Hygienic Macros for ACL2

Carl Eastlund and Matthias Felleisen

Northeastern University
Boston, MA, USA
{cce,matthias}@ccs.neu.edu

Abstract. ACL2 is a theorem prover for a purely functional subset of Common Lisp. It inherits Common Lisp's unhygienic macros, which are used pervasively to eliminate repeated syntactic patterns. The lack of hygiene means that macros do not automatically protect their producers or consumers from accidental variable capture. This paper demonstrates how this lack of hygiene interferes with theorem proving. It then explains how to design and implement a hygienic macro system for ACL2. An evaluation of the ACL2 code base shows the potential impact of this hygienic macro system on existing libraries and practices.

1 Unhygienic Macros Are Not Abstractions

ACL2 [1] is a verification system that combines a first-order functional subset of Common Lisp with a first-order theorem prover over a logic of total functions. It has been used to model and verify large commercial hardware and software artifacts. ACL2 supports functions and logical statements over numbers, strings, symbols, and s-expressions. Here is a sample program:

```
(defun double (x) (+ x x))

(defthm double⇒evenp (implies (integerp x) (evenp (double x))))
```

The **defun** form defines double, a function that adds its input to itself. The **defthm** form defines double⇒evenp, a conjecture stating that an integer input to double yields an even output. The conjecture is implicitly universally quantified over its free variable x. ACL2 validates double⇒evenp as a theorem, using the definition of double and axioms about **implies**, **integerp**, and **evenp**.

From Common Lisp, ACL2 inherits *macros*, which provide a mechanism for extending the language via functions that operate on syntax trees. According to Kaufmann and Moore [2], "*one can make specifications more succinct and easy to grasp ... by introducing well-designed application-specific notation.*" Indeed, macros are used ubiquitously in ACL2 libraries: there are macros for pattern matching; for establishing new homogenous list types and heterogenous structure types, including a comprehensive theory of each; for defining quantified claims using skolemization in an otherwise (explicit) quantifier-free logic; and so on.

In the first-order language of ACL2, macros are also used to eliminate repeated syntactic patterns due to the lack of higher-order functions:

```
(defmacro defun-map (map-fun fun)
  `(defun ,map-fun (xs)
     (if (endp xs)
       nil
       (cons (,fun (car xs)) (,map-fun (cdr xs))))))
```

This macro definition captures the essence of defining one function that applies another pointwise to a list. It consumes two inputs, map-fun and fun, representing function names; the body constructs a suitable **defun** form. ACL2 *expands* uses of defun-map, supplying the syntax of its arguments as map-fun and fun, and continues with the resulting function definition. Consider the following term:

```
(defun-map map-double double)
```

Its expansion fills the names map-double and double into defun-map's template:

```
(defun map-double (xs)
  (if (endp xs)
    nil
    (cons (double (car xs)) (map-double (cdr xs)))))
```

Unfortunately, ACL2 macros are *unhygienic* [3], meaning they do not preserve the meaning of variable bindings and references during code expansion. The end result is accidental capture that not only violates a programmer's intuition of lexical scope but also interferes with logical reasoning about the program source. In short, macros do not properly abstract over syntax.

To make this concrete, consider the **or** macro, which encodes both boolean disjunction and recovery from exceptional conditions, returning the second value if the first is **nil**:

```
(defthm excluded-middle (or (not x) x))

(defun find (n xs) (or (nth n xs) 0))
```

The first definition states the law of the excluded middle. Since ACL2 is based on classical logic, either (**not** x) or x must be true for any x. The second defines selection from a list of numbers: produce the element of xs at index n, or return 0 if **nth** returns **nil**, indicating that the index is out of range.

A natural definition for **or** duplicates its first operand:

```
(defmacro or (a b) `(if ,a ,a ,b))                    (1)
```

This works well for excluded-middle, but the expanded version of find now traverses its input twice, doubling its running time:

```
(defun find (n xs) (if (nth n xs) (nth n xs) 0))
```

Macro users should not have to give up reasoning about their function's running time. Consequently, macros should avoid this kind of code duplication.

The next logical step in the development of **or** saves the result of its first operand in a temporary variable:

```
(defmacro or (a b) `(let ((x ,a)) (if x x ,b)))       (2)
```

This macro now produces efficient and correct code for find. Sadly though, the expanded form of excluded-middle is no longer the expected logical statement:

```
(defthm excluded-middle (let ((x (not x))) (if x x x)))
```

The **or** macro's variable x has captured excluded-middle's second reference to x. As a result, the conjecture is now equivalent to the statement (**not** x).

ACL2 resolves this issue by dealing with the **or** macro as a special case. For symbolic verification, **or** expands using code duplication. For execution, it expands by introducing a fresh variable. The regular macro language of ACL2 does not come with the same expressive power, however. Allowing the creation of fresh variables would introduce uninterned symbols that violate ACL2's axioms and thus corrupt its carefully crafted logical foundation; allowing a separation of executable behavior from the logical semantics would also invite unsoundness.

The **case-match** macro, also provided with ACL2, does not have any such special cases. This macro is used for pattern matching and case dispatch. Its implementation is designed to work around ACL2's lack of hygiene: the macro's expansion never binds any temporary variables. Here is an example use of **case-match** to destructure a 3-element list:

```
(let ((x (quote (1 2 3))))
  (case-match x ((a b c) (list a b c))))
```

The macro expands into the following code:

```
(let ((x (quote (1 2 3))))
  (if (if (consp x)
          (if (consp (cdr x))
              (if (consp (cdr (cdr x)))
                  (eq (cdr (cdr (cdr x))) nil)
                  nil)
              nil)
          nil)
      (let ((a (car x)) (b (car (cdr x))) (c (car (cdr (cdr x)))))
        (list a b c))
      nil))
```

Note that the input to **case-match** is a variable. The macro requires that the user bind the input to a variable, because the input is duplicated many times in the macro's output and the macro cannot safely bind a variable itself. Applications of **car** and **cdr** to walk down the input list are duplicated for the same reason; as a result, the size of the output increases quadratically.

In a hygienic system, **case-match** would be able to safely bind temporary variables in its expanded form. Thus, the user would not need to explicitly bind the input to **case-match** to a variable:

```
(case-match (quote (1 2 3)) ((a b c) (list a b c)))
```

This also makes **case-match** available for use by other macros. In ACL2's unhygienic macro system, other macros cannot safely bind a variable to store **case-match**'s input without risking unintended capture.

Furthermore, the intermediate results of **car** and **cdr** could be bound to temporary variables, yielding fewer function calls in the expanded code. Here is the expansion of the above use of **case-match** produced by one possible implementation in a hygienic macro system:

```
(let ((x0 (quote (1 2 3))))
  (flet ((fail0 () nil))
    (if (consp x0)
      (let ((x1 (car x0)) (y1 (cdr x0)))
        (if (consp y1)
          (let ((x2 (car y1)) (y2 (cdr y2)))
            (if (consp y2)
              (let ((x3 (car y2)) (y3 (cdr y2)))
                (if (eq y3 nil)
                  (let ((a x1) (b x2) (c x3)) (list a b c))
                  (fail0))))
              (fail0)))
          (fail0))
      (fail0))))
```

This version of **case-match** uses temporary variables to perform each **car** and **cdr** only once, producing output with a linear measure.

In general, macro writers tread a fine line. Many macros duplicate code to avoid introducing a variable that might capture bindings in the source code. Others introduce esoteric temporary names to avoid accidental capture. None of these solutions is universal, though. Finding itself in the same place, the Scheme community introduced the notion of *hygienic* macros [3,4,5]. This paper presents an adaptation of hygienic macros to ACL2. It motivates the design and the ACL2-specific challenges, sketches an implementation, and finally presents a comprehensive evaluation of the system vis-a-vis the ACL2 code base.

2 The Meaning of Hygiene for ACL2

Hygienic macro systems ensure that variables in macro-generated code respect the intended lexical scope of the program. Hence, our first step is to analyze the notion of lexical scope in ACL2 and to formulate appropriate goals and policies for the adaptation of hygienic expansion. This section presents the design goals and interprets them in the context of ACL2.

2.1 Design Goals

Our hygienic macro expander is designed to observe four key principles.

Referential transparency means that variables derive their meaning from where they occur and retain that meaning throughout the macro expansion process. Specifically, variable references inserted by a macro refer to bindings inserted by the macro or to bindings apparent at its definition site. Symmetrically, variable references in macro arguments refer to bindings apparent at the macro call site. Following tradition, a hygienic macro system comes with a disciplined method for violating the hygiene condition on individual variables as needed.

Next, *separate compilation* demands that libraries can be expanded, verified, and compiled once and loaded many times. There is no need to re-expand, re-verify, or re-compile a library each time it is used in a new context.

Thirdly, we preserve *logical soundness*. We do not change the logical axioms of ACL2, nor its verification system or compiler. Our few changes to its runtime system are made carefully to observe ACL2's existing axioms. Existing reasoning in the logic of ACL2 remains valid in our system, and execution remains in sync with symbolic reasoning.

Finally, *source compatibility* means that most well-behaved macros continue to function as before. When the revised expansion process affects the behavior of an existing program, the changes are due to a potentially flawed macro.

Unfortunately, we are not able to provide a formal statement of correctness of our macro system with respect to these principles. The correctness of hygienic macros is an open research problem; early proof attempts have since been shown to be flawed. The only known proof of correctness of a hygienic macro system [6] does not support such features as recursive macros, case dispatch during macro expansion, or decomposing lists of arbitrary length during expansion.

2.2 Reinterpreting ACL2

Our hygienic macro system redefines the existing **defmacro** form in ACL2. We do not introduce hygienic macros as a separate mechanism alongside unhygienic macros because hygiene is a property of an entire macro system, rather than a property of individual macro definitions. The implementation of hygiene requires the collaboration of all macros to track the scope of variables; expanding a single macro unhygienically can ruin the benefits of hygiene for all other macros.

Figure 1 specifies the essential core of ACL2. A program is a sequence of definitions. In source code, any definition or expression may be replaced by a macro application; individual functions may be defined outside of **mutual-recursion**; and string or number literals do not require an explicit **quote**. The grammar is written in terms of symbols (*sym*), strings (*str*), and numbers (*num*). A sequence of elements of the form a is denoted $\overrightarrow{a}$, or $\overrightarrow{a}^n$ when its length is significant. We use this core language to explain ACL2-specific challenges to hygienic macro expansion.

Lexical Bindings: ACL2 inherits Common Lisp's namespaces: function and variable bindings are separate and cannot shadow each other. The position of a variable reference determines its role. In an expression position, a variable refers to a value, in application position to a function or macro. For example, the following code uses both kinds of bindings for **car**:

```
(let ((car (car x))) (car car))
```

$def =$	$(\textbf{mutual-recursion}\ \overrightarrow{(\textbf{defun}\ sym\ (\overrightarrow{sym})\ exp)})$	mutually recursive functions
$\mid$	$(\textbf{defmacro}\ sym\ (\overrightarrow{sym})\ exp)$	macro definition
$\mid$	$(\textbf{defthm}\ sym\ exp\ \overrightarrow{(sym\ \overrightarrow{(sym\ exp)})})$	conjecture with proof hints
$\mid$	$(\textbf{include-book}\ str)$	library import
$\mid$	$(\textbf{encapsulate}\ (\overrightarrow{(sym\ num)})\ \overrightarrow{def})$	definition block
$\mid$	$(\textbf{local}\ def)$	local definition
$exp =$	sym	variable reference
$\mid$	$(sym\ \overrightarrow{exp})$	function call
$\mid$	$(\textbf{let}\ (\overrightarrow{(sym\ exp)})\ exp)$	lexical value bindings
$\mid$	$(\textbf{flet}\ (\overrightarrow{(sym\ (\overrightarrow{sym})\ exp)})\ exp)$	lexical function bindings
$\mid$	$(\textbf{quote}\ sexp)$	literal value
$sexp =$	$num \mid str \mid sym \mid (\overrightarrow{sexp})$	s-expression

Fig. 1. Abridged syntax of fully-expanded ACL2 programs

Hygienic expansion must track both function and variable bindings for each possible reference. After all, during expansion, the role of a symbol is unknown until its final position in the expanded code is determined.

Hygienic expansion must also be able to distinguish macro-inserted lexical bindings from those in source code or in other macros. With hygienic expansion, version (2) of the **or** macro in section 1 should work. For example, the excluded-middle conjecture should expand as follows:

(**defthm** excluded-middle (**let** ((x$_2$ (**not** x$_1$))) (**if** x$_2$ x$_2$ x$_1$)))

The macro expander differentiates between the source program's x$_1$ and the macro's x$_2$, as noted by the subscripts; the conjecture's meaning is preserved.

Function Bindings: Functions bound by **flet** are substituted into their applications prior to verification. To prevent unintended capture of free variables during unhygienic expansion, **flet**-bound functions may not refer to enclosing bindings. Consider the following expression that violates this rule:

```
(let ((five 5))
  (flet ((add5 (x) (+ five x))) ;; illegal reference to five, bound to 5
    (let ((five "five"))
      (add5 0))))
```

Under unhygienic expansion, the reference to five in add5 would be captured:

```
(let ((five 5))
  (let ((five "five"))
    (let ((x 0)) (+ five x)))) ;; five is now bound to "five"
```

Hygienic macro expansion allows us to relax this restriction, as lexical bindings can be resolved before substitution of **flet**-bound functions. The same expression expands as follows:

```
(let ((five_1 5))
  (let ((five_2 "five"))
    (let ((x 0)) (+ five_1 x))))
```

Quantification: ACL2 conjectures are implicitly universally quantified:

```
;; claim: ∀x(x > 0 ⇒ x ≥ 0)
(defthm non-negative (implies (> x 0) (≥ x 0)))
```

Here the variable x is never explicitly bound, but its scope is the body of the **defthm** form. ACL2 discovers free variables during the expansion of conjectures and treats them as if they were bound.

This raises a question of how to treat free variables inserted by macros into conjectures. Consider the following definitions:

```
(defmacro imply (var) ‘(implies x ,var))

(defthm x⇒x (imply x))
```

The body of x⇒x expands into (**implies** x_2 x_1), with x_1 from x⇒x and x_2 from imply. In x⇒x, x is clearly quantified by **defthm**. In the template of imply, however, there is no apparent binding for *x*. Therefore, the corresponding variable x_2 in the expanded code must be considered unbound. Enforcing this behavior yields a new design rule: macros must not insert new free variables into conjectures.

We must not overuse this rule, however, as illustrated by the macro below:

```
(defmacro disprove (name body) ‘(defthm name (not ,body)))

(disprove x=x+1 (= x (+ x 1)))
```

Here we must decide what the apparent binding of x is in the body of x=x+1. In the source syntax, there is nothing explicit to suggest that x is a bound or quantified variable, but during expansion, the macro **disprove** inserts a **defthm** form that captures x and quantifies over it. On one hand, allowing this kind of capture violates referential transparency. On the other hand, disallowing it prevents abstraction over **defthm**, because of the lack of explicit quantification.

To resolve this dilemma, we allow **defthm** to quantify over variables from just a single source—surface syntax or a macro. This permits the common macros that expand into **defthm**, but rejects many cases of accidental quantification, a source of bugs in the ACL2 code base. A more disruptive yet sounder alternative would be to introduce explicit quantification into ACL2.

Definition Scope: ACL2 performs macro expansion, verification, and compilation on one definition at a time. Forward references are disallowed, and no definition may overwrite an existing binding.

Nevertheless, just as hygiene prevents lexical bindings from different sources from shadowing each other, it also prevents definitions from different sources from overwriting each other.

Consider the following macro for defining a semigroup based on a predicate recognizing a set and a closed, associative operation over the set:

```
(defmacro semigroup (pred op)
  `(encapsulate ()
     (defthm closed
       (implies (and (,pred a) (,pred b)) (,pred (,op a b))))
     (defthm associative
       (implies (and (,pred a) (,pred b) (,pred c))
         (equal (,op a (,op b c)) (,op (,op a b) c))))))
```

The **semigroup** macro takes two function names as arguments and proves that they form a semigroup. The name, number, and form of definition used in the proof is not part of the macro's interface. In order to leave these names free for reuse, such as in subsequent reuses of the **semigroup** macro, they must not be visible outside the individual macro application.

```
(semigroup integerp +)
(semigroup stringp string-append)
```

Macros must be able to use defined names that originate outside them, however. For instance, the monoid macro uses the previously defined **semigroup** macro to establish a closed, associative operation with an identity element.

```
(defmacro monoid (pred fun zero)
  `(encapsulate ()
     (semigroup ,pred ,fun)
     (defthm identity
       (implies (,pred a)
         (and (equal (,fun ,zero a) a) (equal (,fun a ,zero) a))))))

(monoid rationalp * 1)
```

Macros frequently rely on prior definitions; therefore these definitions must remain visible to the expanded form of macros.

Because prior definitions are visible inside macros, macros must not redefine any name that is visible at their definition. Such a redefinition would allow a logical inconsistency, as the macro would be able to refer to both the old and new meanings for the defined name. The following example shows how redefinition could be used to prove (f) equal to both **t** and **nil**.

```
(defun f () t)

(defmacro bad ()
  `(encapsulate ()
     (defthm f=t (equal (f) t))
     (defun f () nil)
     (defthm f=nil (equal (f) nil))))
```

Our policy for the scope of definitions during hygienic expansion is therefore three-fold. First, defined names from inside macros are not externally visible. Second, macros may refer to any name that is visible at their definition. Third, macros may not redefine any name that is visible at their definition.

Encapsulated Abstractions: The **encapsulate** form in ACL2 delimits a block of definitions. Definitions are exported by default; these definitions represent the block's *constraint*, describing its logical guarantees to the outside. Definitions marked **local** represent a *witness* that can be used to verify the constraint, but they are not exported.

For example, the following block exports a constraint stating that $1 \leq 1$:

```
(encapsulate ()
  (local (defthm x≤x (≤ x x)))
  (defthm 1≤1
    (≤ 1 1) ;; use the following hint:
    (x≤x (x 1))))
```

The local conjecture states that (≤ x x) holds for all values of x. The conjecture 1≤1 states that (≤ *1 1*) holds; the subsequent hint tells ACL2 that the previously verified theorem x≤x is helpful, with *1* substituted for x.

Once the definitions in the body of an **encapsulate** block have been verified, ACL2 discards hints and local definitions (the witness) and re-verifies the remaining definitions (the constraint) in a second pass. The end result is a set of exported logical rules with no reference to the witness. Local theorems may not be used in subsequent hints, local functions and local macros may no longer be applied, and local names are available for redefinition.

An **encapsulate** block may have a third component, which is a set of *constrained functions*. The header of the **encapsulate** form lists names and arities of functions defined locally within the block. The function names are exported as part of the block's constraint; their definitions are not exported and remain part of the witness.

The following block exports a function of two arguments whose witness performs addition, but whose constraint guarantees only commutativity:

```
(encapsulate ((f 2))
  (local (defun f (x y) (+ x y)))
  (defthm commutativity (equal (f x y) (f y x))))
```

Definitions following this block can refer to f and reason about it as a commutative function. Attempts to prove it equivalent to addition fail, however, and attempts to call it result in a run-time error.

Our hygienic macro system preserves the scoping rules of **encapsulate** blocks. Furthermore, it enforces that names defined in the witness are not visible in the constraint, ensuring that a syntactically valid **encapsulate** block has a syntactically valid constraint prior to verification. Our guarantee of referential transparency also means that local names in exported macros cannot be captured. For instance, the following macro m constructs a reference to w:

```
(encapsulate ()
  (local (defun w (x) x))
  (defmacro m (y) ‘(w ,y)))

(defun w (z) (m z)) ;; body expands to: (w z)
```

When a new w is defined outside the block and **m** is applied, the new binding does not capture the w from **m**. Instead, the macro expander signals a syntax error, because the inserted reference is no longer in scope.

Books: A *book* is the unit of ACL2 libraries: a set of definitions that is verified and compiled once and then reused. Each book acts as an **encapsulate** block without constrained functions; it is verified twice—once with witness, and once for the constraint—and the constraint is saved to disk in compiled form. When a program includes a book, ACL2 incorporates its definitions, after ensuring that they do not clash with any existing bindings.

ACL2 allows an exception to the rule against redefinition that facilitates compatibility between books. Specifically, a definition is considered *redundant* and skipped, rather than rejected, if it is precisely the same as an existing one. If two books contain the same definition for a function f, for instance, the books are still compatible. Similarly, if one book is included twice in the same program, the second inclusion is considered redundant.

This notion of redundancy is complicated by hygienic macro expansion. Because hygienic expanders generally rename variables in their output, there is no guarantee that identical source syntax expands to an identical compiled form. As a result, redundancy becomes a question of α-equivalence instead of simple syntactic equality. Coalescing redundant definitions in compiled books would thus require renaming all references to the second definition. This code rewriting defeats the principle of separate compilation.

Rather than address redundancy in its full generality, we restrict it to the case of loading the same book twice. If a book is loaded twice, the new definitions will be syntactically equal to the old ones because books are only compiled once. That is, this important case of redundancy does not rely on α-equivalence, and thus allows us to load compiled books unchanged.

Macros: Macros use a representation of syntax as their input and output. In the existing ACL2 system, syntax is represented using primitive data types: strings and numbers for literals, symbols for variables, and lists for sequences of terms.

Hygienic macro systems must annotate syntax with details of scope and macro expansion. Kohlbecker et al. [3] incorporate these annotations into the existing symbol datatype; in contrast, Dybvig et al. [5] introduce a separate class of *syntax objects*. To preserve existing ACL2 macros, we cannot introduce an entirely new data type; instead, we adopt the former method.

In adapting the symbol datatype, we must be sure to preserve the axioms of ACL2. On one hand, it is an axiom that any symbol is uniquely distinguished by the combination of its name and its *package*—an additional string used for manual namespace management. On the other hand, the hygienic macro expander must distinguish between symbols sharing a name and a package when one originates in the source program and another is inserted by a macro. We resolve this issue by leaving hygienic expansion metadata transparent to the logic: only macros and unverified, *program mode* functions can distinguish between two symbols with the same name and package. Conjectures and verified, *logic mode* functions cannot make this distinction, i.e., ACL2's axioms remain valid.

The symbols inserted by macros must derive their lexical bindings from the context in which they appear. To understand the complexity of this statement, consider the following example:

```
(defun parse-compose (funs arg)
  (if (endp funs) arg `(,(car funs) (compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg) (parse-compose funs arg))

(compose (string-downcase symbol-name) (quote SYM))
;; ⇒ (string-downcase (compose (symbol-name) (quote SYM)))
```

The auxiliary function parse-compose creates recursive references to compose, but compose is not in scope in parse-compose. To support this common macro idiom, we give the code inserted by macros the context of the macro's definition site. In the above example, the symbol *compose* in parse-compose's template does not carry any context until it is returned from the compose macro, at which point it inherits a binding for the name. This behavior allows recursive macros with helper functions, at some cost to referential transparency: the reference inserted by parse-compose might be given a different meaning if used by another macro.

This quirk of our design could be alleviated if these macros were rewritten in a different style. If the helper function parse-compose accepted the recursive reference to compose as an argument, then the quoted symbol *compose* could be passed in from the definition of compose itself, where it has meaning:

```
(defun parse-compose (compose funs arg)
  (if (endp funs) arg `(,(car funs) (,compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg) (parse-compose (quote compose) funs arg))
```

Symbols in macro templates could then take their context from their original position, observing referential transparency. However, to satisfy our fourth design goal of source compatibility and accommodate common ACL2 macro practice, our design does not mandate it.

Breaking Hygiene: There are some cases where a macro must insert variables that do not inherit the context of the macro definition, but instead intentionally capture—or are captured by—variables in the source program. For instance, the defun-map example can be rewritten to automatically construct the name of the map function from the name of the pointwise function:

```
(defmacro defun-map (fun)
  (let ((map-fun-string (string-append "map-" (symbol-name fun))))
    (let ((map-fun (in-package-of map-fun-string fun)))
      `(defun ,map-fun (xs)
         (if (endp xs)
           nil
           (cons (,fun (car xs)) (,map-fun (cdr xs))))))))

(defun-map double) ;; expands to: (defun map-double (xs) ...)
```

state	$= \langle \mathit{str}, \mathit{bool}, \mathit{bool}, \mathit{ren}, \mathit{table}, \{\overrightarrow{\mathit{sym}}\}, \{\overrightarrow{\mathit{key}}\}\rangle$	expansion state
table	$= [\overrightarrow{\mathit{sym} \mapsto \mathit{rec}}]$	def. table
rec	$= \langle \mathit{sig}, \mathit{fun}, \mathit{thm}\rangle$	def. record
sig	$= \mathsf{fun}(\mathit{bool}, \mathit{num}) \mid \mathsf{macro}(\mathit{id}, \mathit{num}) \mid \mathsf{thm}(\{\overrightarrow{\mathit{sym}}\}) \mid \mathsf{special}$	def. signature
fun^n	$= \cdot \mid \overrightarrow{\mathit{sexp}}^n \rightarrow \mathit{sexp}$	n-ary function
thm	$= \cdot \mid \cdots$	theorem formula
sexp	$= \mathit{num} \mid \mathit{str} \mid \mathit{id} \mid \mathsf{cons}(\mathit{sexp}, \mathit{sexp})$	s-expression
id	$= \mathit{sym} \mid \mathsf{id}(\mathit{sym}, \{\overrightarrow{\mathit{mark}}\}, \mathit{ren}, \mathit{ren})$	identifier
sym	$= \mathsf{sym}(\mathit{str}, \mathit{str}, \{\overrightarrow{\mathit{mark}}\})$	symbol
bool	$= \mathbf{t} \mid \mathbf{nil}$	boolean
ren	$= [\overrightarrow{\mathit{key} \mapsto \mathit{sym}}]$	renaming
key	$= \langle \mathit{sym}, \{\overrightarrow{\mathit{mark}}\}\rangle$	identifier key
mark	$= \langle \mathit{str}, \mathit{num}\rangle$	mark

Fig. 2. Representation of expansion state and s-expressions

In this macro, the name double comes from the macro caller's context, but map-double is inserted by the macro itself. The macro's intention is to bind map-double in the caller's context, and the caller expects this name to be bound.

This implementation pattern derives from the Common Lisp package system. Specifically, the **in-package-of** function builds a new symbol with the given string as its name, and the package of the given symbol. In our example, map-double is defined in the same package as double.

We co-opt the same pattern to transfer lexical context. Thus the name map-double shares double's context and is visible to the macro's caller. Macro writers can use **in-package-of** to break the default policy of hygiene.

3 Hygienic Macro Expansion

The ACL2 theorem prover certifies saved books and verifies interactive programs using a process of iteratively expanding, verifying, and compiling each term in turn. The expansion process takes each term and produces a corresponding, fully-expanded definition; it also maintains and updates an *expansion state* recording the scope and meaning of existing definitions so far. Our hygienic macro system requires these changes: an augmented representation of unexpanded terms and expansion state; an adaptation of Dybvig et al.'s expansion algorithm [5]; and new versions of ACL2's primitives that manipulate the new forms of data while satisfying existing axioms.

Figure 2 shows the definition of expansion states. An expansion state contains seven fields. The first names the source file being expanded. The second and third determine expansion modes: global versus local definition scope and logic mode versus program mode. Fields four and five provide mappings on the set of compiled definitions; the fourth is added for hygienic expansion to map bindings in source code to unique compiled names, and the fifth is ACL2's mapping from compiled names to the meaning of definitions. The sixth field is the subset of compiled definition names that are exported from the enclosing scope, and the seventh is the set of constrained function names that have been declared but not yet defined; we update this final field with hygienic metadata to properly track macro-inserted bindings.

A *definition table* maps each definition to a record describing its signature, executable behavior, and logical meaning. We use ACL2's existing definition signatures; we augment macro signatures to carry an identifier representing the lexical context of the macro's definition. An executable function implements a function or macro, and a logical formula describes a function or theorem equationally; we do not change either representation.

Figure 2 also shows the low-level representation of s-expressions. Symbols and sequences as shown in figure 1 are represented using the sym and cons constructors, respectively. An s-expression is either a number, a string, an identifier, or a pair of s-expressions. Numbers and strings are unchanged. The most important difference to a conventional representation concerns *identifiers*, which extend symbols to include information about expansion. An identifier is either a symbol or an annotated symbol. A symbol has three components: its name, its package, and a set of *inherent marks* used to support unique symbol generation. Annotated symbols contain a symbol, a set of *latent marks* used to record macro expansion steps, and two renamings; unlike standard identifier representations, we must differentiate function and value bindings. We represent booleans with symbols, abbreviated **t** and **nil**.

Identifiers represent variable names in unexpanded programs; unique symbol names are chosen for variables in fully expanded programs. The mapping between the two is mediated by *keys*. Each function or value binding's key combines the unique symbol corresponding to the shadowed binding—or the unmodified symbol if the name has no prior binding—and the (latent) marks of the identifier used to name the binding. A *renaming* maps keys to their corresponding symbols.

A *mark* uniquely identifies an event during macro expansion: a variable binding or single macro application. Each one comprises its source file as a string—to distinguish marks generated during the compilation of separate books, in an adaptation of Flatt's mechanism for differentiating bindings from separate modules [7]—as well as a number chosen uniquely during a single session.

This representation of s-expressions is used both for syntax during macro expansion and for values during ordinary runtime computation. Hence, ACL2 functions that deal with symbols must be updated to work with identifiers in a way that observes the axioms of regular symbols. The basic symbol observations name, package, eq, and symbolp are defined to ignore all identifier metadata. The symbol constructor intern produces a symbol with empty lexical context, while in-package-of copies the context of its second argument.

We also introduce four new identifier comparisons: $=^b_f$, $=^r_f$, $=^b_v$, and $=^r_v$. They are separated according to the ACL2 function and value namespaces, as signified by the subscripts, and to compare either binding occurrences or references, as signified by the superscripts. *These procedures do not respect ACL2's axioms.*

They can distinguish between symbols with the same name and package, so we provide them in program mode only. As such, they may be used in macros as variable comparisons that respect apparent bindings.

4 Evaluating Hygiene

Our design goals for hygienic ACL2 macros mention four guiding principles: referential transparency, separate compilation, logical soundness, and source compatibility. As explained, the macro expansion process preserves referential transparency by tracking the provenance of identifiers, with two key exceptions: symbols inserted by macros take their context from the macro definition site rather than their own occurrence, and conjecture quantification can "capture" free variables in macro inputs. Furthermore, our representation for compiled books guarantees separate compilation. We preserve logical soundness by obeying ACL2's axioms for symbols in operations on identifiers, and by reusing the existing ACL2 compiler and theorem proving engine. Only the principle of source compatibility remains to be evaluated.

Our prototype does not support many of the non-macro-related features of ACL2 and we are thus unable to run hygienic expansion on most existing books. To determine the degree of compatibility between our system and existing macros, we manually inspected all 2,954 **defmacro** forms in the books provided with ACL2 version 3.6, including the separate package of books accumulated from the ACL2 workshop series. Of these, some 488 nontrivial macros might be affected by hygiene. The rest of the section explains the details.

Code Duplication: The behavior of macro-duplicated code does not change with hygienic expansion; however, hygiene encourages the introduction of local variables in macros and thus avoids duplication. With our system, all 130 code-duplicating macros can be rewritten to benefit from hygiene.

Variable Comparison: Comparing variable names with eq does not take into account their provenance in the macro expansion process and can mistakenly identify two symbols with the same name but different lexical contexts. We found 26 macros in ACL2 that compare variable names for assorted purposes, none of which are served if the comparison does not respect the variable's binding. The new functions $=^b_f$, $=^r_f$, $=^b_v$, and $=^r_v$ provide comparisons for variables that respect lexical context. Once again, the result of eq does not change in our system, so these macros will function as they have; however, macro writers now have the option of using improved tools. Each of the 26 macros can be rewritten with these functions to compare variable names in a way that respects lexical bindings during macro expansion.

Free Variables: Free variables in macros usually represent some protocol by which macros take their meaning from their context; i.e., they must be used in a context where the names in question have been bound. Much like mutable state in imperative languages, free variables in macros represent an invisible channel of communication. When used judiciously, they create succinct programs, but they can also be a barrier to understanding. Of the 90 macros that insert free variables, 83 employ such a protocol. Our hygienic macro expander rejects such macros; they must be rewritten to communicate in an explicit manner.

Five further cases of free variables are forward references, in which a macro's body constructs a reference to a subsequent definition. To a macro writer, this may not seem like a free reference, but it is, due to the scope of ACL2 definitions. Therefore this use of forward references does not satisfy the principle of referential transparency. These macros must also be rewritten or reordered to mesh with hygienic macro expansion.

The final two cases of free variables in a macro are, in fact, symptoms of a single bug. The macro is used to generate the body of a conjecture. It splices several expressions into a large implication. One of the inputs is named top, and its first reference in the macro is accidentally quoted—instead of filling in the contents of the input named top, the macro inserts a literal reference to a variable named top. By serendipity, this macro is passed a variable named top, and nothing goes wrong. Were this macro ever to be used with another name, it would construct the wrong conjecture and either fail due to a mysterious extra variable or succeed spuriously by proving the wrong proposition. Our hygienic macro system would have flagged this bug immediately.

Variable Capture: We found 242 instances of variable (85) or definition (157) names inserted by macros that introduce bindings to the macro's input or surrounding program. Of the macros that insert definition names, there were 95 that used in-package-of to explicitly bind names in the package of their input, 44 that used intern to bind names in their own package, 16 that used hard-coded names not based on their input at all, and two that used the make-event facility [8] to construct unique names.

The package-aware macros will continue to function as before due to our interpretation of in-package-of. As written, the intern-based macros guarantee neither that the constructed names bind in the context of the input, nor that they don't, due to potential package mismatches. Hygienic expansion provides a consistent guarantee that they don't, making their meaning predictable. Hard-coded names in macros will no longer bind outside of the macro itself. These are the other side of free variable protocols; they must be made explicit to interoperate with hygiene. The make-event utility allows inspection of the current bindings to construct a unique name, but nothing prevents that name from clashing with any subsequent binding. Hygiene eliminates the need to manually scan the current bindings and guarantees global uniqueness.

Lexical variables account for the other 85 introduced bindings. We discovered nine whose call sites exploited these bindings as part of an intentional protocol. These macros can be made hygienic by taking the variable name in question as an argument, thus making the macro compatible with hygienic expansion, freeing up a name the user might want for something else, and avoiding surprises if a user does not know the macro's protocol.

Of the other 76 macros that bind local variables in the scope of their arguments, 59 attempt to avoid capture. There are 12 that choose long, obscure names; for instance, gensym::metlist (meaning "metlist" in the "gensym" package), indicating

	Improves for free	Improves with work	Unchanged	Broken; improves	Broken; restores
Code Duplication	–	130	–	–	–
Free variable	2	–	–	83	5
Lexical capture	29	47	–	9	–
Definition capture	–	2	95	44	16
Variable comparison	–	26	–	–	–
Total	31	205	95	136	21

Fig. 3. Impact of hygienic expansion on nontrivial ACL2 macros

a wish for the Lisp symbol-generating function gensym, which is not available in ACL2. There is also a convention of adding -do-not-use-elsewhere or some similar suffix to macro-bound variables; in one case, due to code copying, a variable named hyp--dont-use-this-name-elsewhere is in fact bound by *two* macros in different files. Obscure names are a poor form of protection when they are chosen following a simple formula, and a macro that binds a hard-coded long name will never compose properly with itself, as it always binds the same name.

A further 40 macros generate non-capturing names based on a known set of free variables, and seven more fail with a compile error if they capture a name as detected by check-vars-not-free. These macros are guaranteed not to capture, but the latter still force the user to learn the name bound by the macro and avoid choosing it for another purpose. Some of these macros claim common names, such as val and x, for themselves.

Finally, we have found 17 macros in the ACL2 books that bind variables and take no steps to avoid capture. All of the accidentally variable-capturing macros will automatically benefit from hygienic expansion.

Exceptions: The notable exceptions to hygiene we have not addressed are make-event, a tool for selective code transformation, and **state**, a special variable used to represent mutation and i/o. We have not yet inspected most uses of make-event in the ACL2 code base, but do not anticipate any theoretical problems in adapting the feature. For **state** and similar "single-threaded" objects, our design must change so as to recognize the appropriate variables and not rename them.

Summary: Figure 3 summarizes our analysis. We categorize each macro by row according to the type of transformation it applies: code duplication, free variable insertion, capture of lexical or definition bindings, and variable comparison. We omit the trivial case of simple alias macros from this table.

We split the macros by column according to the anticipated result of hygienic expansion. In the leftmost column, we sum up the macros whose expansion is automatically improved by hygienic expansion. Next to that, we include macros that work as-is with hygiene, but permit a better definition. In the center, we tally the macros whose expansion is unaffected. To the right, we list macros that must be fixed to work with hygienic macro expansion, but whose expansion becomes more predictable when fixed. In the rightmost column, we list those macros that must be fixed, yet do not benefit from hygienic expansion.

Many libraries and built-in features of ACL2 rely on the unhygienic nature of expansion and use implicit bindings; as a result, our system cannot cope with every macro idiom in the code base. These macros must be rewritten in our system. We anticipate that all of the macros distributed with ACL2 can be fixed straightforwardly by either reordering definitions or adding extra arguments to macros. However, this process cannot be automated and is a potential source of new errors. Fortunately, the bulk of macros will continue to work, and we expect most of them to benefit from hygiene. The frequent use of code duplication, obscure variable names, and other capture prevention mechanisms shows that ACL2 users recognize the need for a disciplined approach to avoiding unintentional capture in ACL2 macros.

5 Related Work and Conclusions

ACL2 is not the only theorem prover equipped with a method of syntactic extensions. PVS has macros [9]; however, they are restricted to definitions of constants that are inlined during the type-checking phase. As a result, preserving the binding structure of the source program is simple.

The Agda, Coq, Isabelle, and Nuprl theorem provers all support extensible notation. These include issues of parsing, precedence, and associativity that do not arise in ACL2's macros, which are embedded in the grammar of s-expressions. The notation mechanisms of Agda and Isabelle are limited to "mixfix" operator definitions [10,11]. These definitions do not introduce new variable names in their expansion, so the problem of variable capture does not arise.

Nuprl and Coq have notation systems that permit the introduction of new binding forms. Nuprl requires each notational definition to carry explicit binding annotations. These annotations allow Nuprl to resolve variable references without the inference inherent in hygienic macro systems [12]. The notation system of Coq ensures that introduced variables do not capture source program variables and vice versa [13], although the precise details of this process are undocumented. Neither Nuprl nor Coq allow case dispatch or self-reference in notational definitions. Our work combines the predictability of variable scope present in Nuprl and Coq notation with the expressive power of ACL2 macros.

Hygienic macros have been a standardized part of the Scheme programming language for over a decade [14]. They have been used to define entire new programming languages [15,16], including an implementation of the runtime components of ACL2 in Scheme [17]. These results are feasible because of hygiene and are facilitated by further advances in macro tools [7,18].

With hygienic macros, ACL2 developers gain the power to write more trustworthy and maintainable proofs using macros. Furthermore, adding a scope-respecting macro mechanism is a necessary step for any future attempt to make ACL2 reason about its source programs directly instead of expanded terms. Our techniques may also be useful in adapting hygienic macros to languages other than Scheme and ACL2 that have different binding constructs, different scope mechanisms, multiple namespaces, implicit bindings, and other such features.

At the 2009 ACL2 Workshop's panel on the future of theorem proving, panelist David Hardin of Rockwell Collins stated a desire for domain-specific languages in automated theorem proving. This paper is the first of many steps toward user-written, domain-specific languages in ACL2.

References

1. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: an Approach. Kluwer Academic Publishers, Dordrecht (2000)
2. Kaufmann, M., Moore, J.S.: Design goals of ACL2. Technical report, Computational Logic, Inc. (1994)
3. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Proc. 1986 ACM Conference on LISP and Functional Programming, pp. 151–161. ACM Press, New York (1986)
4. Clinger, W., Rees, J.: Macros that work. In: Proc. 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 155–162. ACM Press, New York (1991)
5. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and Symbolic Computation 5(4), 295–326 (1992)
6. Herman, D., Wand, M.: A theory of hygienic macros. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 48–62. Springer, Heidelberg (2008)
7. Flatt, M.: Composable and compilable macros: you want it when? In: Proc. 7th ACM SIGPLAN International Conference on Functional Programming, pp. 72–83. ACM Press, New York (2002)
8. Kaufmann, M., Moore, J.S.: ACL2 Documentation (2009), `http://userweb.cs.utexas.edu/users/moore/acl2/current/acl2-doc.html`
9. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference (2001), `http://pvs.csl.sri.com/doc/pvs-language-reference.pdf`
10. Danielsson, N.A., Norell, U.: Parsing mixfix operators. In: Proc. 20th International Symposium on the Implementation and Application of Functional Languages, School of Computer Science of the University of Hertfordshire (2008)
11. Wenzel, M.: The Isabelle/Isar Reference Manual (2010), `http://isabelle.in.tum.de/dist/Isabelle/doc/isar-ref.pdf`
12. Griffin, T.G.: Notational definition—a formal account. In: Proc. 3rd Annual Symposium on Logic in Computer Science, pp. 372–383. IEEE Press, Los Alamitos (1988)
13. The Coq Development Team: The Coq Proof Assistant Reference Manual (2009), `http://coq.inria.fr/coq/distrib/current/refman/`
14. Kelsey, R., Clinger, W., Rees, J. (eds.): Revised^5 report on the algorithmic language Scheme. ACM SIGPLAN Notices 33(9), 26–76 (1998)
15. Gray, K., Flatt, M.: Compiling Java to PLT Scheme. In: Proc. 5th Workshop on Scheme and Functional Programming, pp. 53–61 (2004)
16. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: Proc. 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 395–406. ACM Press, New York (2008)
17. Vaillancourt, D., Page, R., Felleisen, M.: ACL2 in DrScheme. In: Proc. 6th International Workshop on the ACL2 Theorem Prover and its Applications, pp. 107–116 (2006)
18. Culpepper, R.: Refining Syntactic Sugar: Tools for Supporting Macro Development. PhD dissertation, Northeastern University (2010)