp1 =
    mlpair(mlsym("COMMON-LISP", "CONS"),
           mlpair(mlnum("3", "1", "0", "1"),
                  mlpair(mlsym("COMMON-LISP", "NIL"),
                         mlsym("COMMON-LISP", "NIL")))) : mlsexp
```

Printing such values into a form that can be input to ACL2 is straightforward: we have implemented a function `pr_mlexp` that performs such printing. Continuing the interactive session started above:

```
- pr_mlsexp mlsexp1;
(COMMON-LISP::CONS 3/1 COMMON-LISP::NIL)
```

Note that numbers are printed in ACL2 notation and symbols are printed with explicit package names.

We have implemented a more general printing function `print_mlsexp` that can print to an arbitrary output stream (e.g. to a file). The function `pr_mlsexp` is the special case with the current interactive session as the output stream. The function `pr_sexp` is the composition of `term_to_mlsexp` and `pr_mlsexp`:

```
(*****************************************************************************)
(* Print an s-expression term to the interactive session                    *)
(*****************************************************************************)
fun pr_sexp t = pr_mlsexp(term_to_mlsexp t);
```

An example is:

```
- pr_sexp ``cons (nat 1) (cons (cons (str "foo") (nat 2)) (cons (nat 3) nil))``;
(COMMON-LISP::CONS
  1/1
  (COMMON-LISP::CONS
    (COMMON-LISP::CONS "foo" 2/1)
    (COMMON-LISP::CONS 3/1 COMMON-LISP::NIL)))
```

Complex rational numerals like ``` ``cpx 3 (~1) 0 1`` ``` with negative denominators are allowed in HOL, but for simplicity we currently only support translation to ACL2 of numerals in which the denominators of both the real and imaginary parts of a number are non-negative. This simplifies conversion to ACL2, since `-5/3` is valid input to ACL2, but `5/-3` isn't. Attempting to translate such a 'bad' numeral will currently result in a runtime translation exception. This is illustrated at the end of the example below.

```
- val mlsexp2 = term_to_mlsexp ``(cons (int ~3) nil)``;

> val mlsexp2 =
    mlpair(mlsym("COMMON-LISP", "CONS"),
           mlpair(mlnum("-3", "1", "0", "1"),
                  mlpair(mlsym("COMMON-LISP", "NIL"),
                         mlsym("COMMON-LISP", "NIL")))) : mlsexp

- pr_mlsexp mlsexp2;
(COMMON-LISP::CONS -3/1 COMMON-LISP::NIL)

- val mlsexp3 = term_to_mlsexp ``(cons (cpx (~3) 1 4 1) nil)``;
> val mlsexp3 =
    mlpair(mlsym("COMMON-LISP", "CONS"),
           mlpair(mlnum("-3", "1", "4", "1"),
                  mlpair(mlsym("COMMON-LISP", "NIL"),
                         mlsym("COMMON-LISP", "NIL")))) : mlsexp

- pr_mlsexp mlsexp3;
(COMMON-LISP::CONS (COMMON-LISP::COMPLEX -3/1 4/1) COMMON-LISP::NIL)

- val mlsexp4 = term_to_mlsexp ``(cons (cpx 3 (~1) 4 1) nil)``;
cpx 3 (~1) 4 1
bad use of cpx
! Uncaught exception:
! HOL_ERR
```

An s-expression in HOL is moved to ACL2 by first converting it to an ML s-expression (using `term_to_mlsexp`), printing it to a file (using print_mlsexp) and then reading the file into ACL2.

We have also implemented a function in ACL2 that will translate an ACL2 file into an ML file that when read into HOL4 will create a list of ML s-expressions. Further tools then translate these ML s-expressions to HOL terms (e.g. to HOL s-expressions or to definitions of functions on s-expressions).

Definitions and theorems are encoded as s-expressions (see below). Thus the link between the HOL4 and ACL2 systems is based on files of s-expressions. We chose this rather primitive mechanism partly for its simplicity, but also because the generated files provide a trail for use in reviews (this may be important in applications that are required to be scrutinised by an assurance evaluator).

4.3 Definitions, axioms and theorems

We have written a function that converts a HOL definition to an s-expression in ML. For example, it converts a HOL function definition:

```
|- double x = cons x (cons x nil)
```

to the ML s-expression that prints as:

```
(COMMON-LISP::DEFUN ACL2::double (ACL2::x)
  (COMMON-LISP::CONS ACL2::x (COMMON-LISP::CONS ACL2::x COMMON-LISP::NIL)))
```

The package name `ACL2` that is added to the symbols `double` and `x` is the default name, which is stored as a string in an ML reference variable `current_package`.

We can also import functions defined in ACL2 to create definitions in HOL. This is done by printing an ACL2 `defun` to a file that can be read into ML to create a corresponding s-expression in ML. This s-expression is then converted to a higher order logic equation, which can either be submitted as a definition to HOL4's definition mechanism (TFL) or made into an ACL2-tagged axiom.

When defining a function over s-expressions in the `SEXP` layer, one must prove that it is total before it can be admitted. This can either be done in HOL4, or it could be done in ACL2. In the latter case, we envision transferring a proposed definition to ACL2, proving termination there, and then declaring a HOL constant together with an axiom asserting that it satisfies the defining equation validated in ACL2 (suitably tagged to show that the axiom is justified by ACL2).

We can also import ACL2 axioms (`defaxiom` events) and theorems (`defthm` events). Whether to trust ACL2 and just assert these as axioms in HOL is up to the user, though any theorems which have not been proved in HOL4 will be tagged to indicated their provenance.

As a test of our tools for transferring s-expressions between HOL4 and ACL2 we have imported several hundred ACL2 definitions, axioms and theorems into HOL, exported them back to ACL2, and then proved in ACL2 that the results are equivalent to the originals. This gives us reasonably high confidence that our tools are sound.

5 Future Work

Further testing and validation is needed. We would like to prove all the ACL2 axioms from the HOL definitions of the ACL2 primitives.

Theorem proving infrastructure for encoding HOL4's higher order logic into ACL2's first order logic is essential if the ideas described here are to be deployed on significant examples.

Once we are happy with our methodology we will want to try some larger examples, such as running Anthony Fox's ARM model using ACL2 by translating it into ACL2. We will also need to create suitable initial states so that a co-simulation can be performed so we can validate the ARM model against ARM's internal simulator and against actual ARM hardware. Such examples should help us improve our flow and the associated tool-chain.

6 Conclusion

Our approach depends crucially on having a trustworthy formalization of the ACL2 logic embedded in the higher order logic theory `SEXP` and accurate translations between `SEXP` and ACL2. We are assuming that the HOL4 and ACL2 systems are trustworthy. We want the use of the combination of the two systems to introduce the smallest possible additional soundness worries.

Our goal is to enable the complementary strengths of HOL and ACL2 to be smoothly and safely deployed together for large scale formal verification efforts. The ease with which these two systems can be used together will be an important factor in the eventual usefulness of such a combined system.

7 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`.

Appendix 1: Documentation

This appendix describes how to install and use the HOL-ACL2 link. Because the ACL2 logic is represented inside HOL, much development can be done entirely in the HOL4 system. For easy access we have created a directory `examples/acl2` in the `Kananaskis 4` version of HOL4 system at the SourceForge repository[4]. Note that this was done in December 2005, so you will need to get a sufficiently recent version of the `Kananaskis 4` system.

The ML code is in the directory `examples/acl2/ml` and the Lisp code is in `examples/acl2/lisp`.

Installing and compiling

To compile the ML code run `Holmake` in the directory `acl2/ml`. This will take a few minutes, and should produce copious output that looks like the following (where the "`...`" represents a lot of deleted stuff).

```
Analysing basic_defaxiomsScript.sml
Trying to create directory .HOLMK for dependency files
Analysing sexp.sml
Analysing complex_rationalScript.sml
Compiling complex_rationalScript.sml
Linking complex_rationalScript.uo to produce theory-builder executable
<<HOL message: Created theory "complex_rational">>
...
Exporting theory "basic_defaxioms" ... done.
Theory "basic_defaxioms" took 3.146s to build
Analysing basic_defaxiomsTheory.sml
Analysing basic_defaxiomsTheory.sig
Compiling basic_defaxiomsTheory.sig
Compiling basic_defaxiomsTheory.sml
```

To use ACL2 with HOL you need to install the ACL2 system[5] and then edit the 'editable variables' in `acl2/lisp/Makefile`, `acl2/lisp/a2ml.csh` and `acl2/lisp/pprint-file.csh` to have the correct paths for your installation. After making these edites, run `make` in the directory `acl2/lisp`. The result should be something like the following.

```
echo 'Using ACL2=/local/scratch/mjcg/ACL2/matt/v2-9-3/saved_acl2'
Using ACL2=/local/scratch/mjcg/ACL2/matt/v2-9-3/saved_acl2
make -s -f Makefile  pprint-file.cert  a2ml.cert  filter-forms.cert
 INHIBIT='(assign inhibit-output-lst (list (quote prove) (quote proof-tree) (quote warning) (quote observation) (quote event)))'
 ACL2='/local/scratch/mjcg/ACL2/matt/v2-9-3/saved_acl2'
Making /local/scratch/mjcg/HOL98/hol98/examples/acl2/lisp/filter-forms.cert on Fri Dec 16 09:42:48 GMT 2005
-rw-rw-r--  1 mjcg mjcg 389 Dec 16 09:42 filter-forms.cert
Making /local/scratch/mjcg/HOL98/hol98/examples/acl2/lisp/pprint-file.cert on Fri Dec 16 09:42:49 GMT 2005
-rw-rw-r--  1 mjcg mjcg 531 Dec 16 09:42 pprint-file.cert
Making /local/scratch/mjcg/HOL98/hol98/examples/acl2/lisp/a2ml.cert on Fri Dec 16 09:42:49 GMT 2005
-rw-rw-r--  1 mjcg mjcg 363 Dec 16 09:42 a2ml.cert
```

Note that a couple of extra line breaks have been added to the third line of the output shown above.

---

[4] http://sourceforge.net/projects/hol/

[5] http://www.cs.utexas.edu/users/moore/acl2/

Overview of theories and files

The theory of s-expressions is called `sexpTheory`. This theory depends on two others: `complex_rationalTheory` and `package_alistTheory`. The first of these defines the complex rational numbers using the theory `ratTheory` of rationals (due to Jens Brandt of the Department of Computer Science at the University of Kaiserslautern). The theory `package_alistTheory` defines two constants, `ACL2_PACKAGE_ALIST` and `ACL2_KNOWN_PACKAGES`, which are used to represent the package structure of ACL2.

The file `sexpTheory` defines the type `sexp` of s-expressions (see Section 3.1) and the 31 ACL2 primitives (see Section 3.2). The definitions of the primitives are listed in Appendix 2.

The file `acl2/ml/sexp.sml` contains ML utilities, including `acl2Define`, which is used to define the ACL2 primitives. The file `acl2/ml/sexp.ml` contains utilities (currently just `new_defun`) that cannot be compiled and so have to be loaded into an interactive session. ACL2 utilities are defined in the directory `acl2/lisp`. These utilities are documented below.

The file `acl2/ml/load_sexp.ml` should be input (with `use`) in an interactive session to load the basic HOL4-ACL2 infrastructure.

Representation of ACL2 s-expressions in higher order logic

The higher order logic datatype `sexp` that represents s-expressions inside the HOL4 logic is described in Section 3.1. The constructors on this datatype are:

```
sym  : string -> string -> sexp  (symbols    -- abbreviates ACL2_SYMBOL)
str  : string -> sexp            (strings    -- abbreviates ACL2_STRING)
chr  : char -> sexp              (characters -- abbreviates ACL2_CHARACTER)
num  : complex_rational -> sexp  (numbers    -- abbreviates ACL2_NUMBER)
cons : sexp -> sexp -> sexp      (pairs      -- abbreviates ACL2_PAIR)
```

The names starting with `ACL2_` are the constant names in the theory; the short names are abbreviations overloaded onto them, so are just a user-interface device.

The complex rationals are defined using the type `rat` of rational numbers defined in Jens Brandt's package, which is documented in Section 4.3.5 of the HOL System DESCRIPTION. Rationals are a quotient type[6] built over a type `frac` of fractions. The constructor for fractions is `abs_frac : int # int -> frac` and the destructor is `rep_frac : frac -> int # int`. The constructor for rationals is is `abs_rat : frac -> rat` and the destructor is `rep_rat : rat -> frac`. The type `complex_rational` is defined by:

```
Hol_datatype `complex_rational = COMPLEX_RATIONAL of rat => rat`
```

The short name `com` is overloaded onto the constructor name `COMPLEX_RATIONAL`.

[6] See http://www.trustworthytools.com for documentation of HOL4's quotient type package, which is due to Peter Homeier.

If `rat` is defined by `rat n d = abs_rat(abs_frac(n,d))`, and if `a`, `b`, `p`, `q` are integers, then the term `com (rat a b) (rat p q)` represents the complex rational number `(a/b) + (p/q)`i.

The following useful auxiliary functions for constructing s-expression numbers are defined:

```
cpx : int -> int -> int -> int -> sexp  (* cpx a b p q = num((a/b) + (p/q)i)  *)
int : int -> sexp                       (* int n       = cpx n 1 0 1          *)
nat : num -> sexp                       (* nat n       = int(int_of_num n)    *)
```

The function that converts a HOL natural number to the corresponding ACL2 number is called `nat` because the name `num` is already used for the function that converts HOL complex rationals to the corresponding ACL2 complex rationals (`num` is the short name for `ACL2_NUMBER`).

Representation of ACL2 s-expressions as data in ML

The ACL2 events that can be imported into HOL are `defun`, `mutual-recursion`, `defaxiom` and `defthm`. The ML datatype `acl2def` is used to hold such events, and is defined by

```
datatype acl2def =
   defun    of string * thm
 | defaxiom of string * term
 | defthm   of string * term
```

Mutual recursions (`mutual-recursion`) are currently represented as lists of `defun`s (this may change).

When a `defun` is imported from ACL2 it is translated into a HOL theorem tagged with `ACL2_DEFUN`. Thus we currently trust that ACL2 correctly admits functions (proves termination of recursion etc.). Axioms and theorems are currently not automatically asserted as theorems, though users can do so (with an appropriate tag) themselves.

The ML type `mlsexp` described in Section 4.2 is used both for reading files generated by ACL2 into HOL4, and also as a stepping stone when converting logic terms representing s-expressions into ACL2 terms. It is not expected that users of the HOL4-ACL2 link will need to manipulate ML data-structures representing s-expressions. However, the constructors are described here because they may be seen in files, error messages etc. The constructors are:

```
mlsym  : string * string -> mlsexp
mlstr  : string -> mlsexp
mlchr  : char -> mlsexp
mlnum  : string * string * string * string -> mlsexp
mlpair : mlsexp * mlsexp -> mlsexp
```

Complex rational numbers are represented by four ML strings: numerator and denominator of the real part and numerator and denominator of the imaginary part. The strings are signed decimal numerals, for example `"-327"`.

Note that the types `string` and `char` here are ML types. There is potential for confusion with the types of the same name in higher order logic! Terms representing s-expressions inside higher order logic have logic type `sexp`, and the constructors are constants in higher order logic. The constructors of the ML type `mlsexp` are functions written in the ML programming language.

There are curried versions of the ML s-expression constructors that are named with `mk` instead of `ml`:

```
mksym  : string -> string -> mlsexp
mkstr  : string -> mlsexp
mkchr  : char -> mlsexp
mknum  : string -> string -> string -> string -> mlsexp
mkpair : mlsexp -> mlsexp -> mlsexp
```

ML variables

Variables of reference type are global and modifiable and can be updated dynamically by assignment (`:=`). Functions accessing them will see the latest value.

`current_package : string ref`

Holds the current package name. This is used when adding package names to variables occurring in terms that are exported to ACL2. The initial contents of `current_package` is `"ACL2"`.

`acl2_simps : thm list ref`

Holds all definitional theorems created by `acl2Define` together with a few other theorems useful for simplification. The initial contents of `acl2_simps` is `[|- ("T" = "NIL") = F]`.

`acl2_name_list : (string * string) list ref`

Holds a list of pairs of strings. The second component of each pair is the name of an ACL2 function, and the first component is the 'HOL friendly' name that is overloaded onto this name by `acl2Define` (i.e. `c` in the description of `acl2Define` below). The contents is initially empty, though after `sexpTheory` is loaded it will contain all the `sexp`-datatype constructors and the names of the ACL2 primitives.

`acl2_list_ref : mlsexp list ref`

Holds a list of s-expressions representing `defun`, `mutual-recursion`, `defaxiom` and `defthm` events imported from ACL2. Initially empty.

`a2ml : string`

Holds the absolute path of the shell script `a2ml.csh`. Useful for invoking the script from ML using `Process.system a2ml "`*input-file*`" "`*output-file*`"`.

`pp : string`

Holds the absolute path of the shell script `pprint-file.csh`. Useful for invoking the script from ML using `Process.system pp "`*input-file*`" "`*output-file*`"`.

ML functions

`acl2Define : string -> term frag list -> thm`

The first argument is an arbitrary string that is the name of a constant to be defined. The second argument should be a set of defining clauses for a new constant, `c` say, of the sort one would give to `Define`. The name `c` should be a letter followed by a sequence of letters, numbers, prime characters (`'`) or underscores (_), as required by `Define`.

The effect of `acl2Define` is to replace the `c` in the supplied defining clauses by the supplied string, and then to make a definition using the resulting clauses which will result in a new constant being defined whose name is the string. The name `c` will then be overloaded on this constant (using `declare_names`), so can be used for input via the parser (see also Section 3.2 for an example). The theorem defining the string is returned, and also added to a global list `acl2_simps` that is used by `ACL2_SIMP_TAC`.

`declare_names : string * string -> unit`

`declare_names("`*acl2name*`","`*holname*`")` checks *acl2name* is the name of exactly one constant and that *holname* isn't an existing abbreviation. If these checks succeed, then HOL's overloading mechanism is used to introduce *holname* as an alternative name for *acl2name*. The name *holname* must be an identifier that can be parsed, i.e. consist of a letter followed by a sequence of letters, numbers, prime characters (`'`) or underscores (_). The pair `("`*holname*`","`*acl2name*`")` (note switched order) is added to the list stored in the variable `acl2_name_list`. Following a common ML convention, functions which just have a 'side effect' return the one-element type `unit`.

`save_acl2_state : unit -> unit`

Evaluating `save_acl2_state()` saves the values of the variables `acl2_simps`, `current_package` and `acl2_name_list` in the current theory and then exports this theory, i.e. evaluates `export_theory()`. Thus the last line of any theory script file that uses `acl2Define` should evaluate `save_acl2_state()` rather than `export_theory()`.

`mk_defs : mlsexp -> acl2def list`

Takes an ML s-expression generated by reading in a file created with `a2ml.csh` and converts it into a list of appropriate `acl2def` values. The list will in fact

have only one element unless the source is a mutual recursion, in which case a list of the mutually recursive definitions is returned.

`print_lisp_file : string -> ((string -> unit) -> 'a) -> unit`

Evaluating `print_lisp_file "`*filename*`"` *printer* creates an output stream to a file called *filename* and then applies the function *printer* to it, and finally flushes and closes the stream. The ML type variable `'a` matches any type. This polymorphism is illustrated in the section on `print_acl2def` below, where `print_lisp_file` is used twice, with different instantiations of `'a`.

`print_acl2def : (string -> unit) -> acl2def -> unit`

Prints an `acl2def` value to a supplied output stream. For example, the ML function `print : string -> unit` is an output stream that writes to the current interactive session, so evaluating:

```
print_acl2def print def
```

will print *def* to the current interactive session. Evaluating

```
print_lisp_file "filename" (fn out => print_acl2def out def)
```

will print *def* to a file named *filename*, and if *def-list* is a list of ML `acl2def` values, then

```
print_lisp_file
  "filename"
  (fn out => map (print_acl2def out) def-list)
```

will print the list of values into a single file names *filename*. Normally one should pretty-print files created this way by using `pprint-file.csh`. Note that `fn x => ...` is ML's notation for lambda-abstractions.

Shell scripts and Lisp utilities

The shell scripts and supporting files described in this section are in the directory `acl2/lisp`. The scripts invoke ACL2 to run the Lisp files.

`filter-forms.csh`

Translates an ACL2 file to use only primitive constructs (e.g. `let`s are translated to `lambda`s). The script is invoked by:

```
filter-forms.csh input-file output-file
```

ACL2 files should be processed with `filter-forms.csh` before being converted to ML using `a2ml.csh` (see below).

`a2ml.csh`

Converts an ACL2 file, starting with an in-package form and followed by `defun`, `mutual-recursion`, `defaxiom` and `defthm` events, into an ML file. The act of reading that ML file into HOL updates the contents of the references bound to the ML variables `current_package` and `acl2_list_ref`. The script is invoked by

`a2ml.csh` *input−file output−file*

where *input-file* is a file of ACL2 events, and *output-file* is the ML file for reading into HOL. An example use is to run the following in the directory `acl2/ml`.

```
../lisp/a2ml.csh ../lisp/basic_defaxioms.lisp basic_defaxioms.lisp.ml
```

The resulting file can then be read into HOL by evaluating:

```
use "basic_defaxioms.lisp.ml"
```

The script `a2ml.csh` is a shell wrapper for `a2ml.lisp`.

`pprint-file.csh`

Pretty prints a file generated from HOL using ACL2. The script is a shell wrapper for `pprint-file.lisp`, which uses the subsidiary files `filter-forms.lisp` and `patch.lisp` (latter shouldn't be needed after ACL2 Version 2.9.3). The script is invoked by

`pprint-file.csh` *input−file output−file*

where *input-file* is a file to be pretty-printed and and *output-file* is the result. An example use is to run the following in the directory `acl2/examples`.

```
../lisp/pprint-file.csh pprint-example.lisp pprint-example.pp.lisp
```

The file `pprint-example.lisp` contains

```
; File created from HOL using print_lisp_file on Wed Dec 14 16:52:08 2005

(IN-PACKAGE "ACL2")

; Defun:    ACL2::EQUAL2_acl2_defun
(COMMON-LISP::DEFUN ACL2::EQUAL2 (ACL2::x ACL2::y) (ACL2::IF (ACL2::= ACL2::x
ACL2::y) COMMON-LISP::T COMMON-LISP::NIL))
```

and the result of pretty printing, `pprint-example.pp.lisp`, is:

```
(IN-PACKAGE "ACL2")

(DEFUN EQUAL2 (X Y) (AND (= X Y) T))
```

Appendix 2: Definitions of the ACL2 primitives in HOL

The definitions that follow are taked from the source file `sexpScript.sml` used to define the primitives. Text enclosed between `(*` and `*)` are comments. Each definition has the form:

```
val ML−name =
 acl2Define "ACL2−name" `HOL−defining−equation`
```

where *ML−name* is the metalanguage (i.e. ML) name of the theorem created by the definition, *ACL2−name* is the name of the primitive in ACL2 (written in fully expanded form to show the package) and *HOL−defining−equation* is a definition of the HOL4-friendly name used in SEXP for inputting the ACL2 name. Note that, as described earlier, it is actually the ACL2 name that is used as the constant name in the theory: `acl2Define` pre-processes the defining equation to replace the HOL4-friendly name by the proper ACL2 name before performing the definition.

```
(*****************************************************************************)
(* From axioms.lisp                                                          *)
(*                                                                           *)
(* ; (equal x x) = T                                                         *)
(*                                                                           *)
(* ; x/=y -> (equal x y) = NIL                                               *)
(*****************************************************************************)
val equal_def =
 acl2Define "COMMON-LISP::EQUAL"
  `equal (x:sexp) (y:sexp) = if x = y then t else nil`;

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

(*****************************************************************************)
(* characterp                                                                *)
(*****************************************************************************)
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)`;
```

```
(*****************************************************************************)
(* acl2-numberp                                                              *)
(*****************************************************************************)
val acl2_numberp_def =
 acl2Define "ACL2::ACL2-NUMBERP"
  `(acl2_numberp(num x) = t) /\ (acl2_numberp _ = nil)`;

(*****************************************************************************)
(* binary-+                                                                  *)
(*                                                                           *)
(* ; *1* definition:                                                         *)
(* (defun-*1* binary-+ (x y)                                                 *)
(*   (the number                                                             *)
(*        (if (numberp x)                                                    *)
(*            (if (numberp y)                                                *)
(*                (+ (the number x) (the number y))                          *)
(*              (gv binary-+ (x y) x))                                       *)
(*          (gv binary-+ (x y)                                               *)
(*              (if (numberp y)                                              *)
(*                  y                                                        *)
(*                0)))))                                                     *)
(*                                                                           *)
(* ; Hand-optimized ACL2:                                                    *)
(* (defun-*1* binary-+ (x y)                                                 *)
(*   (+ (if (numberp x) x 0) (if (numberp y) y 0)))                         *)
(*****************************************************************************)
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)`;

(*****************************************************************************)
(* [Note: space added between "(" and "*" to avoid confusing ML!]           *)
(*                                                                           *)
(* binary-*                                                                  *)
(*                                                                           *)
(* ; *1* definition:                                                         *)
(* (defun-*1* binary-* (x y)                                                 *)
(*   (the number                                                             *)
(*        (if (numberp x)                                                    *)
(*            (if (numberp y)                                                *)
(*                ( * x y)                                                   *)
(*              (gv binary-* (x y) 0))                                       *)
(*          (gv binary-* (x y) 0))))                                         *)
(*****************************************************************************)
val mult_def =
 acl2Define "ACL2::BINARY-*"
  `(mult (num x) (num y) = num(x*y)) /\ (mult _ _ = int 0)`;

(*****************************************************************************)
(* ; *1* definition (not very helpful)::                                     *)
(* (defun-*1* < (x y)                                                        *)
(*  (if (and (rationalp x)                                                   *)
(*           (rationalp y))                                                  *)
(*      (< (the rational x) (the rational y))                                *)
(*    (gv < (x y)                                                            *)
(*        (let ((x1 (if (numberp x) x 0))                                    *)
(*              (y1 (if (numberp y) y 0)))                                   *)
(*          (or (< (realpart x1) (realpart y1))                              *)
(*              (and (= (realpart x1) (realpart y1))                         *)
(*                   (< (imagpart x1) (imagpart y1))))))))                   *)
(*****************************************************************************)
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)`;

(*****************************************************************************)
(* unary--                                                                   *)
(*                                                                           *)
(* ; *1* definition:                                                         *)
(* (defun-*1* unary-- (x)                                                    *)
(*   (the number                                                             *)
(*        (if (numberp x)                                                    *)
(*            (- x)                                                          *)
(*          (gv unary-- (x) 0))))                                            *)
(*****************************************************************************)
val unary_minus_def =
 acl2Define "ACL2::UNARY--"
  `(unary_minus(num x) = num(COMPLEX_SUB com_0 x)) /\
   (unary_minus _      = int 0)`;

(*****************************************************************************)
(* unary-/                                                                   *)
(*                                                                           *)
(* ; *1* definition:                                                         *)
(* (defun-*1* unary-/ (x)                                                    *)
(*   (the number                                                             *)
(*        (if (and (numberp x) (not (= x 0)))                                *)
(*            (/ x)                                                          *)
(*          (gv unary-/ (x) 0))))                                            *)
(*                                                                           *)
(* ; Hand-optimized ACL2:                                                    *)
(* (defun-*1* unary-/ (x)                                                    *)
(*   (if (and (numberp x) (not (equal x 0)))                                 *)
(*       (/ x)                                                               *)
(*     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)`;

(*****************************************************************************)
(* consp                                                                     *)
(*                                                                           *)
(* ; *1* definition (not helpful):                                           *)
(* (defun-*1* consp (x)                                                      *)
(*   (consp x))                                                              *)
(*                                                                           *)
(*****************************************************************************)
val consp_def =
 acl2Define "COMMON-LISP::CONSP"
  `(consp(cons x y) = t) /\ (consp _ = nil)`;

(*****************************************************************************)
(* car                                                                       *)
```

```
(*                                                                          *)
(* ; *1* definition:                                                        *)
(* (defun-*1* car (x)                                                       *)
(*   (cond                                                                  *)
(*    ((consp x)                                                            *)
(*     (car x))                                                             *)
(*    ((null x)                                                             *)
(*     nil)                                                                 *)
(*    (t (gv car (x) nil))))                                                *)
(*                                                                          *)
(* ; Hand-optimized ACL2:                                                   *)
(* (defun-*1* car (x)                                                       *)
(*   (if (consp x) (car x) nil))                                            *)
(****************************************************************************)
val car_def =
 acl2Define "COMMON-LISP::CAR"
  `(car(cons x _) = x) /\ (car _ = nil)`;

(****************************************************************************)
(* cdr                                                                      *)
(*                                                                          *)
(* ; *1* definition:                                                        *)
(* (defun-*1* cdr (x)                                                       *)
(*   (cond                                                                  *)
(*    ((consp x)                                                            *)
(*     (cdr x))                                                             *)
(*    ((null x)                                                             *)
(*     nil)                                                                 *)
(*    (t (gv cdr (x) nil))))                                                *)
(*                                                                          *)
(* ; Hand-optimized ACL2:                                                   *)
(* (defun-*1* cdr (x)                                                       *)
(*   (if (consp x) (cdr x) nil))                                            *)
(****************************************************************************)
val cdr_def =
 acl2Define "COMMON-LISP::CDR"
  `(cdr(cons _ y) = y) /\ (cdr _ = nil)`;

(****************************************************************************)
(* realpart                                                                 *)
(*                                                                          *)
(* ; *1* definition:                                                        *)
(* (defun-*1* realpart (x)                                                  *)
(*   (if (numberp x)                                                        *)
(*       (realpart x)                                                       *)
(*     (gv realpart (x) 0)))                                                *)
(****************************************************************************)
val realpart_def =
 acl2Define "COMMON-LISP::REALPART"
  `(realpart(num(com a b)) = num(com a rat_0)) /\
   (realpart _            = int 0)`;

(****************************************************************************)
(* imagpart                                                                 *)
(*                                                                          *)
(* ; *1* definition:                                                        *)
(* (defun-*1* imagpart (x)                                                  *)
(*   (if (numberp x)                                                        *)
(*       (imagpart x)                                                       *)
(*     (gv imagpart (x) 0)))                                                *)
(****************************************************************************)
val imagpart_def =
 acl2Define "COMMON-LISP::IMAGPART"
  `(imagpart(num(com a b)) = num(com b rat_0)) /\
   (imagpart _            = int 0)`;

(****************************************************************************)
(* rationalp                                                                *)
```

```
(*                                                                        *)
(* ; *1* definition (not helpful):                                        *)
(* (defun-*1* rationalp (x)                                               *)
(*   (rationalp x))                                                       *)
(**************************************************************************)
val rationalp_def =
 acl2Define "COMMON-LISP::RATIONALP"
  `(rationalp(num(com a b)) = if b = rat_0 then t else nil) /\
   (rationalp _             = nil)`;

(**************************************************************************)
(* complex-rationalp                                                      *)
(*                                                                        *)
(* Complex-rationalp holds only of numbers that are not rational, i.e.,   *)
(* have a non-zero imaginary part.  Here is the ACL2 documentation for    *)
(* COMPLEX-RATIONALP.                                                     *)
(*                                                                        *)
(* COMPLEX-RATIONALP    recognizes complex rational numbers               *)
(*                                                                        *)
(* Examples:                                                              *)
(*  (complex-rationalp 3)       ; nil, as 3 is rational, not complex rational*)
(*  (complex-rationalp #c(3 0)) ; nil, since #c(3 0) is the same as 3     *)
(*  (complex-rationalp t)       ; nil                                     *)
(*  (complex-rationalp #c(3 1)) ; t, as #c(3 1) is the complex number 3 + i  *)
(**************************************************************************)
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)`;

(**************************************************************************)
(* complex                                                                *)
(*                                                                        *)
(* ; *1* definition:                                                      *)
(* (defun-*1* complex (x y)                                               *)
(*   (complex (the rational (if (rationalp x) x (gv complex (x y) 0)))    *)
(*            (the rational (if (rationalp y) y (gv complex (x y) 0)))))  *)
(*                                                                        *)
(* COMPLEX    create an ACL2 number                                       *)
(*                                                                        *)
(*      Examples:                                                         *)
(*      (complex x 3) ; x + 3i, where i is the principal square root of -1  *)
(*      (complex x y) ; x + yi                                            *)
(*      (complex x 0) ; same as x, for rational numbers x                 *)
(*                                                                        *)
(* The function complex takes two rational number arguments and returns an  *)
(* ACL2 number.  This number will be of type (complex rational) [as defined  *)
(* in the Common Lisp language], except that if the second argument is   *)
(* zero, then complex returns its first argument.  The function           *)
(* complex-rationalp is a recognizer for complex rational numbers, i.e. for  *)
(* ACL2 numbers that are not rational numbers.                           *)
(*                                                                        *)
(* The reader macro #C (which is the same as #c) provides a convenient way  *)
(* for typing in complex numbers.  For explicit rational numbers x and y,  *)
(* #C(x y) is read to the same value as (complex x y).                    *)
(*                                                                        *)
(* The functions realpart and imagpart return the real and imaginary parts  *)
(* (respectively) of a complex (possibly rational) number.  So for example,  *)
(* (realpart #C(3 4)) = 3, (imagpart #C(3 4)) = 4, (realpart 3/4) = 3/4,  *)
(* and (imagpart 3/4) = 0.                                                *)
(*                                                                        *)
(* The following built-in axiom may be useful for reasoning about complex  *)
(* numbers.                                                               *)
(*                                                                        *)
(*      (defaxiom complex-definition                                      *)
(*        (implies (and (real/rationalp x)                                *)
(*                      (real/rationalp y))                               *)
```

```
(*                (equal (complex x y)                                    *)
(*                       (+ x ( * #c(0 1) y))))                           *)
(*       :rule-classes nil)                                               *)
(*                                                                        *)
(* A completion axiom that shows what complex returns on arguments        *)
(* violating its guard (which says that both arguments are rational       *)
(* numbers) is the following.                                             *)
(*                                                                        *)
(*      (equal (complex x y)                                              *)
(*             (complex (if (rationalp x) x 0)                            *)
(*                      (if (rationalp y) y 0)))                          *)
(**************************************************************************)
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)`;


(**************************************************************************)
(* integerp                                                               *)
(*                                                                        *)
(* ; *1* definition (not helpful):                                        *)
(* (defun-*1* integerp (x)                                                *)
(*   (integerp x))                                                        *)
(**************************************************************************)
val integerp_def =
 acl2Define "COMMON-LISP::INTEGERP"
  `(integerp(num n) = if IS_INT n then t else nil) /\
   (integerp _      = nil)`;

(**************************************************************************)
(* numerator                                                              *)
(*                                                                        *)
(* ; *1* definition::                                                     *)
(* (defun-*1* numerator (x)                                               *)
(*   (if (rationalp x)                                                    *)
(*       (numerator x)                                                    *)
(*     (gv numerator (x) 0)))                                             *)
(*                                                                        *)
(* ; Hand-optimized ACL2:                                                 *)
(* (defun-*1* numerator (x)                                               *)
(*   (if (rationalp x)                                                    *)
(*       (numerator x)                                                    *)
(*     0))                                                                *)
(**************************************************************************)
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)`;

(**************************************************************************)
(* denominator                                                            *)
(*                                                                        *)
(* ; *1* definition::                                                     *)
(* (defun-*1* denominator (x)                                             *)
(*   (if (rationalp x)                                                    *)
(*       (denominator x)                                                  *)
```

```
(*     (gv denominator (x) 1)))                                          *)
(*                                                                       *)
(* ; Hand-optimized ACL2:                                                *)
(* (defun-*1* denominator (x)                                            *)
(*   (if (rationalp x)                                                   *)
(*       (denominator x)                                                 *)
(*     1))                                                               *)
(*************************************************************************)
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)`;

(*************************************************************************)
(* char-code                                                             *)
(*                                                                       *)
(* ; *1* definition                                                      *)
(* (defun-*1* char-code (x)                                              *)
(*   (if (characterp x) (char-code x) (gv char-code (x) 0)))             *)
(*************************************************************************)
val char_code_def =
 acl2Define "COMMON-LISP::CHAR-CODE"
  `(char_code(chr c) = int (&(ORD c))) /\ (char_code _ = int 0)`;

(*************************************************************************)
(* code-char                                                             *)
(*                                                                       *)
(* ; *1* definition:                                                     *)
(* (defun-*1* code-char (x)                                              *)
(*   (if (and (integerp x)                                               *)
(*            (>= x 0)                                                   *)
(*            (< x 256))                                                 *)
(*       (code-char x)                                                   *)
(*     (gv code-char (x) (code-char 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))`;

(*************************************************************************)
(* if                                                                    *)
(*                                                                       *)
(* From axioms.lisp                                                      *)
(*                                                                       *)
(* ;         (if NIL y z) = z                                            *)
(*                                                                       *)
(* ; x/=NIL -> (if x y z) = y                                            *)
(*                                                                       *)
(* Can't overload "if" onto ``ACL2_IF`` because of HOL's                 *)
(* `if ... then ... else` construct. Using "acl2_if" instead.            *)
(*************************************************************************)
val acl2_if_def =
 acl2Define "COMMON-LISP::IF"
  `acl2_if 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.   *)
(*                                                                       *)
(* list_to_sexp maps a function down a HOL list and then conses up an    *)
(* s-expression from the resulting list. For example:                    *)
(*                                                                       *)
```

```
(* 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 nee 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.                                          *)
(*                                                                          *)
(* ; *1* definition:                                                        *)
(* (defun-*1* coerce (x y)                                                  *)
(*   (cond                                                                  *)
(*    ((equal y 'list)                                                      *)
(*     (if (stringp x)                                                      *)
(*         (coerce x 'list)                                                 *)
(*       (gv coerce (x y) nil)))                                            *)
(*    ((character-listp x)                                                  *)
(*     (if (equal y 'string)                                                *)
(*         (coerce x 'string)                                               *)
(*       (gv coerce (x y) (coerce x 'string))))                             *)
(*    (t                                                                    *)
(*     (gv coerce (x y)                                                     *)
(*         (coerce (make-character-list x) 'string)))))                     *)
(****************************************************************************)
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)`;

(****************************************************************************)
(* 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`;

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

(***************************************************************************)
(* symbol-name                                                             *)
(*                                                                         *)
(* ; *1* definition:                                                       *)
(* (defun-*1* symbol-name (x)                                              *)
(*   (if (symbolp x)                                                       *)
(*       (symbol-name x)                                                   *)
(*     (gv symbol-name (x) "")))                                           *)
(***************************************************************************)
val symbol_name_def =
 acl2Define "COMMON-LISP::SYMBOL-NAME"
  `(symbol_name (sym p n) = acl2_if (symbolp (sym p n)) (str n) (str ""))
   /\
   (symbol_name _ = (str ""))`;

(***************************************************************************)
```

```
(* symbol-package-name                                                      *)
(*                                                                          *)
(* ; *1* definition:                                                        *)
(* (defun-*1* symbol-package-name (x)                                       *)
(*   (if (symbolp x)                                                        *)
(*       (symbol-package-name x)                                            *)
(*     (gv symbol-package-name (x) "")))                                    *)
(****************************************************************************)
val symbol_package_name_def =
 acl2Define "ACL2::SYMBOL-PACKAGE-NAME"
  `(symbol_package_name (sym p n) =
     acl2_if (symbolp (sym p n)) (str p) (str ""))
   /\
   (symbol_package_name _ = (str ""))`;

(****************************************************************************)
(* pkg-witness                                                              *)
(*                                                                          *)
(* ; *1* definition (not very helpful):                                     *)
(* (defun-*1* pkg-witness (pkg)                                             *)
(*   (if (stringp pkg)                                                      *)
(*       (if (find-non-hidden-package-entry                                 *)
(*            pkg (known-package-alist *the-live-state* ))                  *)
(*           (intern *pkg-witness-name* pkg)                                *)
(*         (throw-raw-ev-fncall (list 'pkg-witness-er pkg)))                *)
(*     (gv pkg-witness (pkg) nil)))                                         *)
(*                                                                          *)
(* ; Here, we treat the ACL2 constant *pkg-witness-name* as its value,      *)
(* ; the string "ACL2-PKG-WITNESS" -- in fact, ACL2 treates constants       *)
(* ; (defconst events) much as it treats macros, in the sense that they     *)
(* ; are eliminated when passing to internal terms.                         *)
(*                                                                          *)
(* ; 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 "ACL2-PKG-WITNESS" x in acl2_if (symbolp s) s nil`;

(****************************************************************************)
(* intern-in-package-of-symbol                                              *)
(*                                                                          *)
(* ; *1* definition::                                                       *)
(* (defun-*1* intern-in-package-of-symbol (x y)                             *)
(*   (if (and (stringp x)                                                   *)
(*            (symbolp y))                                                  *)
(*       (intern x (symbol-package y))                                      *)
(*     (gv intern (x y) nil)))                                              *)
(*                                                                          *)
(* ; Hand-optimized ACL2:                                                   *)
(* (defun-*1* intern-in-package-of-symbol (x y)                             *)
(*   (if (and (stringp x) (symbolp y)) (intern x (symbol-package y)) 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) =
     acl2_if (symbolp (sym p n)) (BASIC_INTERN x p) nil)
   /\
   (intern_in_package_of_symbol _ _ = nil)`;
```