Well-Formedness Guarantees for ACL2 Metafunctions and Clause Processors

Matt Kaufmann and J Strother Moore
Department of Computer Science
University of Texas at Austin
email:{kaufmann,moore}@cs.utexas.edu

September 11, 2015

Abstract

Some runtime checks can be safely removed from code if appropriate program properties are proved. We describe how we have applied this idea to the ACL2 theorem prover to speed up the application of user-defined proof procedures. In particular, we discuss how and why we have added a new feature to ACL2 that allows the user to verify certain well-formedness properties of the expressions produced by user-defined proof procedures. Of special interest are the issues of extensibility (how we know that guarantees proved in one theory are adequate in another), formalization of the problem, and design decisions affecting the user interface.

Keywords: metatheory, derived rules, tactics, soundness, runtime testing

1 Introduction

ACL2 is a theorem proving system for an applicative subset of Common Lisp and is largely written in that subset. ACL2 is also a programming language that takes advantage of Common Lisp compilers so, by necessity, ACL2 adheres to the specification of Common Lisp[12]. ACL2 is used in industry to verify functional correctness and other properties of commercial hardware and some low-level software. Companies that have used ACL2 for such purposes include AMD, Centaur, IBM, Intel, Oracle, and Rockwell Collins. See [4] for some case studies dating from 2000. Today the most complete integration of ACL2 into an industrial workflow is at Centaur [11].

The ACL2 theorem prover allows the user to define new simplifiers (called *metafunctions*) and provers (called *clause processors*). These are just functions on Lisp s-expressions.

Every term in ACL2 is represented internally by a Lisp s-expression. Conveniently, each such s-expression, if preceded by a quote mark, is also a term, in particular a constant term whose evaluation is that s-expression. If α is a term, then `'`α is called its *quotation*.

By applying an evaluator to the quotation of a term, with the appropriate assignment of values to variable symbols, one obtains the value of the term. For example, `(+ x (+ y (- x)))`

is a term whose quotation is `'(+ X (+ Y (- X)))`, and `(eval '(+ X (+ Y (- X))) σ)` = `(+ x (+ y (- x)))`, if σ is the association list (assignment) that pairs the symbol `'X` with `x` and the symbol `'Y` with `y`.[1] Such a σ can be constructed with `(list (cons 'X x) (cons 'Y y))`.

In the simplest case, a metafunction f is just a Lisp function that takes an s-expression and produces an s-expression with an equivalent value under a certain evaluator, as suggested by the theorem:

ACL2 Metafunction Correctness:

```
(implies (and (pseudo-termp x)
              (alistp a))
         (equal (eval x a)
                (eval (f x) a)))
```

This looks very strong, since it says that f transforms every input to an equivalent one. But practically speaking, f will be the identity function on all inputs except those of the "expected" shape. When a metafunction f is proved correct, we require the user to specify the top-level function symbols of those terms which might be usefully simplified by f. Thus, the prover does not actually try f on every term it encounters (though that would be sound). See Section 4.

Assume the formula above has been proved for a user-defined function f. Then if α and β are terms and `(`f `'`α`)` = `'`β, then it is permitted to replace any occurrence of α by β, i.e., to use f as a simplifier. Of course, ACL2 does this only if α and β are different. To see why we can replace α by β, let σ be the list binding the variable symbols of α and β to their values. Then

α = `(eval '`α σ`)`
= `(eval (`f `'`α`)` σ`)`
= `(eval '`β σ`)`
= β.

In the ACL2 Metatheorem Correctness theorem above, `pseudo-termp` is a function that returns `t` (*true*) or `nil` (*false*) according to whether its argument is an s-expression that has the basic shape of a well-formed term. But `pseudo-termp` does not check that every symbol used as a variable is a legal variable symbol, that every symbol used as a function is a function symbol in the current theory, or that the number of arguments supplied is correct.

`Alistp` checks that its argument is a list of pairs "binding" symbols to values.

The name `eval` above is merely suggestive. ACL2 is first-order and does not permit the axiomatization of Lisp's universal interpreter. Instead, ACL2 can recognize a function as an *evaluator*, where (according to :DOC `defevaluator`[2]):

> an evaluator function for a fixed, finite set of function symbols is a restriction of the universal evaluator to terms composed of variables, constants, lambda expressions, and applications of the given functions. However, evaluator functions are constrained rather

[1] We lie a little bit for clarity's sake! `+` is not really an ACL2 function symbol; it is a macro that expands into a call of the function symbol `BINARY-+` of arity 2. We hope the reader familiar with ACL2 will overlook our misrepresentations. By the way, ACL2 is case insensitive for the function and variable symbols used in this paper; the symbol `BINARY-+` can also be written `binary-+`, `Binary-+`, etc. Ignore case if it is confusing.

[2] When we refer to ":DOC x" we mean see the documentation topic x in the ACL2 documentation, which may be found by visiting the ACL2 home page[5], clicking on The User's Manuals, then clicking on the ACL2+Books Manual and typing x into the "Jump to" box.

than defined, so that the proof that a given metafunction is correct vis-a-vis a particular evaluator function can be lifted (by functional instantiation) to a proof that it is correct for any larger evaluator function

The evaluator appropriate for proving the metafunction f correct is generally one that handles all and only the function symbols recognized and generated by that metafunction. For example, if f checks that its first argument is a list whose first element is the symbol `'+`, and for some such inputs f ultimately returns a list whose first element is the symbol `'FIX`, then the evaluator needed to prove f correct is probably one that includes `+` and `fix` in the finite subset of function symbols known to that evaluator. That evaluator, which we now denote by `ev` rather than Lisp's `eval`, would satisfy:

```
(ev (list '+ x y) a)
 = (+ (ev x a) (ev y a))
```

and

```
(ev (list 'FIX x) a)
 = (fix (ev x a)).
```

But in any case, if *ev* is any ACL2 evaluator and it is a theorem that

```
(implies (and (pseudo-termp x)
              (alistp a))
         (equal (ev x a)
                (ev (f x) a)))
```

then any time the ACL2 rewriter encounters an s-expression it can, if it chooses, replace that s-expression by the value of f on that s-expression, *provided the value represents a term.* The italicized proviso is the crux of this paper.

A clause processor is similar to a metafunction except it takes a clause (a list of terms) as input and produces a list of clauses as output. The meaning of a clause is the disjunction of the evaluations of its elements and the correctness of a clause processor says that the conjunction of the meanings of the output clauses implies the meaning of the input clause. Otherwise, the issues with using clause processors to simplify a goal clause to a set of stronger subgoals are analogous to the issues of using a metafunction and we do not discuss clause processors in detail further.

2 Back to the Future

Metafunctions were introduced to the Nqthm theorem prover [2], the predecessor of ACL2, in 1979 [1].[3] As that paper says, "There is nothing magic or 'meta' about this function," referring to a restricted evaluator and the modern connotation of "meta." "Meta" in Greek means "after," and in the received order of Aristotle's works, being, substance, cause, etc., are treated after physical matters. So in the most literal sense, evaluators and metafunctions are indeed "meta" because they must be defined and verified after all the function symbols they handle have been introduced.

In [1], the evaluator was called `meaning` and the correctness theorem for a metafunction was of the form

```
(implies (formp x)
         (and (equal (meaning x a)
                     (meaning (f x) a))
              (formp (f x))))
```

[3] Publication [1] is dated 1981, but is virtually identical to SRI Technical Report CSL-108 of the same name, dated December, 1979.

Note the conjunct in the conclusion requiring that f return a "`formp`". `Formp` recognized well-formed terms and was definable in Nqthm because it relied on an axiomatized function `arity` which "grew" as new function symbols were introduced.

The paper [1] presents a detailed proof of the validity of replacing α by β, along the lines sketched in the previous section. While that proof is in the context of Nqthm's term representation the basic argument is exactly the same. See also :DOC `meta`.

When we implemented ACL2, starting in 1989, we deleted the ACL2 analogue of the `formp` conjunct (and replaced the `formp` in the hypothesis by the weaker `pseudo-termp`) for at least two reasons. First, we chose not to introduce a function like `arity` that is extended with every new function symbol. Second, the requirements of Common Lisp forced a much more complicated definition of a well-formed term. See Appendix A. In 1989, the formal definition of well-formedness sufficient to allow proof of well-formedness just seemed daunting and likely to lead to proof obligations that were inordinately hard compared to the alternative: a runtime check on the well-formedness of the output of each instance of a metafunction.

This runtime check persisted, unnoticed, for a quarter of a century, despite significant extensions to metafunctions and the addition of the analogous handling of clause processors [3, 7]. Meanwhile, ACL2 was applied successfully to increasingly large and complex problems, with concomitant improvements in its algorithms and engineering. In 2015 we noticed that the runtime well-formedness check was costing a lot because of the size of the formulas ACL2 is now dealing with. See the discussion of the decompilation of machine code for the DES algorithm in Section 5 for an example.[4]

Thus, between the release of ACL2 Versions 7.1 and 7.2 (unreleased as of this writing, but available from GitHub), we decided to allow the runtime well-formedness check to be "proved away," i.e., to be made unnecessary by the proof of a well-formedness condition on the metafunction, partially undoing the earlier decision and making ACL2 more like Nqthm in this regard.

The reliance on a proof to replace a runtime check highlights the importance of the consistency of the theory. ACL2 discourages any "logically dangerous" act that can render the theory inconsistent. For example, new axioms about existing function symbols are discouraged; instead, ACL2 provides a conservative definitional principle and a widely used mechanism for extending the theory with constrained functions whose constraints are proved to be satisfiable. But, in the interests of allowing the user some flexibility in developing proof scripts, ACL2 permits the addition of axioms, redefinition of functions, skipping of proof obligations, and other logically dangerous acts – provided the user has signaled an understanding that the consequences are now his or hers. One might think the consequences are just that invalid formulas might be proved. But that is wrong! Far worse things can happen, as is easy to see in connection with the elimination of the well-formedness check. If well-formedness is proved in an inconsistent theory, then a metafunction might introduce a "term" that violates the fundamental invariants exploited by the theorem prover. Consequently, the Common Lisp execution engine could crash, overwrite code,

[4] In fact, decompilation is the motivating application in this work but we expect this work primarily to benefit industrial users of ACL2's metafunctions after it has been distributed.

delete files, etc. This issue is discussed in the 1979 version of [1] on page 67:

> Finally, we should observe that we could have stated [the metatheorem] so that [(FORMP (*f* x))] did not have to be proved in [theory] T1 and then could have implemented a run-time check that [the output] indeed represents a term. We then could have permitted T1 to be inconsistent without catastrophic consequences. We did not adopt this approach because in most cases the proof of the FORMP part of [the metatheorem] is straightforward and buys efficiency at the mere expense of complicating this paper.

The bracketed material was added by the current authors to make the paragraph make sense with the current notation.

It should be noted that such consequences of inconsistency already pervade ACL2: if the guards (preconditions) of a function are proved in an inconsistent theory, then ACL2's trust in the fidelity of the Common Lisp execution engine for ground evaluation is misplaced. See :DOC guard.

3 Well-Formedness Guarantees

A data structure x represents a well-formed term with respect to an ACL2 "logical world" w iff (termp x w) is true. Termp and its important subfunctions are defined in Appendix A. Termp is mutually recursive with term-listp which checks that its argument is a list of terms with respect to a logical world. A "logical world" is a Lisp data structure that captures a logical theory. More precisely, a *logical world* is an encoding of a set of properties (as stored on the property lists in the ACL2 session) that, among other things, assigns a list of formal variables (and hence, an arity) to the FORMALS property of every function symbol in the theory. The ACL2 function arity uses the FORMALS property to determine whether a symbol is a function symbol in the current theory and how many arguments it takes.

A *well-formedness guarantee* for metafunction f is a theorem of the form:

ACL2 Well-Formedness Guarantee:

```
(implies (and (termp x w)
              (arities-okp 'alist w))
         (termp (f x) w))
```

where *alist* is an alist pairing symbols to purported arities and arities-okp checks that the arity recorded in alist agrees with the arity recorded in the logical world w. See Appendix A.

Note that the alist in a well-formedness guarantee presents a finite list of symbols and their purported arities. The theorem does not talk about the current logical theory but *any* logical theory with the same arities for that finite set of symbols. The symbols mentioned in *alist* are typically just the symbols known to the evaluator used in the correctness result for f.

If such a theorem has been proved for a metafunction and cited by the user when the metafunction is proved correct and built into the prover, then no runtime well-formedness check is done on the output of the metafunction. However, a runtime check is done on the alist constant from the well-formedness guarantee and the then-current logical world. That is, we replace a potentially expensive call of termp, on the output of f and the current world, with a typically cheap call of arities-okp on the *alist* of the theorem and the current world.

Let us sketch the correctness argument for skipping the runtime well-formedness check.

Assume that a well-formedness guarantee for f is a theorem of the current ACL2 logical world, Γ. Suppose ACL2 replaces term α by β because `(`f `'`α`)` $=$ `'`β and the `arities-okp` check against Γ is true. Then by the well-formedness theorem for f and the fact that we know `'`α is a `termp` in Γ, we know that `(termp '`β Γ`)` is true; hence, β is a `termp` in Γ. But this is exactly the runtime test that is skipped because of the well-formedness guarantee.

Prior to adding well-formedness guarantees, the function `termp` and some of its subfunctions were defined as ACL2 programs but not admitted into the logic with definitional axioms. To implement our guarantees we had to admit these definitions by proving their termination and verifying their guards. The latter involves proving formulas that establish that when called in accordance with their guards, no program will cause an error (other than resource exhaustion like running out of memory). ACL2 provides the user with facilities for "lifting" any program that is part of the ACL2 implementation into the logic (provided, of course, the program can be proved to terminate and satisfy its guards). When an ACL2 function's guards have been verified and the function is invoked on arguments satisfying its guard, then the compiled Common Lisp code for the function is trusted to comply with ACL2's axioms; otherwise ACL2 runs slower code that ensures compliance.

ACL2 provides for more sophisticated metafunctions than indicated here. For example, a metafunction can have an associated "hypothesis metafunction" that generates a term that must be shown true by the theorem prover before the result of the metafunction is used. In addition, besides the input term to rewrite, a metafunction can take additional arguments that allow the function to access the context in which the target term occurs, the ACL2 world, and the ACL2 state to make heuristic decisions. We do not discuss these extensions here but they are documented in :DOC `well-formedness-guarantee`.

4 User Interface

The ACL2 user tags theorems with tokens that tell the system how the theorem is to be used in the future. The default token is `:REWRITE` and indicates that the theorem is to be interpreted as a conditional rewrite rule. These tokens allow the specification of additional pragmatic attributes. For example, in the case of a rewrite rule, one can rearrange the theorem to identify the left- and right-hand sides of the rewriting equality and identify the conditions that must be relieved before the rule fires, one can limit the resources expended to relieve hypotheses, etc.

To install a metafunction f one states its correctness theorem and tags it with the token `:META`, further annotating it with a list of the top-level function symbols of those terms that should trigger the application of f (since it would be inefficient to try every metafunction on every subterm of every goal). For example, the correctness theorem of a metafunction that actually changes only some terms that are calls of `+` or `*` would be tagged: `(:meta :trigger-fns (+ *))`.

We decided to make well-formedness guarantees optional since the runtime code for checking well-formedness is already present and there are many (approximately 150) metafunctions defined and proved in the ACL2 regression suite.

To provide a well-formedness guarantee one adds an additional annotation so that the one

above becomes:

```
(:meta :trigger-fns (+ *)
       :well-formedness-guarantee name)
```

where *name* is the name of a previously proved well-formedness theorem for the same metafunction. If that metafunction has an associated hypothesis metafunction one must provide the names of two well-formedness theorems, one for the metafunction's output and one for its hypothesis function's output. The downside of this decision is that one must prove well-formedness for both functions if one wants to omit runtime checks. We thought doing otherwise would enable misunderstanding by the user, who might forget to prove the well-formedness of the hypothesis metafunction, say, and have no indication (other than unexplained slowness) that the runtime check on its output is still being done. When two theorems are provided, the arity alists extracted from each must be compatible and they are combined into a single alist to check at application time.

We also decided to exploit the arity alists for another purpose. ACL2 permits the user to declare that a function symbol is "untouchable". This allows a user to define a function, prove theorems about it, and then wall it off from future explicit use. Metafunctions should not introduce such untouchable functions. Every time a well-formedness theorem is proved for a metafunction, we add the function symbols from its arity alist to a growing list. The user is then prohibited from adding functions from that list to the list of untouchables. The design of this feature is actually much more complicated, e.g., to facilitate error messages that "blame" the appropriate metafunction when the user tries to make a now-prohibited symbol untouchable.

5 Performance

The book[5] `meta-wf-guarantee-example.lisp` [6] defines a metafunction and proves its guards, its well-formedness guarantee, and its correctness. It illustrates the ideas in this paper and sets up an experiment the reader can run that demonstrates, on a particular example, the `termp` test taking several thousand times longer than the metafunction. See the book for additional comments on performance.

Of the other ($\sim$ 150) metafunctions defined in the ACL2 regression suite, none have had their well-formedness guarantees proved yet. We discuss this further below.

Our only other test of this as-yet unreleased feature is in the work described in [8], where we discuss "decompiling" a 15,361 instruction machine code program using a formal operational semantics in ACL2, the Codewalker tool[6], and an experimental machine state management book, called Stateman. The state management book is based entirely on metafunctions and these metafunctions can generate very large terms on long paths through code. For example, the 5,280 instruction path through the DES decryption loop generates a state term containing 2,158,895 function calls. This is just one of four states with over two million function symbols generated during the decompilation.

If the metafunctions in Stateman are guard verified but have no well-formedness guarantees, the decompilation time for the 15,361 instructions is 955 seconds on a MacBook Pro laptop

[5] A *book* is an ACL2 input file.

[6] Codewalker extracts ACL2 functions from machine code given the formal operational semantics of the ISA and is similar to the HOL decompilation work by Magnus Myreen[9, 10]. See the `README` file in the Community Book directory `projects/codewalker/`.

with a 2.6 GHz Intel Core i7 processor and 16 GB of 1600 MHZ DDR3 memory, running Clozure Common Lisp. If well-formedness guarantees are proved, the time drops to 618 seconds. Thus we see that the runtime `termp` checks were costing us 337 seconds or about 35% of the time devoted to decompilation.

Unlike many performance related changes, this one has no real downside other than the user's effort to state and prove the guarantees. Once the guarantees are proved, skipping the runtime well-formedness check is always faster than doing it.

The question comes down to how users and maintainers of the regression suite want to spend their time. The metafunctions in Stateman are among the most complicated ever coded by the authors and we found the proofs of their well-formedness guarantees straightforward, following the same decompositions used to prove they return `pseudo-termp`s. Roughly speaking, every function that returns a term should be specified to do so, conditional on appropriate `termp` hypotheses about its arguments. Because ACL2 is untyped, this obvious signature information may not be apparent in the regression suite. The task of proving these `termp` signatures, which seemed daunting in 1989, is now routinely automatic because of engineering improvements in handling "large" definitions like `termp` (see Appendix A). But whether it is worth the time to write out these signatures depends on *the size of the terms the metafunction generates*, which typically depends on the size of the input terms. If one omits the well-formedness guarantees, then the runtime check is performed and that is very fast on small terms. But if the metafunction typically traffics in "galactic" terms, it is worthwhile.

Acknowledgements

This material is based upon work supported by DARPA under Contract No. N66001-10-2-4087 and by ForrestHunt, Inc.

A The Formal Definitions of Termp and Arities-Okp

In this appendix we present the ACL2 definitions of `termp` and its main subfunctions, and `arities-okp`. Here `*main-...` abbreviates `*main-lisp-package-name*`, and `*common-...1` and `*common-...2` abbreviate `*common-lisp-specials-and-constants*` and `*common-lisp-symbols-from-main-lisp-package*`, respectively.

```
(defun legal-variable-or-constant-namep
 (name)
 (and
  (symbolp name)
  (cond
   ((or (eq name t) (eq name nil))
    'constant)
   (t
    (let
      ((p (symbol-package-name name)))
      (and
       (not (equal p "KEYWORD"))
       (let ((s (symbol-name name)))
         (cond
          ((and
            (not (= (length s) 0))
            (eql (char s 0) #\*)
            (eql (char s (1- (length s)))
                 #\*))
           (if (equal p *main-...)
               nil 'constant))
          ((and (not (= (length s) 0))
                (eql (char s 0) #\&))
           nil)
          ((equal p *main-...)
           (and
            (not
```

```
             (member-eq
              name
              *common-...1))
            (member-eq
             name
             *common-...2)
            'variable))
          (t 'variable)))))))))

(defun legal-variablep (name)
 (eq (legal-variable-or-constant-namep
      name)
     'variable))

(mutual-recursion
 (defun termp (x w)
  (declare
   (xargs
    :guard
    (plist-worldp-with-formals w)))
   (cond
    ((atom x) (legal-variablep x))
    ((eq (car x) 'quote)
     (and (consp (cdr x))
          (null (cddr x))))
    ((symbolp (car x))
     (let ((arity (arity (car x) w)))
      (and arity
           (true-listp (cdr x))
           (eql (length (cdr x)) arity)
           (term-listp (cdr x) w))))
    ((and (consp (car x))
          (true-listp (car x))
          (eq (car (car x)) 'lambda)
          (equal 3 (length (car x)))
          (arglistp (cadr (car x)))
          (termp (caddr (car x)) w)
          (null
           (set-difference-eq
            (all-vars (caddr (car x)))
            (cadr (car x))))
          (term-listp (cdr x) w)
          (equal
           (length (cadr (car x)))
           (length (cdr x))))
     t)
    (t nil)))

(defun term-listp (x w)
  (declare
   (xargs
    :guard
    (plist-worldp-with-formals w)))
  (cond ((atom x) (equal x nil))
        ((termp (car x) w)
         (term-listp (cdr x) w))
        (t nil))))

(defun arities-okp (alist w)
 (declare
 (xargs
 :guard
 (and (symbol-alistp alist)
      (plist-worldp-with-formals w))))
 (cond
  ((endp alist) t)
  (t
   (and (equal
         (arity (car (car alist)) w)
         (cdr (car alist)))
        (arities-okp (cdr alist) w)))))
```

References

[1] R. S. Boyer & J S. Moore (1981): *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.* In: *The Correctness Problem in Computer Science*, Academic Press, London.

[2] R. S. Boyer & J S. Moore (1997): *A Computational Logic Handbook, Second Edition.* Academic Press, New York.

[3] W. A. Hunt, Jr., M. Kaufmann, R. B. Krug, J S. Moore & E. W. Smith (2005): *Meta Reasoning in ACL2.* In J. Hurd & T. Melham, editors: *18th International Conference*

on Theorem Proving in Higher Order Logics: TPHOLs 2005, *Lecture Notes in Computer Science* 3603, Springer, pp. 163–178, doi:10.1007/11541868_11.

[4] M. Kaufmann, P. Manolios & J S. Moore, editors (2000): *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA.

[5] M. Kaufmann & J S. Moore (2014): *The ACL2 Home Page*. In: `http://www.cs.utexas.edu/users/moore/acl2/`, Dept. of Computer Sciences, University of Texas at Austin.

[6] M. Kaufmann & J S. Moore (2015): *Example of Well-Formedness Guarantee for a Metafunction*. Technical Report, CS Department, University of Texas at Austin. Available at `https://raw.githubusercontent.com/acl2/acl2/master/books/demos/meta-wf-guarantee-example.lisp`.

[7] Matt Kaufmann, J Strother Moore, Sandip Ray & Erik Reeber (2009): *Integrating External Deduction Tools with ACL2*. Journal of Applied Logic 7(1), pp. 3–25.

[8] J S. Moore (2015): *Stateman: Using Metafunctions to Manage Large Terms Representing Machine States*. In: *ACL2 Workshop 2015*.

[9] Magnus O. Myreen (2009): *Formal verification of machine-code programs*. Ph.D. thesis, University of Cambridge.

[10] Magnus O. Myreen, Konrad Slind & Michael J. C. Gordon (2012): *Decompilation into Logic Improved*. In: *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pp. 78–81.

[11] Anna Slobodova, Jared Davis, Sol Swords & Jr. Warren Hunt (2011): *A Flexible Formal Verification Framework for Industrial Scale Validation*. In Satnam Singh, editor: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, IEEE, pp. 89–97.

[12] G. L. Steele, Jr. (1990): *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803.