

Version 1.5

Connecting HOL to ACL2

Michael J.C. Gordon¹, Warren A. Hunt, Jr.² and Matt Kaufmann²

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 that is 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 HOL4 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 an equivalent 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 too. 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 up "translation specifications" that are pattern-matching rewrite rules to per-

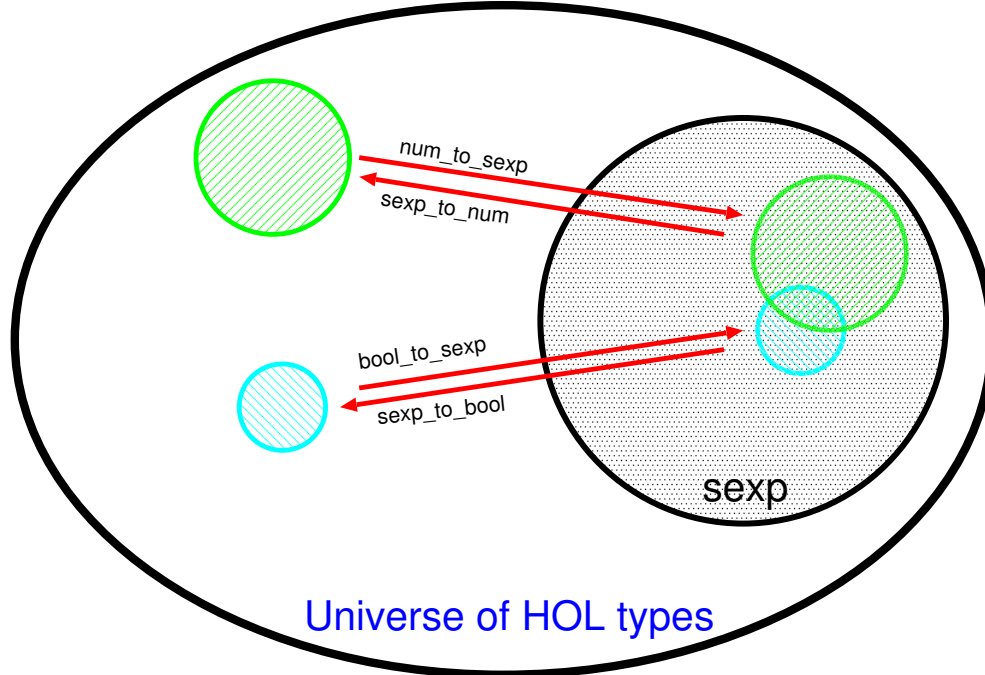
¹ George Fink, Myla Archer, and Lie Yang, PM: A Proof Manager for HOL and Other provers, Proceedings of the 1991 International Workshop on the HOL theorem proving system and its applications, IEEE Computer Society Press, 1991.

form 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 in our approach we mechanically prove the correctness of translating formulas between ACL2 and HOL using a formal model of the ACL2 logic represented in HOL. Thus scripts of PM or ACL2PII are replaced by deductions in the HOL4 system.

The idea of our approach (see 3.1 for more detail) 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. A subset of HOL datatypes (e.g. booleans, strings, enumerations) can then be translated into s-expressions represented in HOL.

For example, we can illustrate embedding functions `num_to_sexp` mapping HOL numbers to s-expressions (with inverse `sexp_to_num`) and `bool_to_sexp` mapping HOL truth-values to s-expressions (inverse `sexp_to_bool`) with the following diagram:



This diagram just shows how two ground types (`num` and `bool`) in HOL are embedded into s-expressions, represented as a type in HOL. Functions operating on such `sexp`-embeddable types (e.g. `num` and `bool`) can be encoded as functions operating on s-expressions, and each time an encoding is generated its correctness can be validated by proof in HOL4 (see Section 5 for examples).

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 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 – possibly it could be the ACL2 definition of `EVAL`?

After consulting various people 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.

3.1 Formalization Levels

We have chosen to create a layer between the typical use of HOL and the usual approach to using ACL2 that we call the `SEXP` level. `SEXP` is a theory in HOL that defines constants corresponding the the ACL2 primitive constants and functions. All constants in `SEXP` have type `sexp` or `sexp → ... → sexp → sexp`, i.e. all functions are curried first-order operations on the HOL `sexp` datatype. This recursive datatype is composed of four kinds of atoms (symbols, strings, characters and natural numbers²) and pairs, whose components are s-expressions.

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.

² The number atoms will eventually be replaced by ACL2's complex rationals.

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‘);

(*****
(* 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 num
        | ACL2_PAIR of sexp => sexp’; (* only curried for style *)
```

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

```
ACL2_SYMBOL : string -> string -> sexp
ACL2_STRING : string -> sexp
ACL2_CHARACTER : char -> sexp
ACL2_NUMBER : num -> sexp
ACL2_PAIR : sexp -> sexp -> sexp
```

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)`). Note that the HOL type `char` only has 256 values, so exactly matches ACL2’s characters.

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.

We now give the start of the development of the definitions of ACL2 primitives in HOL. The lexicographic rules for identifiers in HOL and ACL2 are different (e.g. in ACL2 “-” can occur in a name, but not in HOL). We therefore maintain a table showing equivalent names in the two systems. In general, we always define HOL constants representing ACL2 objects with a name prefixed by `ACL2_`. We then normally introduce a shorter mnemonic HOL name (usually lower case) that the HOL4 parser and printer can use (e.g. `cons` is introduced as the short name for `ACL2_PAIR`). This naming methodology might change with more experience. An ML function `declare_names` has been implemented to add entries to the table recording the various names. Evaluating:

```
declare_names (HOL-constant, (short-HOL-name, ACL2-name));
```

declares that *HOL-constant* (a term) corresponds to *short-HOL-name* (an ML string representing the short HOL name) and to *ACL2-name* (an ML string representing the ACL2 name). Our convention is to use upper case for ACL2 names. For example:

```
declare_names ('ACL2_PAIR', ("cons", "CONS"));
```

records that the a short name of the constant `ACL2_PAIR` is `cons`, and also declares that the ACL2 name is `CONS`. The ACL2 name is used when printing HOL in a form for input to ACL2 (see below).

The four constructor constants `ACL2_SYMBOL`, `ACL2_NUMBER`, `ACL2_STRING` and `ACL2_CHARACTER` are given short names `sym`, `nat`, `str`, `chr` in HOL, respectively. They are not given ACL2 names because the printer that creates s-expressions for input into ACL2 recognises atoms as special cases, and prints `ACL2_SYMBOL pkg name` as `pkg::name`, `ACL2_NUMBER n` as `n`, `ACL2_STRING s` as `s` and `ACL2_CHARACTER c` as `c`.

The first primitives we define are the special ACL2 symbols `NIL` and `T`. Constants are defined in HOL4 using the ML function `Define` used below.³

```
Define 'ACL2_NIL = sym "COMMON-LISP" "NIL";
Define 'ACL2_T   = sym "COMMON-LISP" "T";

declare_names ('ACL2_NIL', ("nil", "NIL"));
declare_names ('ACL2_T', ("t", "T"));
```

This defines HOL constants `ACL2_NIL` and `ACL2_T`, both of type `sexp`, with parser-printer abbreviations `nil` and `t`, respectively.

Next we define ACL2's `CONSP` predicate:

```
Define
'(ACL2_CONSP(ACL2_SYMBOL pname name) = nil) /\
 (ACL2_CONSP(ACL2_STRING s) = nil) /\
 (ACL2_CONSP(ACL2_CHARACTER c) = nil) /\
 (ACL2_CONSP(ACL2_NUMBER n) = nil) /\
 (ACL2_CONSP(ACL2_PAIR x y) = t)';

declare_names ('ACL2_CONSP', ("consp", "CONSP"));
```

This definition shows that pairs are mapped to `t` and all other s-expressions (i.e. the four kinds of atoms) are mapped to `nil`. HOL4 provides a more 'ML-like' way of writing such definitions that uses pattern matching. The definition of `CONSP` could have been written more compactly as:

```
Define '(ACL2_CONSP(cons x y) = t) /\ (ACL2_CONSP _ = nil)';
```

where the conjuncts are matched left-to-right, and `'_'` matches any value. We will use the compact form from now on.

The definitions of ACL2's `CAR` and `CDR` are straightforward.

³ The principles underlying HOL's various kinds of quotations, namely `'...'`, `'...'` and `"..."`, are not explained here, except to note that the first two denote terms in higher order logic and the third denotes string literals in the metalanguage ML.

```

Define '(ACL2_CAR(cons x y) = x) /\ (ACL2_CAR _ = nil)';
Define '(ACL2_CDR(cons x y) = y) /\ (ACL2_CDR _ = nil)';

declare_names (('ACL2_CAR', ("car", "CAR"));
declare_names (('ACL2_CDR', ("cdr", "CDR"));

```

Note that `car` and `cdr` map atoms to `nil`.

To define the ACL2 primitives `IF` and `EQUAL` we first note that `axioms.lisp` gives the informal semantics as follows:

```

; The following four axioms define if and equal but are not expressed
; in the ACL2 language.

;      (if NIL y z) = z

; x/=NIL -> (if x y z) = y

; (equal x x) = T

; x/=y -> (equal x y) = NIL

```

The symbols `=`, `/=` and `->` are *pseudo-symbols* and are part of the meta-language used by Kaufmann and Moore to specify the semantics of the ACL2 logic.⁴ We are formally specifying the semantics in higher order logic, and so these pseudo-symbols can be represented by the corresponding logical notions in HOL. Thus we define:

```

Define 'ACL2_IF x (y:sexp) (z:sexp) = if x = nil then z else y';
Define 'ACL2_EQUAL (x:sexp) (y:sexp) = if x = y then t else nil';

```

The reason that the formal parameters are explicitly typed is technical (to prevent HOL4 inferring too general types). We have used HOL's conditional notation `if ... then ... else` here. To avoid confusing HOL4's parser, we will abbreviate ACL2's `if` by `acl2_if`.

```

declare_names (('ACL2_IF', ("acl2_if", "IF"));
declare_names (('ACL2_EQUAL', ("equal", "EQUAL"));

```

Various other ACL2 functions can now be defined:

```

Define 'ACL2_IMPLIES p q = acl2_if p (acl2_if q t nil) t';
Define 'ACL2_NOT p      = acl2_if p nil t';
Define 'ACL2_ATOM x     = not(consp x)';
Define 'ACL2_ENDP x     = atom x';

declare_names (('ACL2_IMPLIES', ("implies", "IMPLIES"));
declare_names (('ACL2_NOT', ("not", "NOT"));
declare_names (('ACL2_ATOM', ("atom", "ATOM"));
declare_names (('ACL2_ENDP', ("endp", "ENDP"));

```

⁴ Matt Kaufmann and J Strother Moore, "A Precise Description of the ACL2 Logic", Section 5.

3.3 Numbers

Currently we only represent natural numbers, but we plan to support ACL2's complex rationals eventually. In ACL2, numbers are specified by a mixture of axioms in `axioms.lisp` (e.g. associativity and commutativity of `+` and `*`) and code in the built-in linear arithmetic decision procedure. In HOL, numbers are defined using a construction based on classical methods. Our approach is to define the ACL2 versions of arithmetical functions using the native HOL versions. We will then try to prove all the axioms of the ACL2 logic in `axioms.lisp`. 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. The key property we must ensure is that anything provable in ACL2 is provable in the SEXP theory in HOL.

The function `nat` (which is the HOL short name for `ACL2_NUMBER`) converts a HOL number to an s-expression; thus `nat : num -> sexp`. The inverse function is (partially) defined by:

```
Define '∀n. sexp_to_num(nat n) = n'
```

The HOL type of `sexp_to_num` is `sexp -> num`. We give below our development of numbers so far.

```
Define
  '(ACL2_NFIX(nat n) = nat n) /\ (ACL2_NFIX _ = nat 0)';
declare_names ('ACL2_NFIX', ("nfix", "NFIX"));

Define
  '(ACL2_NATP(nat _) = t) /\ (ACL2_NATP _ = nil)';
declare_names ('ACL2_NATP', ("natp", "NATP"));

Define
  'ACL2_BINARY_NAT_ADD x y = nat(sexp_to_num(nfix x) + sexp_to_num(nfix y))';
declare_names ('ACL2_BINARY_NAT_ADD', ("add", "+"));

Define
  'ACL2_BINARY_NAT_SUB x y = nat(sexp_to_num(nfix x) - sexp_to_num(nfix y))';
declare_names ('ACL2_BINARY_NAT_SUB', ("sub", "-"));

Define
  'ACL2_NAT_LESS x y =
    if sexp_to_num(nfix x) < sexp_to_num(nfix y) then t else nil';
declare_names ('ACL2_NAT_LESS', ("less", "<"));

Define
  'ACL2_NAT_LESS_EQ x y = not(less y x)';
declare_names ('ACL2_NAT_LESS_EQ', ("leq", "<="));

Define
  'ACL2_ZP x = acl2_if (natp x) (leq x (nat 0)) t';
declare_names ('ACL2_ZP', ("zp", "ZP"));
```

Note that when we have complex rationals in `SEXP`, then `natp` will be replaced by `integerp` in the definition of `ACL2_ZP`.

4 Semantics of ACL2 terms and formulas

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. We therefore make the definition:

```
Define '(|-- p) = (not(equal p nil) = t)'
```

The occurrences of '=' here are HOL's equality. If `p` is a HOL term of type `sexp` then `|-- p` is a HOL term of type `bool` (i.e. a HOL formula).

We can now sanity check our definitions by verifying in HOL4 some simple properties, including axioms given in `axioms.lisp`.

```
∀p. |-- ( |-- p) = ¬(p = nil)
```

```
∀x. |-- implies (cons x) (equal (cons(car x) (cdr x)) x)
```

```
∀x y. |-- equal (cdr (cons x y)) y
```

Our goal is eventually to verify all the axioms of ACL2 given in `axioms.lisp` to ensure that our formalization of the ACL2 logic is completely accurate.

5 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. 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 auxiliary 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.

5.1 Explicit data

HOL and `ACL2` each have their own notions of numbers, characters and strings, though HOL sometimes has stronger axiomatisations than `ACL2`. For example HOL natural numbers do not include the ‘non-standard numbers’ that are possible because of `ACL2`’s first-order axiomatisation. However, as discussed earlier in 3.3, we do not consider this a problem. Fortunately the match between characters and strings in HOL and `ACL2` is perfect.

For every explicit value that we wish to write functions to manipulate, we must provide a bi-directional mapping from HOL to `ACL2` and back again. To implement this, we defined a means to represent s-expressions with a single HOL datatype `sexp`. We have implemented a printer that permits us to take a HOL term representing an s-expression and print it into the syntax corresponding to `ACL2`’s representation of the same explicit value. Our printer can print both to the interactive session or to a file. Here is an example showing the ML function `pr-sexp` which prints a term interactively (‘-’ is the ML prompt):

```
- pr-sexp ‘(cons (nat 1) (cons (cons (str "foo") (nat 2)) (cons (nat 3) nil)))’;
(1 ("foo" . 2) 3)
```

We have implemented prototype tools for moving s-expressions between HOL and `ACL2`. A tool coded in HOL can create a file that is suitable for reading by `ACL2` and a second tool, implemented in `ACL2`, creates files that can be read into HOL.

5.2 Definitions

The next class of objects that we wish to be able to transfer from HOL to `ACL2` is definitions. `ACL2` functions take s-expressions as actual parameters and produce a result s-expression. Once again, we use the `SEXP` theory as a means to bridge between typical HOL definitions and typical `ACL2` definitions. We have written a function that converts a HOL definition to an s-expression, and then we can use our data printer to print this in a form that `ACL2` can read. For example, recall:

```
Define
  ‘ACL2_ZP x = acl2_if (natp x) (leq x (nat 0)) t’;
declare_names (‘ACL2_ZP’, ("zp", "ZP"));
```

If the HOL theorem created by `Define` (a definitional axiom) is named `ACL2_ZP_def` in ML, then using `print_sexp_defun` (which prints a HOL definitional axiom as an s-expression) we have:

```
- print_sexp_defun ACL2_ZP_def;
(DEFUN ZP (x) (IF (NATP x) (<= x 0) T))
```

Note that the ACL2 names of `acl2_if`, `natp` and `leq` have been used, and `nat 0` is printed as 0.

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`, using `TFL` or it could be done in `ACL2`. In the latter case, we envision transferring a proposed 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 have not yet implemented facilities for admitting HOL `SEXP` definitions using `ACL2`, and so all our examples so far have been admitted by manual proof in `HOL4` (though we expect simple recursions over s-expressions can be admitted automatically using an extension of `TFL`). We use `acl2Define` in what follows instead of `Define` to show when a definition has been admitted via a manual termination proof.

Example: unary addition

Consider the following definition in `HOL4` of addition by repeated incrementing:

```
Define 'pplus x y = if x=0 then y else 1 + (pplus (x - 1) y)';
```

This function has type `num -> num -> num` in `HOL` (the name is a mnemonic for 'Peano Plus'). We can define in `HOL` a corresponding function that operates on s-expressions:

```
acl2Define
  'ACL2_PPLUS x y =
    (acl2_if (zp x)
             (nfix y)
             (add (nat 1) (ACL2_PPLUS (sub x (nat 1)) y)))'
```

This function has type `sexp -> sexp -> sexp` in `HOL`. It can be proved to be correct by proving a simple theorem:

```
∀m n. pplus m n = sexp_to_num(ACL2_PPLUS (nat m) (nat n))
```

Our printer uses a simple recursive algorithm to convert the HOL definition of `ACL2_PPLUS` to the following `ACL2` definition:

```
(DEFUN PPLUS (x y) (IF (ZP x) (NFIX y) (+ 1 (PPLUS (- x 1) y))))
```

We can thus evaluate a HOL term `pplus m n` by evaluating `(PPLUS m n)` in `ACL2`.

The soundness of moving from `pplus` of type `num->num->num` to `ACL2_PPLUS` of type `sexp->sexp->sexp` is justified by the simple theorem above, but the

soundness of the correspondence between `ACL2_PPLUS` in HOL and `PPLUS` in `ACL2` depends on our printer correctly converting the definition of `ACL2_PPLUS` to an `ACL2` definition. The printer is thus ‘critical code’, and is optimized for clarity rather than efficiency.

Example: appending lists

In the previous example we related HOL and `ACL2` functions operating on numbers. We now consider a simple example of relating functions operating on lists. HOL lists are terms of type $(\sigma)\text{list}$, where σ is the type of elements of the list (unlike in `ACL2`, all elements in HOL lists must have the same type). Literal lists have the form $[x_1; \dots; x_n]$ (the empty list is `NIL` or equivalently `[]`). The `cons`-operator in HOL is an infix `::` and is the primitive list constructor. Explicit lists are thus represented as `conses`, so the term ‘`1::2::3::4::NIL`’ is identical to the term ‘`[1;2;3;4]`’. The `append` operator is a defined function and is predefined as an infix `++`. This is all illustrated by:

```
- ‘1::[2;3;4]’;
> val it = ‘[1; 2; 3; 4]’ : term

- ‘[1;2]++[3;4]’;
> val it = ‘[1; 2] ++ [3; 4]’ : term

- EVAL ‘[1;2]++[3;4]’;
> val it = |- [1; 2] ++ [3; 4] = [1; 2; 3; 4] : thm
```

The function `EVAL` is an optimized rewriter that simplifies terms using definitions of the constants occurring in them. The ML responses

```
> val it = ...
```

are part of the ML read-eval-print loop and say that the value of the last expression evaluated at top-level is bound to the ML name `it`.

If `f` is a HOL function that maps to s-expressions, then we can define a higher order function `list_to_sexp` such that `list_to_sexp f` maps lists of values in `f`’s domain to s-expressions:

```
Define
  ‘(list_to_sexp f [] = nil) /\
   (list_to_sexp f (x::l) = cons (f x) (list_to_sexp f l))’;
```

For example:

```
- EVAL ‘list_to_sexp nat [1; 2; 3]’;
> val it =
  |- list_to_sexp nat [1; 2; 3] =
     cons (nat 1)
       (cons (nat 2) (cons (nat 3) (sym "COMMON-LISP" "NIL"))) : thm
```

The HOL type of `list_to_sexp` is $(\alpha \rightarrow \text{sexp}) \rightarrow \alpha \text{ list} \rightarrow \text{sexp}$, where α is a type variable (actually printed as ‘`a`’ in HOL).

To go from s-expressions to HOL, we define `sexp_to_list` by:

```
Define
  ‘(sexp_to_list g (cons x y) = g x :: sexp_to_list g y) /\
   (sexp_to_list g _ = [])’;
```

The HOL type of `sexp_to_list` is $(sexp \rightarrow \alpha) \rightarrow sexp \rightarrow \alpha \text{ list}$. As a sanity check it is easy to prove in HOL that:

```
 $\forall f\ g. (\forall x. g(f\ x) = x) \implies \forall l. \text{sexp\_to\_list } g\ (\text{list\_to\_sexp } f\ l) = l$ 
```

Let us now define an s-expression appending function:

```
acl2Define
  'ACL2_BINARY_APP x y =
    acl2_if (endp x) y (cons (car x) (ACL2_BINARY_APP (cdr x) y))'
declare_names (('ACL2_BINARY_APP', ("app", "APP"));
```

Using our definition printer we can automatically create:

```
- print_sexp_defun ACL2_BINARY_APP;
(DEFUN APP (x y) (IF (ENDP x) y (CONS (CAR x) (APP (CDR x) y))))
```

To verify that `ACL2_BINARY_APP` (which has short name `app` in HOL) is a correct implementation of HOL's append function `++`, we can prove:

```
 $\forall f\ g. (\forall x. g(f\ x) = x) \implies \forall u\ v. u\ ++\ v = \text{sexp\_to\_list } g\ (\text{app } (\text{list\_to\_sexp } f\ u)\ (\text{list\_to\_sexp } f\ v))$ 
```

If we specialise `f`, `g` to `nat`, `sexp_to_num`, respectively, and simplify, we get:

```
 $\forall u\ v. u\ ++\ v = \text{sexp\_to\_list } \text{sexp\_to\_num } (\text{app } (\text{list\_to\_sexp } \text{nat } u)\ (\text{list\_to\_sexp } \text{nat } v))$ 
```

which shows that to compute `u ++ v` one can convert `u` and `v` to s-expressions using `list_to_sexp nat`, apply `APP` in `ACL2` and then convert the resulting s-expression back to a HOL list using `sexp_to_list sexp_to_num`.

This example proof shows that by mapping HOL data objects into HOL `sexp` terms, applying the newly introduced HOL function, and finally, mapping the result `sexp` term or terms back into HOL data objects, produces the same result. This step is introduced to reduce the *semantic gap* between HOL definitions and data and `ACL2` definitions and data. Once such a definition is successfully introduced in HOL, we use the same printer mentioned above and print `ACL2` definitions that correspond to the HOL definitions that operate on HOL `sexp` terms.

5.3 Theorems

The third class of objects we want to pass back and forth between `ACL2` and HOL are theorems. The main scenario we are considering is to use `ACL2` to evaluate ground terms and to solve HOL goals in `SEXP`, as described above.

6 Future Work

We need to complete the definitions of the HOL functions corresponding to the primitive ACL2 functions, and we need to verify that these HOL functions satisfy all the axioms of ACL2 given in the ACL2 source file `axioms.lisp`.

We need to do more examples in order to better develop our flow and associated tool-chain. 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.

7 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 the user of these two systems can used together will be an important factor in the eventual usefulness of such a combined system.

Appendix: some translated definitions

For visual sanity checking, here is a selection of HOL definitions and the resulting ACL2 output generated by our printer. For readability, we have manually inserted spaces and line-breaks manually as our printer prints each ACL2 definition as a single line. No other changes have been made. A file-to-file shell command `pprint-file.csh` has been implemented that uses ACL2 to reformat a file containing a sequence of definitions into a pretty-printed form. We do not use the output of this here since it puts everything in upper-case (which is equivalent in ACL2).

In the following list of HOL definitions the exclamation mark `!` denotes the universal quantifier \forall . The turnstile symbol `|-` is printed by `HOL4` in front of theorem values (which can be created via definitions or by proof).

```
> val Examples =
  [|- !p q. implies p q = acl2_if p (acl2_if q t nil) t,
   |- !x. atom x = not (consp x),
   |- !x. endp x = atom x,
   |- !x. true_listp x = acl2_if (consp x) (true_listp (cdr x)) (eq x nil),
   |- !x. zp x = acl2_if (natp x) (leq x (nat 0)) t,
   |- !x y.
     acl2_pplus x y =
       acl2_if (zp x) (nfix y)
         (add (nat 1) (acl2_pplus (sub x (nat 1)) y)),
   |- !x y. app x y = acl2_if (endp x) y (cons (car x) (app (cdr x) y)),
   |- !p q. or2 p q = acl2_if p t q, |- !x y. nand x y = not (and2 x y),
```

```

|- !x. len x = acl2_if (consp x) (add (nat 1) (len (cdr x))) (nat 0),
|- !lst.
  boolean_listp lst =
  acl2_if (atom lst) (eq lst nil)
    (and2 (or2 (eq (car lst) t) (eq (car lst) nil))
      (boolean_listp (cdr lst))),
|- !a b. xor a b = acl2_if a (acl2_if b nil t) (acl2_if b t nil),
|- !a b c. xor3 a b c = xor a (xor b c),
|- !a b.
  v_xor a b =
  acl2_if (atom a) nil
    (cons (xor (car a) (car b)) (v_xor (cdr a) (cdr b))),
|- !a b c. b_carry a b c = acl2_if a (or2 b c) (and2 b c),
|- !c a b.
  v_adder c a b =
  acl2_if (atom a) (cons c nil)
    (cons (xor3 c (car a) (car b))
      (v_adder (b_carry c (car a) (car b)) (cdr a) (cdr b))))] :
thm list

```

The file created by printing the ML list Examples shown above is:

```

; File created from HOL using print_lisp_file on Wed Jul 27 12:08:52 2005
(IN-PACKAGE "ACL2")
(DEFUN IMPLIES (p q) (IF p (IF q T NIL) T))
(DEFUN ATOM (x) (NOT (CONSP x)))
(DEFUN ENDP (x) (ATOM x))
(DEFUN TRUE-LISTP (x) (IF (CONSP x) (TRUE-LISTP (CDR x)) (EQ x NIL)))
(DEFUN ZP (x) (IF (NATP x) (<= x 0) T))
(DEFUN PPLUS (x y) (IF (ZP x) (NFIX y) (+ 1 (PPLUS (- x 1) y))))
(DEFUN APP (x y) (IF (ENDP x) y (CONS (CAR x) (APP (CDR x) y))))
(DEFUN OR (p q) (IF p T q))
(DEFUN NAND (x y) (NOT (AND x y)))
(DEFUN LEN (x) (IF (CONSP x) (+ 1 (LEN (CDR x))) 0))
(DEFUN BOOLEAN-LISTP (lst)
  (IF (ATOM lst)
    (EQ lst NIL)
    (AND (OR (EQ (CAR lst) T) (EQ (CAR lst) NIL)) (BOOLEAN-LISTP (CDR lst)))))
(DEFUN XOR (a b) (IF a (IF b NIL T) (IF b T NIL)))
(DEFUN XOR3 (a b c) (XOR a (XOR b c)))
(DEFUN V-XOR (a b) (IF (ATOM a) NIL (CONS (XOR (CAR a) (CAR b)) (V-XOR (CDR a) (CDR b)))))
(DEFUN B-CARRY (a b c) (IF a (OR b c) (AND b c)))
(DEFUN V-ADDER (c a b)
  (IF (ATOM a)
    (CONS c NIL)
    (CONS (XOR3 c (CAR a) (CAR b)) (V-ADDER (B-CARRY c (CAR a) (CAR b)) (CDR a) (CDR b)))))

```