Technical Report

UCAM-CL-TR-476
ISSN 1476-2986

Number 476

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Linking ACL2 and HOL

Mark Staples

November 1999

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Linking ACL2 and Hol

Mark Staples

November 2, 1999

Abstract

This report describes ACL2PII, a system which dynamically links the theorem provers ACL2 and Hol, using the PROSPER project's Plug-In Interface to Hol. The focus of the system is on making ACL2 theorems available from within Hol. In a motivating example we show how to transfer results from ACL2's 'small machine' theory. This theory highlights two of ACL2's strengths: symbolic simulation and the fast execution of operationally defined functions. This allows ACL2 specifications to be readily validated against real-world requirements. The ACL2PII system allows Hol users to capitalise on results about such ACL2 specifications. ACL2 and Hol are both general purpose theorem provers, but Hol is slightly more expressive, and has growing infrastructure for interoperability with other systems. This report assumes a passing knowledge of both ACL2 and Hol.

1 Introduction

ACL2 [8] and Hol [5] are two well-known mechanised theorem provers. This report describes ACL2PII: a dynamic link for translating theorems between two 'live' sessions of Hol and ACL2. We assume a passing knowledge of both ACL2 and Hol98[1]. The link uses the PROSPER project's Plug-In Interface to Hol [11], which allows Hol to interact with external systems. ACL2PII allows a user to run ACL2 from within a Hol session, and allows results from ACL2 to be interpreted in Hol.

ACL2 and Hol have different languages and different logics. ACL2 uses untyped s-expressions to represent first-order logic, whereas Hol uses typed terms for higher-order logic. ACL2PII is based on a scheme for translating ACL2 s-expressions to Hol terms. It is easy enough to create a type of s-expressions in Hol, so that all ACL2 s-expressions can be trivially translated to Hol. (Appendix A describes a Hol theory of s-expressions.) However, this kind of translation is not usually *useful.* For practical purposes, it is important to find a translation which allows ACL2 theories to be interpreted in the environment of the standard Hol theory libraries.

[1]Especially recent advances in Hol such as the oracle mechanism, and the automated function and type definition facilities provided by BossLib.

Thus, ACL2PII's translation scheme is *ad-hoc*—arbitrary translations can be given for different kinds of s-expressions. A set of default translations are available, so that appropriate s-expressions can be translated into Booleans, natural numbers, integers, simple arithmetic expressions, characters, strings, lists and tuples. New translation clauses may be added so that new ACL2 theories can be translated into Hol.

New translations are added in a variety of ways: by the use of ACL2PII commands to translate definitions from ACL2, by specification, or by hand-coding. Using translations to assert theorems in Hol may introduce inconsistencies into Hol, so the authors of translations must be careful to avoid this problem. The danger of this is reduced if users mainly rely on the automated transfer of definitions from ACl2. Section 2 has an example of this for ACL2's small-machine theory [2, 10]. In Section 3 we describe how to specify and hand-code translations. Section 4 lists the main commands for importing results into Hol. We discuss other work in Section 5 before presenting concluding remarks in Section 6. Appendix A presents a Hol theory of s-expressions, which can be used for default translations. Appendix B discusses system requirements and installation instructions for ACL2PII. Appendix C describes commands for interacting with an ACL2 session. Appendix D describes ACL2PII's architecture and code structure. Appendix E gives the source scripts for the small machine example presented in Section 2.

2 Example: Small Machine

Here we demonstrate how to make a link from ACL2 to Hol. We use as an example ACL2's 'small machine' theory [2, 10], which is an operational definition of a simple microcomputer. The theory showcases ACL2's facilities for symbolic simulation and the fast execution of concrete machines. We proceed as follows:

1. **identify** base ACL2 syntax required for our Hol investigation,
2. **define** base Hol constants for that syntax,
3. **specify** translations from ACL2 s-expressions to Hol terms, and
4. **translate** (automatically) ACL2 definitions and theorems to Hol.

We expand upon these steps below. The source scripts for this example appear in Appendix E.

2.1 Identify Base ACL2 Syntax

The small machine theory defines many auxiliary functions and lemmas leading up to interesting high-level properties about small machines. Here we look at some basic functions on states.

The state of a small machine is a list `(PC STK MEM HALT CODE)` of elements representing: the program counter `PC`, a stack of return addresses `STK`, a memory of numbers `MEM`, a machine-halted flag `HALT`,

and a program part CODE which is a collection of named lists of instructions. States can be constructed with the STATE function, e.g. (STATE PC STK MEM HALT CODE).

The small machine has eight instructions, including (MOVE A B) which moves the contents of address B to address A, (MOVI A B), which moves B to address A, (CALL A), which calls the first instruction of the sub-routine named A, and (RET) which returns from a sub-routine or halts the machine if the stack is empty.

2.2 Define Base Hol Constants

In Hol, we mirror the syntax above. First, we define a type for the syntax of instructions:[2]

```
Hol_datatype `instr = MOVE of num => num
                    | MOVI of num => int
                    ...
                    | CALL of string
                    | RET`
```

We assume that in Hol, memory values will be represented as integers, addresses within a program as natural numbers, and program names as strings. States are represented as a tuple:

```
val state = ty_antiq (Type
        `:(string # num) #                (* PC   *)
          (string # num) list #           (* STK  *)
          int list #                      (* MEM  *)
          bool #                          (* HALT *)
          (string # instr list) list`)    (* CODE *)
```

2.3 Translation

We must say how ACL2 s-expressions are interpreted in Hol. There are a collection of default translations provided in the base ACL2PII system, so we need only give cases for the new syntax introduced above.

Translations can be specified using patterns. A pattern is a triple of a string (representing an ACL2 s-expression), a corresponding Hol term quotation, and a list of Hol term quotations representing side-conditions on the translation.[3] The Hol term quotation contains free variables whose names are used to do pattern matching of the ACL2 pattern with actual ACL2 expressions.

Here are four translation specifications for small-machine instructions:

```
("(MOVE A B)" , `MOVE A B` , [])
("(MOVI A B)" , `MOVI A B` , [])
```

[2]Except where noted, throughout this example all teletype font indicates information which must be manually entered by a user of ACL2PII.

[3]No new side-conditions are required for the small machine theory example here, and so all side-condition lists are empty.

```
("(CALL A)"    , 'CALL A',   , [])
("(RET)"       , 'RET'       , [])
```

Finally we give a translation for constructing states as follows:

```
("(STATE P SO M H C)" , '(P,SO,M,H,C):^state' , [])
```

2.4 Transfer Definitions and Theorems

Having established the initial infrastructure above, we can bootstrap ourselves into a richer Hol theory of small machines, and transfer concrete executions and abstract theorems about small machines to Hol.

2.4.1 Base Constants

The evaluation of small machines is given by an operational definition for `(SM S N)` which runs the machine for `N` clock cycles, starting from an initial state `S`. The function `SM` is defined in terms of another function `STEP`. However, say that we are not interested in *how* `SM` is defined, but only in some of its properties. To take a two-argument function symbol `SM` as a typed given constant in Hol, we say:

```
mkbasefun "SM" 2
          "SM" (Type ':^state_ty -> num -> ^state_ty')
```

This introduces an undefined Hol constant `SM` of the type above, and sets up a translation from s-expressions `(SM S N)` to terms `SM S N`. The Hol constant `SM` will only acquire properties by the assertion of theorems translated from ACL2.

2.4.2 Definitions

The small machine theory defines an abbreviation for a multiplication program. The program is reproduced here (comments start with a semi-colon and continue to the end of the line):

```
(DEFUN TIMES-PROGRAM NIL
;          instruction    pc      comment
  '(TIMES (MOVI 2 0)     ; 0   mem[2] <- 0
          (JUMPZ 0 5)    ; 1   if mem[0]=0, go to 5
          (ADD 2 1)      ; 2   mem[2] <- mem[1] + mem[2]
          (SUBI 0 1)     ; 3   mem[0] <- mem[0] - 1
          (JUMP 1)       ; 4   go to 1
          (RET)))        ; 5   return to caller
```

We can transfer this to Hol by using the following ACL2PII command:

```
mkfun "TIMES-PROGRAM"
      "TIMES_PROGRAM" (Type ':^string # (instr list)')
```

This translates the statement of the definition in ACL2, and declares a new Hol constant with the translated definition as follows:

```
TIMES_PROGRAM =
  ("TIMES", [ MOVI 2 0i; JUMPZ 0i 5; ADD 2 1;
              SUBI 0 1i; JUMP 1; RET ])
```

A translation is added, which turns the s-expression (TIMES-PROGRAM) into the constant term TIMES_PROGRAM.

2.4.3 Theorems

The small machine theory proves a theorem called SM-+ reproduced here as follows:

```
(DEFTHM SM-+
  (IMPLIES (AND (NATP I) (NATP J))
           (EQUAL (SM S (+ I J))
                  (SM (SM S I) J))))
```

We can transfer this to Hol by calling getthm [] "SM-+", which returns the Hol theorem: SM S (I + J) = SM (SM S I) J. The 'typing' conditions in the ACL2 theorem are satisfied by the types of the terms I and J, and so have been simplified away.

The first argument to getthm is a list of type-guessing functions (see Section 3.4 below). For example, the small machine theory contains a theorem called j+j reproduced here as follows:

```
(DEFTHM J+J (EQUAL (+ J J) (* 2 J)))
```

In Hol, this could apply to either natural numbers or integers. We can import the natural number version by calling:

```
getthm [num_const_tg] "J+J"
```

which returns the Hol theorem J + J = 2 * J, and we can import the integer version by calling:

```
getthm [int_const_tg] "J+J"
```

which returns the Hol theorem J +_ J = 2i *_ J.

2.4.4 Executions

We have separately implemented an ML function mksexp_state which creates ML representations of ACL2 s-expressions for small machine states. For example we can create an initial state as follows:

```
val s = mksexp_state ("MAIN",0) [] [0,0,0,0,0] false
          [("TIMES",["(MOVI 2 0)",
                     "(JUMPZ 0 5)",
                     "(ADD 2 1)",
                     "(SUBI 0 1)",
                     "(JUMP 1)",
                     "(RET)"]),
           ("MAIN", ["(MOVI 0 10000)",
                     "(MOVI 1 1000)",
                     "(CALL TIMES)",
                     "(RET)"]) ];
```

In this program, the MAIN subroutine initialises two locations, and then calls the TIMES subroutine to multiply those two values. We have also separately implemented an ML function sm_conv—calling sm_conv [] s 40007 executes the state s for 40007 clock cycles within ACL2, and then returns the corresponding Hol theorem, as follows (the program text is elided here):

```
SM (("MAIN",0),[],[0i;0i;0i;0i;0i];F;[...]) 40007 =
   (("MAIN",3),[],[0i;1000i;10000000i;0i;0i];T;[...])
```

3 Translations Explained

This section describes the ACL2PII's scheme for the ad-hoc translation of ACL2's s-expressions to Hol terms. Simple translations can be specified in a declarative manner, and more complex translations can be hand-coded. Side-conditions can also be attached to terms during translation.

For some s-expressions there are many possible translations. For example, the s-expression NIL is used in ACL2 where in Hol you might either use the term F (falsity, for the Boolean type), or [] (the empty list, for list types), or NONE (the empty option, for option types). ACL2PII's translation scheme is type directed, so that each of these alternatives can be generated, depending on the required type. For some translations, the type of sub-components is not unambiguously determined. The translation scheme provides some support for guessing the type of sub-components.

3.1 Translation Specifications

ACL2PII provides a simple method for specifying new translations, based on matching s-expressions. Some example translation specifications have already been seen in Section 2. A translation specification has an s-expression source pattern, a corresponding target Hol term, and a list of any translation side-conditions (see Section 3.3 below). Consider the following translation:

```
("(IMPLIES X Y)"     , `X ==> Y`, [])
```

In the Hol term `X ==> Y`, the infix implication symbol is a constant, and X and Y are the only variable names. These names are used as variables for matching in the s-expression (IMPLIES X Y). That is, the s-expressions matching X and Y will be (recursively) translated into Hol terms, and then substituted for X and Y in the substitution pattern `X ==> Y`. This substitution pattern has type :bool, and so the entire translation will only be applied when a Boolean term is required. The types of X and Y are also both :bool, and so in the translation of matching s-expressions, only translations producing Booleans will be used. The empty list of terms in the translation specification indicates that this translation has no side-conditions.

The variable names in the Hol term and the ACL2 s-expression should all be upper-case, as corresponding ACL2 symbol names are

parsed as upper-case. Currently, the Hol term must be monomorphic—the interpretation of translation specifications does not yet handle polymorphically typed terms. Section 3.2 below briefly describes how to hand-code translations for situations when a translation can not be specified with a simple pattern and monomorphic terms.

3.2 Hand-Coded Translations

If a translation can not be specified using a simple s-expression pattern and a monomorphically typed Hol term, then it is possible to hand-code new translations. The type of a translation is:

```
(typeguess * term list * posttype * sexp,
 term list * term) trans
```

The abstract type `(a,b)trans` supports dynamically extensible recursive functions from `a` to `b`. The `typeguess` type is described in Section 3.4. The `posttype` type is a datatype isomorphic to Hol types, and indicates the required type for translating the `sexp` value. The term list on the left is the currently accrued logical context, and the term list on the right is the collection of side-conditions.

3.3 Side Conditions

The translation of ACL2 s-expressions to Hol terms underlies the translation of the statements of theorems in ACL2 which are then asserted as true in Hol. Section 4 describes the functions `getthm`, `getexec` and `getfun` which all use Hol's oracle mechanism to assert theorems. Translations are extra-logical, and so the normal guarantees of soundness provided by LCF theorem provers [4] do not apply. The authors of translations must take care not to allow inconsistencies to arise in Hol as a result of the translation process.

This danger can be mitigated if s-expressions are translated into otherwise uninterpreted constants in Hol. For example, in Section 2, the constant `SM` does not have any definition in Hol, and so any theorems imported from ACL2 about the constant need only be mutually consistent.

ACL2PII's translation scheme also provides for *side-conditions* on translations. These are accumulated during the translation process, are included as hypotheses on the theorem's conclusion before the theorem is asserted in Hol. For example, consider a translation from an ACL2 s-expression for subtraction `(- X Y)` to a Hol term of subtraction over the natural numbers `X - Y`. If `X` is less than `Y`, then the ACL2 s-expression will be a negative number, but the Hol term will be zero. This semantic discrepancy can be avoided if we add a side-condition `Y <= X` to the translation, which appears:

```
("(- X Y)", `X - Y`, [ `Y <= X` ])
```

Another source of problems is the difference in the level of specification between functions in ACL2 and Hol. For example consider taking the tail of an empty list. In ACL2, the following is a theorem:

```
(EQUAL (CDR NIL) NIL)
```

However in Hol, the term `TL []` is under-specified. That is, in Hol it is impossible to prove any useful facts about the tail of an empty list. Translating the ACL2 theorem to Hol would not introduce an inconsistency into Hol, but it would introduce a new, unusual fact. For the translation of s-expressions like `(CAR L)` into Hol terms `TL L`, we add the side-condition `L ~= []`.

The *guards* on ACL2 functions are not translated to Hol. We can rely upon guards being true of any s-expression we recover from ACL2. However, as we only translate theorems from ACL2 and not to it, we do not need to establish the translations of these guards in Hol.

3.4 Type Guessing

For many translations, the types of sub-components can be determined from the required type. However, sometimes it is necessary to 'guess'[4] the type of a sub-component. For example, the s-expression `(EQUAL X Y)` will translate to a Hol Boolean term, but we must guess the types of `X` and `Y`.

There are various convenient ways to generate type-guessing functions. The ML function `num_const_tg` recognises numerals to be natural numbers, and `int_const_tg` recognises numerals to be integers. The function `name_type_tg` takes a list of string/type pairs, and for s-expressions with one of those names, guesses the corresponding type. These functions are listed in Appendix C.

Like translation, type guessing must be extensible, so that new type-guessing heuristics can be added for new ACL2 theories. The current scheme has an abstract type of type-guessing functions `typeguess`. There are a collection of basic type-guessing functions provided in the system, and these can be extended by using functions described in Appendix C. This abstract type is implemented by the type:

```
(sexp,hol_type)trans
```

4 Importing into Hol

The translations discussed in Section 3 allow s-expressions to be translated to terms. Now we discuss how ACL2 theorems, executions, and definitions are translated from ACL2 to Hol. These commands are documented in Appendix C.

In the description of the commands below, we sometimes say that ACL2 removes macros. In ACL2, the s-expressions visible to users will be constructed from either defined functions, or from macros which expand into defined functions. Macro expansion is arbitrarily complicated, and so we let ACL2 expand all macros before translating s-expressions. This improves our confidence in the correspondence with

[4] As the translation scheme is *ad-hoc*, we prefer to call this process *type guessing*, rather than *type inference*.

ACL2, and also reduces the amount of work required to establish a set of translations from ACL2.

4.1 Declaring Base Constants for Translation

It may not be necessary to import an entire ACL2 theory into Hol, but instead just import definitions above some middle level of abstraction. The function `mkbasefun` declares a Hol constant to correspond to an ACL2 function. It introduces translation and type-guessing functions for the new constant.

4.2 Importing Definitions

There are two commands for importing ACL2 definitions into Hol: `mkfun` and `getfun`. Both take three arguments: a `string` which is the name of the desired ACL2 function, a `string` which will be the name of the corresponding Hol constant, and a `hol_type` which will be the type of the Hol constant. The two functions are similar, in that they both declare a constant in Hol, but `mkfun` introduces a definition using Hol's fundamental type definition principle (via Hol's TFL library [12]), whereas `getfun` merely asserts an equality 'definition' theorem using Hol's oracle mechanism.

As with the translation of theorems in Section 4.3, the statement of the named definition is first recovered from ACL2, parsed into an s-expression, macros are removed, and finally it is translated. The translation will result in an equality term, and a list of side-conditions. Hol does not support conditional definitions, and so the list of side-conditions should be empty (or, equivalent to true after simplification). For `mkfun`, the translated term is used to define a constant, and for `getfun`, the translated term is returned as a theorem. Both functions automatically add translations from the named ACL2 function to the new Hol constant, and also add appropriate type-guessing functions.

Arguments of ACL2 functions can be translated to curried positions or as tuple positions, depending on the required Hol type. For example, ACL2's small-machine theory declares a function `CPLUS`, which adds two natural numbers. There are two alternative ways to import this definition to Hol using `mkfun`:

```
mkfun "CPLUS" "CPLUS" (Type ':num -> num -> num')
mkfun "CPLUS" "CPLUS" (Type ':num # num -> num')
```

Either is acceptable, depending on the user's preference. The resulting translation and type-guessing functions will both work as expected.

4.3 Importing Theorems

Theorems in ACL2 can be imported into Hol by using the `getthm` command. It recovers the statement of a named theorem from ACL2, parses that into an s-expression, lets ACL2 remove macros, and then translates that s-expression. The translation results in a term and a list of side-condition terms, which are together asserted as a theorem

using Hol's oracle mechanism. The oracle tag indicates that ACL2PII was used, and also contains the name of the source theorem.

The command's first argument is a `typeguess list` argument mainly intended for guessing the types of numerals and free variables. The command's second argument is a `string` which is the name of the theorem in ACL2.

4.4 Importing Executions

If a concrete instance of an executable function is given at the ACL2 command line, it will be executed. The command `getexec` takes an ML s-expression, executes it in ACL2, parses the result, puts both sides as arguments in an equality, lets ACL2 remove macros, translates that s-expression, and asserts the resulting term using Hol's oracle mechanism. The command takes as its first argument a list of type-guessing functions, which are used during the translation. This command underlies the implementation of `sm_conv` mentioned earlier in Section 2.

5 Other and Future Work

Theorem provers can be connected to enable translations of many kinds: between commands in user-level *proof scripts*, between *theorems*, or between *proof objects* generated during the proof of those theorems. A link between theorem provers could act either *statically* on source files for theory definitions and proofs, or *dynamically* by translating information between 'live' sessions of the theorem provers. The ACL2PII system is a dynamic link which translates both theorems and elements of theory definitions.

The ACL2 source text for an ACL2 theory is a sequence of s-expressions. It may be possible to adapt ACL2PII's transfer mechanisms to implement the static translation of ACL2 source files to Hol source files. Hol's automatic proof facilities are not as powerful as those in ACL2, so the static translation of proof scripts may not be completely automatic, but may nonetheless serve as a useful starting point.

The translation of proof objects has been demonstrated by the Hol/CLAM system [1], which translates proof plans from CLAM to proof tactics which are re-run independently in Hol. It may be possible to similarly construct Hol tactics from information extracted from ACL2 'proof-trees'. The ACL2PII system would serve as a useful base from which to investigate this idea.

As part of the PROSPER project, many systems are being linked to Hol through the Plug-In Interface. This interface allows Hol to be used in a client or server role with respect to other tools. The ACL2PII system is an example of a tool which uses Hol as the client for a server ACL2 session. Other formal reasoning systems have been connected as servers to Hol. For example, there are Plug-Ins for MONA [9] (a system for deciding propositions in a finitary second-order logic), and Gandalf [7] (a tableaux system for first-order logic). However these

systems operate on a fixed logic and hence their Plug-Ins do not need to be extensible. ACL2 and Hol both support theory definition, and so ACL2 is in this respect more similar to the Nuprl-Hol connection [6] and the Hol/CLAM system, which work with arbitrary theory descriptions.

The ACL2PII system is entirely focussed on using Hol as a client. The PROSPER Plug-In framework also allows Hol to be used as a server, and so it may be possible to feed results from Hol theories to ACL2. Aspects of this translation would be simpler than in ACL2PII. For example, the types of Hol terms could be dropped, or turned into guards or side-conditions on variables. However, translating Hol to ACL2 would present problems of its own: some higher-order functions may have no translation in ACL2's first-order logic, and elements of Hol's expressive logic may not have simple counterparts in ACL2. For example, real numbers are a type in Hol, but are not numbers in ACL2 [3].

6 Conclusion

There are various reasons for linking ACL2 to Hol. Firstly, they are not identical, so a link can to some extent allow each to trade on the other's strengths in automatic proof or logical expressiveness. Secondly, a link may facilitate the sharing of application theories developed in the two communities. Finally, a system for linking theorem provers may help to cross-fertilise ideas within the theorem proving community.

The ACL2PII system enables Hol users to quickly import definitions and theorems from ACL2. ACL2PII provides default translations for strings, numbers, lists, and other basic datatypes, but this functionality can be incrementally extended, so that new theories developed in ACL2 can also be imported into Hol. Translation in the system is type-directed, so as to distinguish between 'morally' distinct uses of ACL2 symbols. The types of sub expressions can often be structurally determined, but for cases where that is not possible, ACL2PII provides a framework for guessing the type of s-expressions.

The translation of ACL2 s-expressions to Hol terms is used to assert theorems in Hol, and so translations require some extra-logical commitment to the logical soundness of their results. This commitment can be minimised by only translating to otherwise uninterpreted Hol constants, or by adding logical side-conditions to translations.

Communication between ACL2 and Hol is established by the PROSPER project's Plug-In Interface. The support provided by the PII has facilitated the rapid incremental development of ACL2PII.

Acknowledgements

Mike Gordon, J Strother Moore, Michael Norrish, Jun Saweda, Susanto Kong Wei and Konrad Slind have all greatly helped the author either in

the preparation of this document, or in the construction and debugging of ACL2PII.

References

[1] Richard Boulton, Konrad Slind, Alan Bundy, and Mike Gordon. An interface between Clam and HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98*, number 1479 in Lecture Notes in Computer Science, pages 87–104, Canberra, 1998. Springer Verlag.

[2] Bob Boyer and J Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos.* MIT Press, 1996.

[3] Ruben Gamboa. Square roots in ACL2: A study in sonata form. Technical Report TR96-34, UTCS, November 1996.

[4] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS.* Springer-Verlag, 1979.

[5] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[6] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of TPHOLs'96: The 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS.* Springer-Verlag, 1996.

[7] Joe Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, Ch. Paulin-Mohring, and L. Thèry, editors, *Proceedings of TPHOLs'99: The 12th International Conference on Theorem Proving in Higher Order Logics*, LNCS. Springer-Verlag, 1999.

[8] Matt Kaufmann and J Moore. A precise description of the ACL2 logic. A working draft of a precise description of the base logic., April 1998.

[9] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual.* BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998.

[10] J Moore. Symbolic simulation: An ACL2 approach. In G. Gopalakrishnan and P. Windley, editors, *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 334–350. Springer-Verlag, November 1998.

[11] PROSPER home page. `http://www.dcs.gla.ac.uk/prosper/`.

[12] K. Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of TPHOLs'96: The 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*. Springer-Verlag, 1996.

A Hol S-expressions

Hol data-type of ACL2 s-expressions

```
val packagename = Type ':string';
val name        = Type ':string';

Hol_datatype
'sexp = ACL2_SYMBOL    of ^packagename => ^name
      | ACL2_STRING    of string
      | ACL2_CHARACTER of ascii
      | ACL2_NUMBER    of num
      | ACL2_PAIR      of sexp => sexp';
```

ACL2 numbers include integers, rationals, and complex rationals. Currently ACL2 numbers are modelled as Hol natural numbers, but we will use integers or rationals as support for them improves in hol98.

B Installing ACL2PII

ACL2PII is available from:

`http://www.cl.cam.ac.uk/users/ms204/acl2pii.tar.gz`

ACL2PII requires appropriate versions of:

1. MoscowML
2. hol98
3. PROSPER PII
4. ACL2

Complete installation instructions, including required version numbers for the above systems, are detailed in the file `README`.

C ACL2PII Command Reference

The `ACL2PluginBoss` module contains functions which establish a link with an ACL2 session, and which shadow many of the commands found in ACL2. Below, we detail commands to:

1. establish a link,
2. run ACL2 commands over the link,
3. manage ACL2PII local context, and

4. transfer over the link.

Throughout this appendix we refer to a ML data-type of ACL2 s-expressions. It is defined in `ACL2Data` as follows:

```
datatype number = rational of rat | complex of rat * rat
datatype sexp = character of char
              | string    of string
              | number    of number
              | symbol    of packagename * name
              | pair      of sexp * sexp
```

C.1 Establishing a Link

To establish a link, the following must happen:

1. the server (ACL2, running inside `pii_harness`) must be running, and
2. the client (ACL2PII, running in a Hol session) must be running.

The server can run either locally and communicate over unix sockets, or remotely and run over internet sockets.

`initialise_auto_acl2 : string -> unit`
: This sets up a local server with a pipe specified by the string argument, and then sets an ACL2PII client going.

`initialise_remote_acl2 : string -> int -> unit`
: This assumes that a remote server is running on a machine with address given by the string argument, and that the server is listening on a port specified by the integer argument. It sets the ACL2PII client going.

`initialise_local_acl2 : string -> unit`
: This assumes that a local server is running and listening on a pipe specified by the string argument. It sets the ACL2PII client going.

C.2 Commands Over the Link

These functions allow commands to be executed on the ACL2 server.

`command : string -> string`
: This is the basic control function, which executes the string argument as if it was typed at the ACL2 command line. The user must be careful not to execute any command which changes the prompt.

`in_package : string -> unit`
: This changes the current package to a package specified by the string argument.

`include_book : string -> unit`
: This reads in an ACL2 book of definitions and theorems from a file whose name is given by the string argument.

`undo : unit -> string`

This undoes the last ACL2 event. However, note that some ACL2PII commands are composed of many ACL2 events.

`print_event : string -> string`

`pe : string -> unit`

These functions return or print information about named events in ACL2's event history.

C.3 Local Link Context

`current_package : unit -> string`

This returns ACL2PII's idea of the current package in the ACL2 server.

```
current_imports :
    unit -> (string * (string * string) list) list
```

This returns ACL2PII's idea of the current imports in the ACL2 server.

`current_packages : unit -> string list`

This returns ACL2PII's idea of the currently defined packages in the ACL2 server.

`update_imports : unit -> unit`

This re-establishes ACL2PII's idea of ACL2's current imports and defined packages.

`is_variable : (string * string) -> bool`

This determines if a package name and symbol name constitute a symbol which is an ACL2 *formal variable* [8].

`normalise : sexp -> sexp`

This normalise an s-expression w.r.t. the current context.

`parse : string -> sexp`

This parses a string as a normalised s-expression under the current context.

`pp_sexp : sexp -> string`

This pretty-prints an s-expression as string which could be re-parsed as a s-expression under the current context. (i.e. the package names of symbols are not printed if they are in or imported in the current package.)

C.4 Transferring Over the Link

These commands are used for extracting information from the ACL2 server, and interpreting it within Hol.

`getthm : typeguess list -> string -> thm`

This imports a named ACL2 theorem into Hol. This is described further in Section 4.3.

`getexec : typeguess list -> sexp -> thm`

This executes the s-expression in ACL2, and returns as a theorem the translation of an equality between the given s-expression and the result. This is described further in Section 4.4.

`mkfun : string -> string -> hol_type -> thm`

`getfun : string -> string -> hol_type -> thm`

This declare new constants in Hol, and automatically add translations and type-guessing. The first string is the ACL2 name, and the second is the name of the corresponding Hol constant. This is described further in Section 4.2.

`mkbasefun : string -> int -> string -> hol_type -> unit`

Take a symbol of arity given by the `int` argument and name given by the first `string` argument as a base translation to a corresponding Hol constant whose name is given by the second `string` argument and whose type is the `hol_type` argument.

```
translate :
    typeguess list -> hol_type -> sexp ->
      (term list * term)
```

This translates an s-expression to a term of the given type. The `typeguess list` are auxiliary type guessing functions used for the translation. The resulting `term list` is a list of side-conditions for the translation.

```
add_translations :
    (string * term_quote * term_quote list) list -> unit
```

This adds a list of translation specifications to the current collection of translations.

`add_typeguess : typeguess -> unit`

This adds a new type guessing function to the current collection of type guessing functions.

`num_const_tg : typeguess`

`int_const_tg : typeguess`

These type guessing functions recognise ACL2 numbers as Hol numbers and Hol integers (respectively).

`name_type_tg : (string * hol_type) list -> typeguess`

This gives a type guessing function to recognise s-expressions which are function calls, where the function symbol name is given by a `string`. For functions with more than zero arguments, the given `hol_type` is a function type from the argument types to the result type.

`var_tg : term list -> typeguess`

This adds a type guessing function so that for each variable in the term list, a symbol with the variable's name is recognised as being of that variable's type.

D ACL2PII Maintenance Reference

We first give an overview of the architecture for ACL2PII, and then an introduction to the code structure.

D.1 Behind the Scenes

ACL2PII works by interacting with ACL2 just as a normal human user would. Command strings are send to an ACL2 session, and output strings are read up to ACL2's next prompt. Figure 1 provides an architectural overview of how users interact with Hol and ACL2PII, and how those systems connect with the PROSPER PII and ACL2.

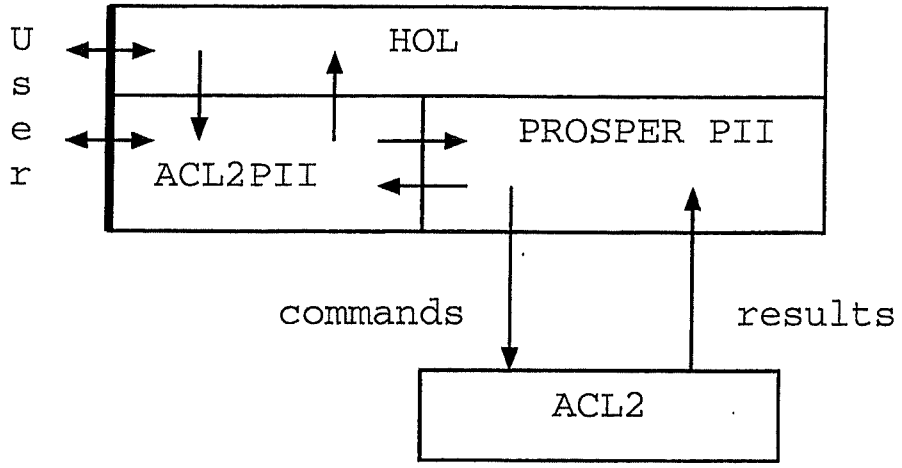

Figure 1: How ACL2 Talks with Hol

The results from ACL2 may be strings representing s-expressions from normal execution, or they may be strings representing error conditions in abnormal circumstances. ACL2PII attempts to catch these error strings and reports them as exceptions in ML.

If the return string represents a normal behaviour, then ACL2PII will often attempt to parse the string as an s-expression. Parsing ACL2 s-expressions should mostly be a trivial matter, but in an ACL2 interactive session, the package names of symbols in (or imported into) the current package are not printed. ACL2PII keeps a local copy of this part of ACL2's logical context to enable parsed s-expressions to be properly normalised.

D.2 Code Structure

Figure 2 shows dependencies for the source code files of the ACL2PII system. It depicts the following modules (and also the `PlugIn` module from the PROSPER PlugIn system):

ACL2PlugInBoss presents an interface to single ACL2 session. It keeps track of references about the session, as well as managing local logical context, translations and type-guessing functions. This provides simplified versions of command functions from `ACL2PlugIn`.

ACL2PlugIn describes an abstract type `acl2sn` for interacting with (possibly many) ACL2 sessions. This structure provides basic functions for executing commands in ACL2 sessions and interpreting their results. It also provides many specialised functions mimicking ACL2 commands.

ACL2SexpTerm defines abstract types for creating and extending translation and type-guessing functions. It contains the function `new_sexp_term` for interpreting translation specifications.

ACL2Parse is a parser for s-expressions. It exports two main functions: `parse` for parsing s-expressions in files, and `parse_str` for parsing strings. The parser is generated in the standard Moscow ML way by using `mosmllex` and `mosmlyac` on the grammar and lexer specifications in `acl2yacc.grm` and `acl2lex.lex`.

ACL2Data defines the ML datatype for s-expressions, as well as derived constructors and destructors, pretty-printing, matching, and normalisation for s-expressions.

ACL2SexpTheory is a Hol theory of s-expressions. The source file `ACL2SexpTheory.sml` is generated from `ACL2SexpScript.sml` in the normal way by using `Holmake`.

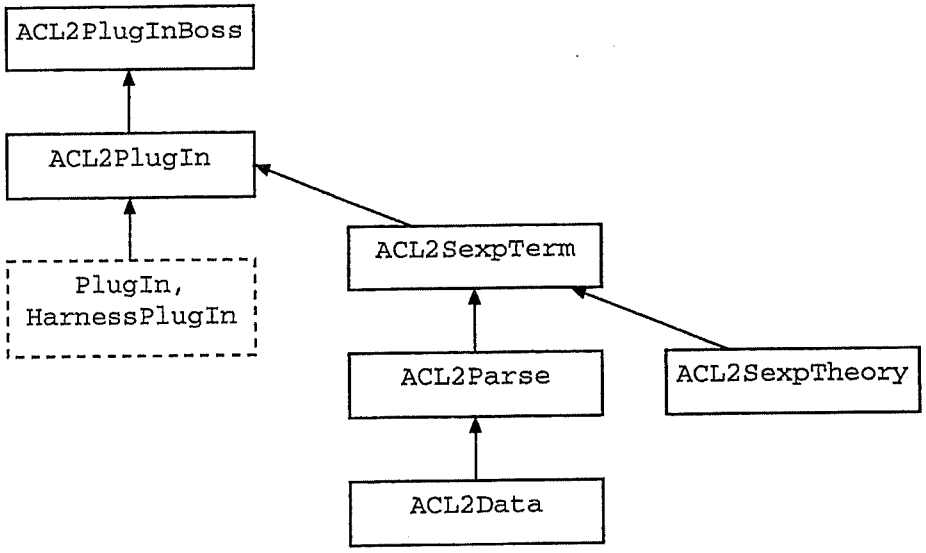

Figure 2: ACL2PII's Architecture

E ACL2PII Small Machine Example

This appendix contains the complete source files for the example link described in Section 2.

E.1 Hol SM Theory Script

```
(* File: SMScript.sml
   Author: Mark Staples
   Last Modified: 6 July 1999
```

```
   Copyright: University of Cambridge 1999
*)

local open integerTheory stringTheory in end;

open HolKernel bossLib Parse;

val _ = new_theory "SM";

val _ = Hol_datatype
 `instr = MOVE of num => num
        | MOVI of num => int
        | ADD of num => int
        | SUBI of num => int
        | JUMPZ of num => num
        | JUMP of num
        | CALL of string
        | RET`;


(* Locations, Programs, and State types *)

val loc = Type `:string # num`;

val prog_ty = Type `:(string # instr list) list`;
val state_ty = Type `:^loc # ^loc list # int list # bool # ^prog_ty`;

val prog = ty_antiq prog_ty;
val state = ty_antiq state_ty;

val _ = adjoin_to_theory {
  sig_ps = SOME ((C Portable_PrettyPrint.add_string)
        "val state_ty : Type.hol_type;\n"),
  struct_ps = SOME ((C Portable_PrettyPrint.add_string)
        "val state_ty = Parse.Type \
\ `:(string # num) # (string # num) list # \
\   int list # bool # (string # instr list) list`;\n")};

val _ = adjoin_to_theory {
  sig_ps = SOME ((C Portable_PrettyPrint.add_string)
        "val state : Term.term;\n"),
  struct_ps = SOME ((C Portable_PrettyPrint.add_string)
        "val state = Term.ty_antiq state_ty;\n")};

val _ = export_theory ();
```

E.2 ACL2PII SM Library Source

```
open ACL2Data;
open ACL2PlugInBoss;
open SMTheory;

val initSM = (fn () => (
 add_translations
  [ ("(MOVE A B)"  , `MOVE A B`  , []),
    ("(MOVI A B)"  , `MOVI A B`  , []),
    ("(ADD A B)"   , `ADD A B`   , []),
    ("(SUBI A B)"  , `SUBI A B`  , []),
    ("(JUMPZ A B)" , `JUMPZ A B` , []),
    ("(JUMP A)"    , `JUMP A`    , []),
    ("(CALL A)"    , `CALL A`    , []),
```

```
   ("(RET)"       , 'RET'       , [])
 ];
add_translations
 [ ("(STATE PC STK MEM HALT CODE)", '(PC,STK,MEM,HALT,CODE):^state', []) ];
add_typeguess (name_type_tg
 [ ("STATE", Parse.Type ':(string # num) -> (string # num)list -> int list
                   -> bool -> (string # instr list)list ->^state_ty') ])
 ));


(* Constructing sexps for machine states: *)

fun mksexp_state pc stk mem halt code =
 mksexp_list I
 [ mksexp_pair (mksexp_symbol "SMALL-MACHINE", mksexp_num) pc,
   mksexp_list
     (mksexp_pair (mksexp_symbol "SMALL-MACHINE", mksexp_num)) stk,
   mksexp_list mksexp_num mem,
   mksexp_bool halt,
   mksexp_list (mksexp_pair
        (mksexp_symbol "SMALL-MACHINE",
         mksexp_list ((ACL2Data.normalise_sexp ("SMALL-MACHINE",[])) o
                      (ACL2Parse.parse_str))))
      code
 ];


(* Recovering concrete executions from ACL2: *)

fun sm_conv tgl isexp n =
  let val nsexp = number(rational(n,1))
      val lhs = mksexp_call (("SMALL-MACHINE","SM"),
                             [sexp_quote isexp,nsexp])
  in
    getexec tgl lhs
  end;
```

E.3 ACL2PII SM Example Script

```
load "Process"; load "FileSys"; load "SMLib";
open SMLib;

(* If you try either of the two options below, you must first
   set pii_harness going with acl2.  This is done automatically
   when calling initialise_auto_acl2

initialise_local_acl2 "/tmp/acl2pii.soc";

initialise_remote_acl2 "smelt.cl.cam.ac.uk" 4000;

initialise_auto_acl2
  "/home/ms204/local-project/acl2-pii/mospii/prosper/pii/c/pii_harness \
\  /home/ms204/bin/acl2";
*)

val PIIHARNESS = "/homes/ms204/projects/acl2-pii/mospii/prosper/pii/c/pii_harness";
val ACL2 = ShouldBeInHol.which "acl2";

initialise_auto_acl2 (PIIHARNESS ^ " " ^ ACL2);

include_book ((FileSys.fullPath ".") ^ "/small-machine");
```

```
in_package "SMALL-MACHINE";


(* initSM sets up initial translations for those portions of the SM syntax
   which correspond with HOL functions defined in SMScript.sml.
 *)

initSM();


(* Now let's add some more translations for various SM "typing" functions.
   If you'd prefer, you could put these into initSM in SMLib.sml too...

   These simple translations don't define new HOL constants, but just
   translate into pre-defined HOL constants.
 *)

add_translations
[ ("(NATP X)", `(K T)(X:num)`, []),
  ("(STATEP X)", `(K T)(X:^state)`, []),
  ("(PCP X)", `(K T)(X:(string # num))`, []),
  ("(STKP X)", `(K T)(X:(string # num)list)`, []),
  ("(MEMP X)", `(K T)(X:(int list))`, []),
  ("(CODEP X)", `(K T)(X:((string # instr list)list))`, [])
];


(* Now we'll declare (using mkbasefun), or define (using mkfun or getfun)
   a bunch of new HOL constants, and their translations from ACL2
   s-expressions.
 *)

(* Here are the state accessors, with definitions translated from ACL2. *)

val PC_def =
  mkfun "PC"    "PC"    (Type `:^state_ty -> (string # num)`);

val STK_def =
  mkfun "STK"   "STK"   (Type `:^state_ty -> ((string # num)list)`);

val MEM_def =
  mkfun "MEM"   "MEM"   (Type `:^state_ty -> (int list)`);

val HALT_def =
  mkfun "HALT"  "HALT"  (Type `:^state_ty -> bool`);

val CODE_def =
  mkfun "CODE"  "CODE"  (Type `:^state_ty -> ((string # instr list)list)`);


(* Say that STEP is our main acl2-interpretation function: take is as basic. *)

mkbasefun "STEP" 1 "STEP" (Type `:^state_ty -> ^state_ty`);


(* Now define SM, in terms of STEP.  We use getfun instead of mkfun because
   of a bug in TFL which should be fixed in hol98 Taupo 1 release and later...
 *)

val SM_def = getfun "SM" "SM" (Type `:^state_ty -> num -> ^state_ty`);
```

```
(* Now we can translate a demonstration execution.  First construct an
   ML s-expression for the initial state:
 *)

val i = mksexp_state ("MAIN",0) [] [0,0,0,0,0] false
          [("TIMES", ["(MOVI 2 0)",
                      "(JUMPZ 0 5)",
                      "(ADD 2 1)",
                      "(SUBI 0 1)",
                      "(JUMP 1)",
                      "(RET)"]),
            ("MAIN", ["(MOVI 0 10000)",
                      "(MOVI 1 1000)",
                      "(CALL TIMES)",
                      "(RET)"]) ];

(* Now execute it, as below.  sm_conv is a simple wrapper around the
   ACL2PII function "getexec"
 *)

val ir = sm_conv [] i 40007;


(* Now let's translate a slew of other functions from ACL2.

   Note the difference between the types of CPLUS and CTIMES.  That
   demonstrates that you can translate ACL2 functions to either
   curried or uncurried HOL functions.
*)

val CPLUS_def =
  mkfun "CPLUS"  "CPLUS"  (Type `:num # num -> num`);
val CTIMES_def =
  mkfun "CTIMES" "CTIMES" (Type `:num -> num -> num`);


val TIMES_CLOCK_def =
  mkfun "TIMES-CLOCK" "TIMES_CLOCK" (Type `:num -> num`);

val TIMES_PROGRAM_def =
  mkfun "TIMES-PROGRAM" "TIMES_PROGRAM" (Type `:string # instr list`);


val PI_def =
  mkfun "PI"    "PI"    (Type `:string # instr list`);

val ALPHA_def =
  mkfun "ALPHA" "ALPHA" state_ty;

val BETA_def =
  mkfun "BETA"  "BETA"  (Type `: int -> int -> int -> int -> int -> ^state_ty`);


(* Now let's retrieve an abstract theorem from the ACL2 SM theory: *)

getthm [] "SM-+";



(* Just because we've translated a theorem, that doesn't mean we can't
```

```
   define more HOL constants and translations from ACL2:
 *)

val DEMO_STATE_def = mkfun "DEMO-STATE" "DEMO_STATE" (Type `:^state_ty`);


(* See, we can use that new DEMO-STATE to translate a new theorem: *)

getthm [] "DEMO-THEOREM";



(* We can still add new translations if we want to translate auxilliary
   functions into standard HOL types... *)

add_translations
  [ ("(INTS . X)", `K T (X:int list)`, []) ];


(* Then transport more definitions, etc...*)


val INCPC = mkfun "PC+1" "INCPC" (Type `:string # num -> string # num`);

(*
 We can't mkfun GET, because:
        val GET_def = mkfun "GET" (Type `:num -> int list -> int`);
 has an error because the translation has side-conditions, namely:
        ~(MEM = [])
        ~(N = 0) ==> ~(MEM = [])
 Perhaps the first one shouldn't have arisen, but for the second, should
 we ditch it and leave GET underspecified in HOL?  We might get "surprising"
 theorems translated from ACL2, but probably not about any theorems we'de
 be interested in.
*)

val GET_def = getfun "GET" "GET" (Type `:num -> int list -> int`);

(* Similarly for FETCH: *)

val FETCH_def = getfun "FETCH" "FETCH"
        (Type `:(string # num) -> ((string # instr list)list) -> instr`);


val CURRENT_INS = mkfun "CURRENT-INSTRUCTION" "CURRENT_INS"
        (Type `:^state_ty -> instr`);

val PUT_def = getfun "PUT" "PUT" (Type `:num -> int -> int list -> int list`);
```