`defstructure` for ACL2 Version 2.0

Bishop Brock*
Computational Logic, Inc.
brock@cli.com

December 1, 1997

Abstract

This article documents the `defstructure` macro, a facility provided by the ACL2 book

`acl2-sources/books/data-structures/structures.lisp`

as distributed with Release 2.0 of ACL2. `defstructure` is an ACL2 macro for defining general purpose record structures. `defstructure` also creates an extensive theory for automated reasoning about specifications defined in terms of the structures. The documentation for `defstructure` included in the source code for the book was not kept up to date as `defstructure` was developed. This article is the definitive documentation for `defstructure`.

*The author's current address is IBM Austin Research Laboratory, 11501 Burnet Road, MS/9460, Austin, Texas 78758. brock@austin.ibm.com

Contents

1 Introduction

`defstructure` is an ACL2 macro that creates and characterizes general purpose record structures. The utility of `defstructure` can be broadly divided into two areas. First, `defstructure` provides a convenient way to group and access data, rendering specifications more readable and modular. This may be sufficient for users only interested in programming in ACL2. `defstructure` also creates an extensive theory about the creation, access, and update of the defined structures, providing many of the definitions and lemmas that a user will normally need to be able to effectively and automatically reason about ACL2 specifications that use the structures. Second, `defstructure` provides a convenient way to describe the type theory of the slots of the structures. Having a strong and comprehensive type theory is often a prerequisite for the efficient creation of guarded ACL2 specifications. `defstructure` allows the user to provide type assertions about each slot of the structure, and to describe how the type assertions are to be stored as ACL2 rules.

Although `defstructure` can be used in a definition-only environment where no proofs are being done, it is designed to fully support formal proof in the context of guarded functions. One of the most important aspects of the theory created by `defstructure` is that the structure objects are accessible only by the accessing and updating functions. Technically, `defstructure` structures are simply ACL2 lists with a few special properties, and each slot is associated with a generalized position in the list. However, `defstructure` does everything in its power to make it impossible for the ACL2 prover to 'see inside' the structure. This makes symbolic reasoning about the structures very efficient, as the logical representation of a structure (an uninterpreted function application) is much more compact that its actual, list-based representation. This design also helps make theorem prover output more readable since structure occurrences will never be expanded into their actual list-based representations.

The remainder of this note is divided into two sections. Section 2 is a user's guide to using `defstructure`. This section describes by way of examples how to use `defstructure` for programming, how to provide type assertions for structures, how to use `defstructure` to help in reasoning about ACL2 functions that return multiple values, and a brief section on recursive structures. Section 3 is a reference guide to `defstructure`. This section includes a formal syntax description of `defstructure` followed by more detailed documentation of all of the `defstructure` options.

This article also includes two appendices. Appendix A contains 'real-world' examples of `defstructure` collected from various sources. Appendix B contains the macro expansion of a simple `defstructure` form.

2 User's Guide

This section is a short user's guide to `defstructure`. The syntax and semantics of `defstructure` and its generated theory were inspired by the Common Lisp `defstruct` facility and the Nqthm `add-shell` facility. Many of the options and generated functions and macros are identical to their Common Lisp `defstruct` analogues. Therefore it may be helpful for the reader to skim the `defstruct` section of a Common Lisp textbook or manual before using `defstructure`.

`defstructure` is distributed with ACL2 as an ACL2 *book*. Prior to using `defstructure`, this book should be *certified* as described in the ACL2 installation guide. To use `defstructure` it is first necessary to load the book into the ACL2 session with the `include-book` command. The actual filename of the book will be installation-dependent, but for example on the author's system the ACL2 command is

```
(include-book "/usr/local/source/acl2-sources/books/data-structures/structures")
```

`defstructure` is an ACL2 macro whose expansion generates a large number of ACL2 forms. `defstructure` is a 'self-documenting' macro in the sense that the macro expansion of an instance of `defstructure` includes commentary about all of the generated forms. To see this documentation it is necessary to view the translated form of `defstructure`. This can be done by using the ACL2 `trans1` command at the ACL2 prompt as in the example below.

```
ACL2 !>:trans1 (defstructure struct a b c)
```

An example macroexpansion of a `defstructure` macro appears as Appendix B.

In this user's guide we describe the use of `defstructure` by means of examples. Section 2.1 begins with an overview of the basic use of `defstructure` for ACL2 programming. Section 2.2 describes how to define and use structures with typed slots. Section 2.3 is a brief introduction to using `defstructure` for reasoning about functions that return multiple values, and Section 2.4 introduces recursive structures. See Section 3 for a more complete reference manual on `defstructure`.

2.1 Basic Use

We demonstrate the basic use of `defstructure` by means of the following example, a generic structure named *struct* with three slots named a, b, and c.

```
(defstructure struct a b c)
```

The above form creates a *constructor* function named `struct` of three arguments, namely the values for the three slots in the order in which they were declared.

```
(struct a b c) ≡ (list 'struct a b c)

(struct 0 'x "foo") ⟹ (struct 0 x "foo")
```

The *structure* is simply a list whose *car* is `'struct`, whose *cadr* stores the value of slot a, *caddr* stores the value of slot b and *cadddr* stores the value of slot c. Instances of this structure are recognized by the *predicate* `struct-p`, and the *weak-predicate* `weak-struct-p`.

```
(struct-p (struct a b c)) ⟹ t
(weak-struct-p (struct a b c)) ⟹ t
```

For untyped structures such as `struct` the *predicate* and *weak-predicate* are identical. The *predicate* and *weak-predicate* only differ for typed structures (see Section 2.2).

The slots of an instance of the structure are accessed by the *reader* functions `struct-a`, `struct-b`, and `struct-c`.

```
(struct-a (struct a b c)) ⟹ a
(struct-b (struct a b c)) ⟹ b
(struct-c (struct a b c)) ⟹ c
```

Instances of the structure may also be created with the *keyword-constructor* named `make-struct`, a macro that creates an instance of the structure and initializes the slots from a keyword argument list.

```
(make-struct :c "foo" :b 'x :a 0) ⟹ (struct 0 x "foo")
```

The *keyword-constructor* allows the slots to be specified in any order. Note that values for every slot must be specified unless default values have been declared for the slots (see Section 3.3.5).

Modified copies of an instance can be created by means of the *keyword-updater* macro named `update-struct`. New values for any or all of the slots can be specified by means of a keyword argument list.

```
(let ((struct (struct 0 'x "foo")))
  (update-struct struct :c "bar" :a 1))
⟶
(struct 1 (struct-b struct) "bar")
⟹
(struct 1 x "bar")
```

As shown above, the *keyword-updater* works by simply copying any unmodified slots from the old instance to the new instance.

By default, no guard assertions are attached to the functions created by `defstructure`. One can specify that guards should be generated by means of the `:guards` option as in the following example.

```
(defstructure struct a b c (:options :guards))
```

If the `:guards` option is specified, then the *predicate* and *weak-predicate* will be defined with guards of `t` (that is, unguarded), and the *reader* functions will be guarded by the *weak-predicate*.

There isn't much else to know about `defstructure` for ACL2 programming in `program` mode. `defstructure` does provide a number of options for customization of generated names etc., as detailed in Section 3.3.

2.2 Typed Structures

One of the most important features of `defstructure` in the context of guarded specifications is the ability to create a theory of assertions about the slots of the structure. Because these assertions usually have the form of type assertions on the slots of the structure, we refer to these as typed structures.

For example, imagine a simple structure with three slots: n, a prime number, o, an arbitrary object, and s, a string. This could be specified as

```
(defstructure struct
  (n (:assert (primep n)))
  o
  (s (:assert (stringp s))))
```

Note that the assertions are *not* guards on the *constructor*. Adding assertions to a structure simply differentiates the *predicate* from the *weak-predicate*. The *weak-predicate* for `struct` recognizes any 4-element proper list whose *car* is `struct`, while the *predicate* extends the *weak-predicate* to include the conditions that the *cadr* is `primep` the *cadddr* is a string.

```
(weak-struct-p (struct 7 'x "foo")) ⟹ t
(struct-p      (struct 7 'x "foo")) ⟹ t

(weak-struct-p (struct 4 'x 0)) ⟹ t
(struct-p      (struct 4 'x 0)) ⟹ nil
```

Assertions may be attached to the slots as in the above example, or to the structure as a whole; there is no difference in meaning. The above could just as well have been written as

```
(defstructure struct n o s
  (:options
   (:assert (primep n))
   (:assert (stringp s))))
```

or even

```
(defstructure struct n o s
  (:options (:assert (and (primep n)
                          (stringp s)))))
```

although attaching assertions to the slots may provide clearer documentation of the specification. Assertions can be arbitrary predicates on multiple slots. For example we can easily add the requirement that n is less than the length of s.

```
(defstructure struct
  (n (:assert (primep n)))
  o
  (s (:assert (stringp s)))
  (:options (:assert (< n (length s)))))
```

By themselves, assertions document a specification and provide a *predicate* that can be used to test any ACL2 object for conformance with a specification. When creating complex guarded specifications using typed structures it is often necessary to store the assertions about the structure slots as `:rewrite` rules or other types of rules.[1] Specifying that an assertion be stored as a lemma is simply a matter of adding the proper `:rule-classes`.

```
(defstructure struct
  (n (:assert (primep n) :rewrite))
  o
  (s (:assert (stringp s) :rewrite))
  (:options :guards))
```

The above example adds to the lemmas generated by `defstructure` a lemma named `defs-struct-assertions`. This lemma generates the two `:rewrite` rules implied by the above `defstructure` form, namely

```
(implies (struct-p struct)
         (primep (struct-n struct)))

(implies (struct-p struct)
         (stringp (struct-s struct)))
```

[1]The type lemmas are not necessary for only the simplest uses of typed structures where the prover can get at the type information stored in the *predicate* by eliminating the *readers* in favor of the *constructor*, then opening the *predicate* on a symbolic call of the *constructor*.

Sometimes it is beneficial to store multiple rule classes for a single slot. For example, `(primep n)` is a fairly strong assertion about slot n. It may turn out to be useful to know simply that n is a positive integer, which could be stated as a `:type-prescription`.

```
(defstructure struct
  (n (:assert (primep n) :rewrite
              (:type-prescription (and (integerp n)
                                       (< 0 n))))
  o
  (s (:assert (stringp s) :rewrite))
  (:options :guards))
```

This adds the following `:type-prescription` rule to `defs-struct-assertions`.

```
(implies (struct-p struct)
         (and (integerp (struct-n struct))
              (< 0 (struct-n struct))))
```

The above example demonstrates the basic syntax of specifying `:rule-classes` for assertions. If the assertion is the `:corollary` for the rule-class, it suffices to specify the name of the rule-class. Otherwise, the rule-class is given along with the corollary formula. More complete information on `defstructure` assertions appears in Section 3.4.

2.3 Multiple Value Structures

Because `defstructure` structures are just lists, `defstructure` can be used to define 'templates' for list structures created elsewhere. One well-supported instance of this idea is the `defstructure` facility for dealing with multiple-value functions. In ACL2, a function that returns multiple values logically returns a list of values, which lists are only destructured with the function `mv-nth`. Syntactic restrictions in ACL2 insure that ACL2 multiple-value functions may be efficiently executed as real Common Lisp multiple-value functions with the same semantics.

If the `defstructure` option `(:representation :mv)` is selected, then the structure created by `defstructure` will be a *multiple-value structure*. Under this option, the definitions and theories created by `defstructure` are designed to support the use of the structure as a template for manipulating multiple values. The *constructor* will be defined as a macro that simply expands into a `mv` form, and the structure will be defined as a simple list (without the usual structure name 'tag') with typed slots as indicated. The types of operations that may be performed on the structure are very limited. In particular, it is not possible to 'pass around' uninterpreted instances of the structure since a multiple-value list is really an ephemeral object that only exists between function calls. However, we *can* assert and prove that multiple-value functions return instances of multiple-value structures. This provides a compact way to specify the types of multiple-value functions, and to reason about individual values returned by multiple-value functions.

For example, imagine a function `(mv-fun x)` returning a prime number, an arbitrary object, and a string. We can use `defstructure` to define a template for the multiple-value list returned by `mv-fun`.

```
(defstructure mv-struct
  (n (:assert (primep n) :rewrite))
  o
  (s (:assert (stringp s) :rewrite))
  (:options :guards (:representation :mv)))
```

In fact, `mv-fun` could even be defined in terms of the *constructor* `mv-struct`,

```
(defun mv-fun (x)
  ...
  (mv-struct n o s))
```

or the *keyword-constructor* `make-mv-struct`.

```
(defun mv-fun (x)
  ...
  (make-mv-struct :n 7 :o 'x :s "foo"))
```

Assuming there is a proof, we can prove that `mv-fun` returns an instance of `mv-struct`.

```
(defthm type-of-mv-fun
  (mv-struct-p (mv-fun x)))
```

When `mv-fun` is used, however, the multiple values returned must be immediately bound by `mv-let` as in

```
(mv-let (n o s) (mv-fun x) . forms)
```

which macroexpands into

```
(let ((n (mv-nth 0 (mv-fun x)))
      (o (mv-nth 1 (mv-fun x)))
      (s (mv-nth 2 (mv-fun x))))
  . forms)
```

If we need to reason about the values returned by `mv-fun` we can't immediately use the type lemmas for `mv-struct` since they mention `mv-struct-n` and `mv-struct-s`, *not* `mv-nth`. To get around this problem, `defstructure` creates a theorem-writing macro (the *mv-intro-macro*) that can be used to rewrite `mv-nth` to the *reader* functions for a multiple-value structure. This macro, named *struct*`-intro` for a structure *struct*, takes one argument, a symbolic call of a multiple-value function, and creates a lemma that rewrites `mv-nth` to the structure *reader* functions. The *mv-intro-macro* must be invoked for every multiple-value function that returns instances of the structure. For example, the macro invocation

```
(mv-struct-intro (mv-fun x))
```

creates a lemma `mv-struct-mv-intro-mv-fun` that rewrites

```
(mv-nth 0 (mv-fun x)) ↦ (mv-struct-a (mv-fun x))
(mv-nth 1 (mv-fun x)) ↦ (mv-struct-o (mv-fun x))
(mv-nth 2 (mv-fun x)) ↦ (mv-struct-s (mv-fun x))
```

We can now reason about the values returned by `mv-fun` using the type lemmas proved about instances of `mv-struct`. This assumes, of course, that both `mv-fun` and `mv-nth` remain `DISABLE`d.[2]

[2]One should always `DISABLE mv-nth`. ACL2 has powerful heuristics for dealing with unexpanded occurrences of `mv-nth`.

To summarize, the multiple-value support offered by `defstructure` can be used as follows.

1. Define a structure with `(:representation :mv)` as a template for multiple-value functions. The structure may include assertions about the types and interrelations of the returned values of the functions.

2. Define multiple-value functions that return 'instances' of the multiple-value structure. The multiple-value functions can use `mv` directly, or can use the *constructor* or *keyword-constructor*.

3. Prove that the multiple-value functions return instances of the structure by proving that the functions satisfy the *predicate*.

4. Use the *mv-intro-macro* to generate lemmas mapping `mv-nth` to the *reader* functions for the structure. The *mv-intro-macro* must be invoked for each multiple-value function that returns an instance of the structure.

2.4 Recursive Structures

`defstructure` was originally developed to meet the need for large, deeply-nested hierarchical structures for a hardware specification problem, thus there is only minimal support for recursive structures. If the user's intent is to manipulate linked lists of structures *à la* Pascal or C, it is almost certainly better to use ACL2 lists of structure occurrences, rather than adding link slots to structure definitions. `defstructure` can be used to define trees of structures, however. `defstructure` provides a bare minimum of lemmas necessary to justify recursion on recursive structures. Figure 1 is a simple worked example demonstrating recursive structures.

```
(defstructure node
  value
  (left (:assert (or (node-p left) (null left))))
  (right (:assert (or (node-p right) (null right))))
  (:options :guards))

;;  defstructure won't work in mutual-recursion, so we have to define the
;;  tree recognizer separately, and reprove the desired lemmas about the node.

(defun tree-p (x)
  (declare (xargs :guard t))
  (or (node-p x) (null x)))

(defthm node-lemmas
  (implies
   (and (tree-p x)
        (not (null x)))
   (and (tree-p (node-left x))
        (tree-p (node-right x)))))

(in-theory (disable tree-p))

;;  Note the use of the weak-predicate as a fast guard check for the readers,
;; and more importantly to justify the recursion.  This is also a case where we
;; need a lemma before we can prove the guards.

(defun flatten-tree (tree)
  (declare (xargs :guard (tree-p tree)
                  :verify-guards nil))
  (cond
   ((null tree) nil)
   ((not (weak-node-p tree)) nil)
   (t (cons (node-value tree)
            (append (flatten-tree (node-left tree))
                    (flatten-tree (node-right tree)))))))

(defthm true-listp-flatten-tree
  (true-listp (flatten-tree tree)))

(verify-guards flatten-tree)
```

Figure 1: Simple example of a recursive structure.

3 Reference Guide

This section is a reference guide for `defstructure`. It begins in Section 3.1.1 with a formal syntax description of `defstructure`, followed in Section 3.2 with a description of the default behavior of `defstructure`. Section 3.3 documents the general structure and slot options. Finally, Section 3.4 provides more information on `defstructure` assertions. As mentioned earlier, perhaps the best way to understand how `defstructure` works is to examine the macroexpansion of a `defstructure` form. An example macroexpansion appears as Appendix B.

3.1 Syntax

3.1.1 Formal Syntax

```
defstructure name [documentation] {slot-and-options}* [option-list]

  option-list ::= (:OPTIONS [[options]])

  options ::= guards-option |
              verify-guards-option |
              slot-writers-option |
              inline-option
              conc-name-option |
              set-conc-name-option |
              keyword-constructor-option |
              keyword-updater-option |
              predicate-option |
              weak-predicate-option |
              force-option |
              representation-option |
              do-not-option |
              mv-intro-macro-option
              update-method-option |
              assertion-lemma-hints-option |
              predicate-guard-hints-option |
              prefix-option |
              {assert-option}*

  slot-and-options ::= slot-name | (slot-name [[slot-options]])

  slot-options ::= default-option |
                   read-only-option |
                   {assert-option}*

  default-option ::= :DEFAULT | (:DEFAULT) | (:DEFAULT slot-initform)

  read-only-option ::= :READ-ONLY

  assert-option ::= (:ASSERT assertion {assertion-rule-descriptor}*)

  assertion-rule-descriptor ::= rule-token |
                                (rule-token corollary [other-rule-forms])

  rule-token ::= NIL | :REWRITE | :LINEAR | :LINEAR-ALIAS |
                 :WELL-FOUNDED-RELATION | :BUILT-IN-CLAUSE |
```

```
              :COMPOUND-RECOGNIZER | :ELIM | :GENERALIZE | :META |
              :FORWARD-CHAINING | :EQUIVALENCE | :REFINEMENT |
              :CONGRUENCE | :TYPE-PRESCRIPTION | :DEFINITION | :INDUCTION |
              :TYPE-SET-INVERTER

guards-option ::= :GUARDS

verify-guards-option ::= :VERIFY-GUARDS | (:VERIFY-GUARDS) |
                         (:VERIFY-GUARDS T) | (:VERIFY-GUARDS NIL)

slot-writers-option ::= :SLOT-WRITERS

inline-option ::= :INLINE

conc-name-option ::= :CONC-NAME | (:CONC-NAME) | (:CONC-NAME conc-name)

set-conc-name-option ::= :SET-CONC-NAME | (:SET-CONC-NAME) |
                         (:SET-CONC-NAME set-conc-name)

keyword-constructor-option ::= :KEYWORD-CONSTRUCTOR |
                               (:KEYWORD-CONSTRUCTOR) |
                               (:KEYWORD-CONSTRUCTOR keyword-constructor)

keyword-updater-option ::= :KEYWORD-UPDATER | (:KEYWORD-UPDATER) |
                          (:KEYWORD-UPDATER keyword-updater)

predicate-option ::=  :PREDICATE | (:PREDICATE) | (:PREDICATE predicate)

weak-predicate-option ::=  :WEAK-PREDICATE | (:WEAK-PREDICATE) |
                           (:WEAK-PREDICATE weak-predicate)

force-option ::= :FORCE

do-not-option ::= (:DO-NOT [[do-not-options]])

do-not-options ::= :TAG | :READ-WRITE | :WRITE-WRITE

representation-option ::= :REPRESENTATION | (:REPRESENTATION) |
                          (:REPRESENTATION representation)

representation ::= :LIST | :MV | :DOTTED-LIST | :TREE | template

mv-intro-macro-option ::=  :MV-INTRO-MACRO |
                           (:MV-INTRO-MACRO) |
                           (:MV-INTRO-MACRO mv-intro-macro)

update-method-option ::= :UPDATE-METHOD | (:UPDATE-METHOD) |
                         (:UPDATE-METHOD update-method)

update-method ::= :HEURISTIC | :SET | :COPY

assertion-lemma-hints-option ::=
  :ASSERTION-LEMMA-HINTS | (:ASSERTION-LEMMA-HINTS) |
  (:ASSERTION-LEMMA-HINTS hints)
```

```
predicate-guard-hints-option ::=
  :PREDICATE-GUARD-HINTS | (:PREDICATE-GUARD-HINTS) |
  (:PREDICATE-GUARD-HINTS hints)

prefix-option ::= :PREFIX | (:PREFIX) | (:PREFIX prefix)
```

3.1.2 Arguments and Values

assertion – a *slots-assertion*.

corollary – a *slots-assertion*.

conc-name – a *string-designator*.

documentation – a string; not evaluated.

hints – an *acl2-hints*.

keyword-constructor – a symbol.

keyword-updater – a symbol.

name – a symbol.

mv-intro-macro – a symbol.

other-rule-forms – Some *acl2-rule-forms*.

predicate – a symbol.

prefix – a *string-designator*.

read-write-lemma – a symbol.

set-conc-name – a *string-designator*.

slot-initform – a form; not evaluated.

slot-name – a *valid-slot-name*.

tag – a symbol.

template – A *slots-template*.

weak-predicate – a symbol.

write-write-lemma – a symbol.

3.1.3 Definitions

acl2-hints – any form valid as the `:hints` argument of `defthm`. See the documentation for `HINTS` in the ACL2 documentation.

acl2-rule-forms – Any forms that would be valid in an ACL2 `:rule-classes` form, *except* for the rule class itself, or a `:corollary` and formula. See the documentation for the `defstructure` assertion theory in Section 3.4, and the ACL2 documentations for `RULE-CLASSES`.

slots-assertion – `defstructure` assertions are covered in Section 3.4.

slots-template – A `cons` tree whose flattened form (by `structures::flatten`) is a permutation of the list of slot names of the structure. See the documentation for the `:representation` option in Section 3.3.2.

string-designator – a character, string or symbol, it designates the string obtained by

```
(string string-designator)
```

except that by convention the symbol `nil` designates the empty string.

valid-slot-name – A *valid-slot-name* is a symbol valid for use as a formal parameter of a function. This is any symbol not in the `"keyword"` package, neither `t` nor `nil`, neither beginning nor ending with '`*`', and not beginning with '`&`'. In addition, no *slot-name* may be the same as *name*, and all *slot-name* must have unique print names, i.e., it is illegal to duplicate slot names, and it is illegal to use symbols from different packages that have the same `symbol-name`.

3.2 Default Behavior

The basic syntax of `defstructure` is

```
(defstructure name [documentation] slot-1 ... slot-n)
```

The following describes the default behavior of `defstructure` for simple invocations as above. Section 3.3 details the effects of the structure and slot options for `defstructure`.

`defstructure` defines a *constructor* function *name* of n arguments, where $n \geq 0$. If *documentation* is supplied, it is attached to the *constructor* as a documentation string. The *constructor* creates instances of the structure, where the initial slot contents are specified in order.

```
(name arg-1 ... arg-n)
```

An instance of the structure is an ACL2 list of length $n + 1$, whose *car* is the symbol *name*, and whose `nth` element is the n^{th} slot.

`defstructure` defines *reader* functions for the slots named *name-slot-1* ... *name-slot-n*. A *reader* is a function of one argument that takes an instance of the structure and returns the value stored in the given slot of the instance.

`defstructure` defines a *keyword-constructor* named `make-`*name*. The *keyword-constructor* is a macro that generates a call of the *constructor*, where initial slot contents are specified in any order by a keyword argument list.

```
(keyword-constructor slot-keyword-i form-i ... slot-keyword-j form-j)
```

Each *slot-keyword-i* must be a keyword whose name corresponds to the name of a structure slot. The keywords are not evaluated; the forms are evaluated. If a slot is not initialized in this way, it is initialized by the *slot-initform* given as the argument of the `:default` slot option. If provided, the *slot-initform* will be evaluated in the context in which the macro is expanded. If no *slot-initform* is supplied for a slot and no form is supplied for the slot in the call of the *keyword-constructor*, then the *keyword-constructor* will signal an error. The *keyword-constructor* expands into the obvious call of the *constructor*.

`defstructure` defines a *keyword-updater* named `update-`*name*. The *keyword-updater* is a macro that updates 0 or more slots of an instance of a structure as specified by a keyword argument list.

```
(keyword-updater instance slot-keyword-i form-i ... slot-keyword-j form-j)
```

By default, the *keyword-updater* generates a call of the *constructor* in which any slots from the original instance of the structure which have not been assigned new values in the keyword argument list are copied from the original instance. This default can be changed with the `:update-method` option; see Section 3.3.3.

`defstructure` defines a *weak-predicate* for the structure named `weak-`*name*`-p`. The *weak-predicate* is a Boolean function of one argument that recognizes list structures that have the proper 'shape' to be an occurrence of the structure. By default, this means a list of the proper length whose *car* is *name*. The *constructor* always creates an ACL2 object that satisfies the *weak-predicate*. The *weak-predicate* is introduced for use as a weak guard on the *readers*, and normally the user seldom writes explicit occurrences of the *weak-predicate* in definitions or theorems. Note that since a structure is simply a list, there is no way to know if the instance was created by the *constructor*, or simply by `cons`. This is an advantage in that `defstructure` can be used to recognize and manipulate list structures created elsewhere, as for example multiple value lists (see Section 2.3). This is a potential disadvantage in that certain types of errors in specifications may be overlooked.

`defstructure` defines a *predicate* for the structure named *name*`-p`. The *predicate* is a Boolean function of one argument that recognizes instances of the structure. The *predicate* extends the *weak-predicate* by including all *assertion* specified for the structure. The *predicate* and *weak-predicate* are identical for structures without type assertions. Definitions and theorems involving the structure will almost always use the *predicate* as the recognizer for instances of the structure.

`defstructure` also produces a number of lemmas about the *constructor*, *readers*, *weak-predicate* and *predicate*. The following is a very brief description of the lemmas. For complete information on their forms and `:rule-classes` the reader is advised to expand an occurrence of `defstructure` using `trans1` as described earlier.

The lemma `defs-acl2-count-`*name* justifies recursion on any slot of the structure. Recursive structures are briefly discussed in Section 2.4.

The lemma `defs-weak-`*name*`-p-`*name* shows that calls of the *constructor* always satisfy the *weak-predicate*.

The lemma `defs-`*name*`-p-includes-weak-`*name*`-p` shows that the *predicate* implies the *weak-predicate*.

The lemma `defs-`*name*`-p-`*name* rewrites instances of the predicate applied to the constructor. For untyped structures, the result is simply `t`. For typed structures, the result is the collection of all of the *assertion* for the structure.

The lemma `defs-read-`*name* simplifies applications of the *readers* to explicit symbolic occurrences of the *constructor*.

The lemma `defs-`*name*`-lift-if` 'lifts' `if` through calls of the *reader* functions.. In certain cases this lemma can produce significant performance improvements when reasoning about complex specifications. A more detailed justification appears in the source code (search for `LIFT-IF-LEMMA`).

Finally, the lemma `defs-eliminate-`*name* is an `:elim` lemma for the *constructor*. For more information on `:elim` rules refer to the `ELIM` section of the ACL2 documentation.

`defstructure` also creates two ACL2 theories. The first is named `defs-`*name*`-definition-theory`. This theory consists of all of the definition runes associated with the *constructor*, *readers*, *predicate* and *weak-predicate*. This theory is `DISABLE`d by default. Users are strongly advised never to `ENABLE` this theory except under very special circumstances (which are beyond the scope of this note). The other theory is named `defs-`*name*`-lemma-theory`. This theory includes every lemma generated by `defstructure`. This theory is `ENABLE`d by default, and should normally remain `ENABLE`d.

3.3 Standard Options

This section details the standard options for `defstructure`. The `:assert` option, which is used to define typed structures, is discussed separately in Section 3.4. Options to `defstructure` are specified by means of a special keyword options list that appears as the last form in the body of the `defstructure`. The *car* of this list is the keyword `:options`. Option values are specified by keywords, or by lists whose *car* is a keyword. A few options may also be attached to the slots as detailed in Section 3.3.5. No part of any of the option specifications are evaluated by `defstructure`.

3.3.1 Name Options

`defstructure` is an ACL2 macro that generates numerous functions and lemmas. In general the user has complete control over the function names, and a small degree of control over generated lemma names. By default, `defstructure` generates names based on standard templates that include the structure name. Like all ACL2 macros, however, `defstructure` can not access the ACL2 *state*, thus `defstructure` can not guarantee that the generated names are unique, and therefore it is possible that a `defstructure` macro expansion will fail due to name collisions. All function, lemma and theory names generated by `defstructure` will be interned in the package of the structure *name*, the first argument of `defstructure`.

`:conc-name`

This option provides for automatic prefixing of names of *reader* functions. The default behavior is to begin the names of all the *reader* functions with the name of the structure followed by a hyphen.

`:conc-name` supplies an alternate prefix to be used. If a hyphen is to be used as a separator, it must be supplied as part of the prefix. If `:conc-name` is `nil` or no argument is supplied, then no prefix is used; then the names of the *reader* functions are the same as the slot names. If a non-`nil` prefix is given, the name of the *reader* functions are constructed by concatenating that prefix and the name of the slot, and interning the resulting symbol in the package of *name*.

Note that no matter what is supplied for `:conc-name`, keywords that match the slot names with no prefix attached are used by the *keyword-constructor* and *keyword-updater* macros.

`:set-conc-name`

This options provides for automatic prefixing of names of *writer* functions. *Writer* functions are only created when the `:slot-writers` option is selected (see Section 3.3.3). The default behavior is to begin the names of all the *writer* functions for structure *name* with `set-`*name*`-`.

`:set-conc-name` supplies an alternate prefix to be used. If a hyphen is to be used as a separator, it must be supplied as part of the prefix. If `:set-conc-name` is `nil` or no argument is supplied, then no prefix is used; then the names of the *writer* functions are the same as the slot names. (This is probably a *very* bad idea.) If a non-`nil` prefix is given, the name of the *writer* function for each slot is constructed by concatenating that prefix and the name of the slot, and interning the resulting symbol in the package of *name*.

`:keyword-constructor`

This option take one argument, which specifies the name of the *keyword-constructor* macro. If the argument is not supplied or if the option itself is not supplied, the name of the *keyword-constructor* will be `make-`*name*. If the argument is provided and is `nil`, no *keyword-constructor* is defined.

`:keyword-updater`

This option take one argument, which specifies the name of the *keyword-updater* macro. If the argument is not supplied or if the option itself is not supplied, the name of the *keyword-updater* will be `update-`*name*. If the argument is provided and is `nil`, no *keyword-updater* is defined.

`:weak-predicate`

This option take one argument, which specifies the name of the *weak-predicate* for the structure. If the argument is not supplied or if the option itself is not supplied, the name of the *weak-predicate* is made by concatenating `"WEAK-"`, the name of the structure, and `"-P"`, and interning the resulting symbol in the package of *name*. If the argument is provided and is `nil`, an error will be signalled.

`:predicate`

This option take one argument, which specifies the name of the full-blown *predicate* for the structure. If the argument is not supplied or if the option itself is not supplied, the name of the *predicate* is made by concatenating the name of the structure with `"-P"`, and interning the resulting symbol in the package of *name*. If the argument is provided and is `nil`, an error will be signalled.

`:prefix`

This prefix will be prepended to every `defthm` and `deftheory` event generated by `defstructure`. This feature helps avoid name collisions with pre-existing function or lemma names. The names of all `defthm` and `deftheory` events are interned in the package of the structure *name*.

This option takes one argument, which must be a character, string, or symbol. The argument is not evaluated. If the argument is not supplied or if the option itself is not supplied it is equivalent to specifying the default prefix `"DEFS-"`. If the argument is `nil`, it is equivalent to specifying the empty string.

3.3.2 Representation and Performance Options

A `defstructure` structure is an ACL2 list. By default, a structure is a proper list whose *car* is the structure name and whose `nth` element is the value of the n^{th} slot. `defstructure` provides several options for changing this default representation, however. This allows `defstructure` to be used as a 'template' for list structures created elsewhere, for example for multiple-value lists. The representation of a structure also has performance implications as discussed below. For this reason the representation and performance options are included together in this section. The `:slot-writers` option also has potential implications for performance as discussed in Section 3.3.3.

Run-time performance of ACL2 executions may be strongly influenced by the number of *cons* operations performed, due to the overhead involved in allocating and garbage collecting large numbers of *cons* cells. Reducing the sizes of structure may result in better performance. There are two easy ways to decrease the size of a structure. First, the structure name can be removed from the structure instances with the `(:do-not :tag)` option. Second, structure instances can be specified to be improper lists by means of the `(:representation :dotted-list)` option. These options may make explicit structure instances less readable, but result in a savings are two *cons* cells per structure instance. `defstructure` also provides an `:inline` option for 'inlining' the *reader* and *writer* functions, thereby reducing subroutine calling overhead.

`:do-not`

The arguments of the `:do-not` option are all keywords. One of the recognized keywords is `:tag`. Other `:do-not` options are discussed in Section 3.3.3.

By default, the *car* of each structure created by `defstructure` is the structure *name*. Specifying `(:do-not :tag)` eliminates this name tag from the structure.

`:representation`

This option takes one argument, which is either `:list`, `:dotted-list`, `:tree`, `:mv`, or a *template*. If the argument is not supplied or if the option itself is not supplied the default value of `:list` is used.

By default structures are true lists, tagged with the structure name. Thus for the structure

```
(defstructure struct a b c)
```

the *constructor* is defined as

(struct *a b c*) ≡ (list 'struct *a b c*)

If struct were defined as

```
(defstructure struct a b c
  (:options (:representation :dotted-list)))
```

then the *constructor* is defined as

(list* 'struct *a b c*)

that is, the internal form of the structure is a 'dotted list'. By dotting the last *cons* one may be able to save a small but significant amount of storage overhead over the course of a long execution of a specification, with minimal impact on the readability of the printed representations.

If struct were defined as

```
(defstructure struct a b c d
  (:options (:representation :tree)))
```

then the constructor would be defined as

(struct *a b c d*) ≡ (cons 'struct (cons (cons *a b*) (cons *c d*)))

The internal representation of the body of a structure defined with the :tree representation is a balanced *cons* tree. These trees have the nice property that reading and writing will occur in $O(\log n)$ *car*/*cdr* accesses or cons operations, respectively, where n is the number of slots. The printed representations of these structures may be quite difficult to read, however, but in certain instances they may provide the most efficient representation. We suggest that the user study all of the issues carefully before using this representation.

Similarly, one may specify a special-purpose tree by using a *template* for the representation. A *template* is any *cons* tree with the property that when it is 'flattened' by structures::flatten it yields a list which is a permutation of the list of slot-names of the structure. One simple example of this representation would be to define a *cons* cell whose *car* and *cdr* were asserted to be integerp.

```
(defstructure icons
  (icar (:assert (integerp icar)))
  (icdr (:assert (integerp icdr)))
  (:options (:do-not :tag) (:representation (icar . icdr))))
```

The option (:representation :mv) specifies that the structure is to be used as a template for coding and reasoning about functions that return multiple values. Please see Section 2.3 for more complete information.

The default symbolic representation of a structure is a symbolic occurrence of the *constructor* function applied to symbolic arguments. The theory created by defstructure DISABLEs every lemma about the :definition of the *constructor* so that the user will always see this symbolic representation in prover output. This symbolic representation is independent of the :representation option supplied for the structure, subject to the caveat that multiple-value structures never appear as such in prover output. The user is strongly advised *not* to ENABLE any :definition runes created by defstructure. The :representation option *will* change the printed representation of concrete instances of the structure.

:inline

If the :inline option is specified, then the defstructure structures will be optimized for run-time efficiency only. The *constructor*, *reader* and *writer* functions (if defined) will be defined as *macros* rather than functions, and none of the lemmas or theories that `defstructure` normally generates will be generated. The *keyword-updater* macro will also expand into the most efficient code for updating the specified slots. The usefulness of this option depends on how the underlying Common Lisp system compiles Lisp functions, and on the subroutine calling conventions of the underlying hardware.

3.3.3 Writable Structure Options

By default, `defstructure` structures are 'read-only' in the sense that `defstructure` does not provide a way to update a single slot of a structure. Instead, `defstructure` creates modified instances of a structure by creating a completely new instance of the structure, and copying in any unmodified slots. This default can be changed by the `:slot-writers` option. If `:slot-writers` is specified, then `defstructure` creates a set of *writer* functions for the structure. A *writer* function is a function that takes a structure instance and a new slot value and returns a modified instance of the structure. For example,

```
(defstructure struct a b c (:options :slot-writers))
```

creates a *writer* function for each slot.

```
(set-struct-a new-a struct)
(set-struct-b new-b struct)
(set-struct-c new-c struct)
```

The *writer* functions allow for *cons*-efficient updating of a single slot of a structure, as they create only the minimum number of new *cons* cells necessary. For the above example,

```
(set-struct-a new-a struct)
≡
(cons (car struct)  ; The structure tag
      (cons new-a (cddr struct)))
```

thus the 'tail' of the original instance is reused in the new-instance. This is a 'safe' definition in ACL2 since ACL2 does not provide any destructive operations. If `:slot-writers` is selected then the *keyword-updater* macro may expand into calls of the *writer* functions as described below under the `:update-method` option.

Despite the potential performance advantages, the *writer* functions cause several difficulties for symbolic reasoning. First, to reason about the contents of structure slots after the application of a *writer* function, it is necessary to have *read-write* lemmas of the form

```
(struct-slot (set-struct-slot new-slot struct)) ↦ new-slot
```

and

```
(struct-slotx (set-struct-sloty new-sloty struct)) ↦ (struct-slotx struct)
```

for all slots $slotx \neq sloty$, i.e., $O(n^2)$ lemmas. By default `defstructure` also creates $O(n^2)$ *write-write* lemmas. These lemmas normalize multiple nested writes of a structure by 'pushing' symbolic writes of 'deep' slots through symbolic writes of 'shallow' slots, and reducing redundant writes of the same slot to a single write, e.g.,

```
(set-struct-c new-c (set-struct-a new-a struct))
  ↦ (set-struct-a new-a (set-struct-c new-c struct))
```

and

```
(set-struct-a new-a (set-struct-a last-new-a struct))
  ↦ (set-struct-a new-a struct)
```

When `:slot-writers` is specified other specialized lemmas are also created to aid reasoning about structures with type assertions.

In summary, the `:slot-writers` option involves significant extra overhead, both in the number of lemmas generated and in the symbolic reasoning process. In general users are discouraged from using this option. However, if the intent is to create a specification simply for execution, then the `:slot-writers` option has the potential of creating more efficient specifications. Note that this will only be the case if the specification typically updates structure instances by modifying only one or two slots of a structure.

`:slot-writers`

If specified, then *writer* functions will be generated as described above. *Writer* functions are never generated for structures with `(:representation :mv)`.

`:do-not`

The arguments of the `:do-not` option are all keywords. Two of the recognized keywords are `:read-write` and `:write-write`. Other `:do-not` options are discussed in Section 3.3.2.

If `(:do-not :read-write)` is selected, then the *read-write* lemmas described above will not be generated. If `(:do-not :write-write)` is selected, then the *write-write* lemmas described above will not be generated.

`:update-method`

This option take one argument, which specifies the expansion of the *keyword-updater* macro. By default, if the argument is not supplied or if the option itself is not supplied, the update method will be `:copy`. If the `:slot-writers` option was selected, however, then the default update method is `:heuristic`. The other legal option when `:slot-writers` is selected is `:set`.

The update method determines how instances of the *keyword-updater* macro will expand. The form of this expansion affects both the execution performance and the course of symbolic reasoning in the ACL2 prover.

Update method `:copy` defines the *keyword-updater* macro to create a new instance of the structure by copying the contents of those slots that were not mentioned in the *keyword-updater* macro call. This is the usual default.

The other two options are only legal when `:slot-writers` is specified. Update method `:set` defines the *keyword-updater* macro to expand into a nested set of *writer* function calls. Update method `:heuristic` (the default) defines the *keyword-updater* macro to expand into the 'cheaper' of the `:copy` form or the `:set` form, where the cost is measured simply by the number of new *cons* cells required to produce the new instance of the structure. Thus the macro expansion will differ on a case-by-case basis. In the case of a tie, the `:set` form is defined to be cheaper, thus updating a single slot will always expand into the `:set` form.

3.3.4 Miscellaneous Options

`:guards`

By default, `defstructure` does not declare guards for the functions it generates. If `:guards` is specified, then the functions are defined with guards declared as follows:

Function	Guard
predicate	`t`
weak-predicate	`t`
readers	*weak-predicate*
writers	*weak-predicate*

Note that for the *writers* only the structure argument is guarded.

`:verify-guards`

If specified as `(:verify-guards nil)`, then the `xargs` declaration

```
(declare (xargs :verify-guards nil))
```

will appear with all guarded functions. If specified as `(:verify-guards t)`, then the `xargs` declaration

```
(declare (xargs :verify-guards t))
```

will appear with all guarded functions. If missing, or specified as `:verify-guards` or `(:verify-guards)`, then no `:verify-guards` declarations will appear with guarded functions.

`:force`

If the `:force` option is specified, then lemmas generated by `defstructure` will `force` their hypotheses whenever those hypotheses correspond to guards on the functions appearing in the left-hand-side of the rule. A table listing the guards for the functions generated by `defstructure` appears just above with the documentation for the `:guards` option. For more information on `force` refer to the ACL2 documentation for `FORCE`.

`:assertion-lemma-hints`

This option takes one argument, which must be a valid `:hints` argument to `defthm` (see the ACL2 documentation for `HINTS`). The argument is not evaluated. If the argument is not supplied or if the option itself is not supplied or if the argument is `nil`, it is equivalent to offering no hints.

The *hints* are given to the proof of the *assertion-lemma*, which is described in Section 3.4.

`:predicate-guard-hints`

This option takes one argument, which must be a valid `:hints` argument to `defthm` (see the ACL2 documentation for `HINTS`). The argument is not evaluated. If the argument is not supplied or if the option itself is not supplied or if the argument is `nil`, it is equivalent to offering no hints.

If the `:guards` option is selected then the *hints* are given to the guard verification proof of the predicate.

3.3.5 Slot Options

Three options may be specified for a slot: `:default`, `:read-only` and `:assert`. The `:assert` option is discussed later in Section 3.4

`:default`

The `:default` option is specified as `(:default` *`slot-initform`*`)`. The *slot-initform* is the value that will be stored into the slot by default when an instance of the structure is created with the *keyword-constructor* macro. If no *slot-initform* is specified for a slot, then it is an error to create an instance of the structure without specifying a value for the slot. The *slot-initform* is not evaluated by `defstructure`. The *slot-initform* is evaluated in the context of each occurrence of the *keyword-constructor* macro.

`:read-only`

By default, all structure slots are read-only, i.e., no *writer* functions are defined. The `:read-only` option is only meaningful for structures that select the global `:slot-writers` option. Specifying *read-only* for a slot suppresses the generation of the *writer* function for that slot. For more information on *writer* functions, see Section 3.3.3.

3.4 `defstructure` Assertions

One of the most important functions for `defstructure` in the context of guarded specifications is the creation of a theory of assertions about the slots of the structure. These assertions specify the user's intention that the contents of the slots should always satisfy a set of predicates. These predicates may simply refer to the individual slot values, or may involve arbitrary relationships among multiple slots.

We say that the assertions "specify the user's intention" because the *constructor* function is *not* a guarded function. By definition it is legal to store any value into any slot of any structure. Instead of guarding the *constructor*, assertions are recorded in the *predicate*, and in lemmas about the *reader* and *writer* functions for the structure. If a value is stored that doesn't satisfy the assertions about a slot, either explicitly with the *keyword-updater* or *writer* function, or implicitly as a default value, this simply create an instance of the structure that does not satisfy the *predicate*, and hence none of the type rules for the structure will apply to that instance. Again, invalidating the guard for a single slot will invalidate the *predicate* for the entire structure.

Assertions may be attached to individual slots or to the structure as a whole. Assertions are uniformly specified by options of the form:

```
assert-option ::= (:ASSERT assertion {assertion-rule-descriptor}*)

assertion-rule-descriptor ::= rule-token |
                              (rule-token corollary [other-rule-forms])
```

The *assertion* is an ACL2 form whose *free variables* (as defined below) are a subset of the slot names of the structure. The *assertion-rule-descriptor* is discussed later.

Since it is impossible for a user-defined macro like `defstructure` to completely analyze the syntax of terms, yet it is necessary for `defstructure` to determine which slots the assertion references, we define a very restricted form of terms that are allowed as an *assertion*. These terms are formally defined by `(structures::assertion-termp` *`assertion`*`)`. Informally, an *assertion* is either an atom, a quoted constant, or a true list whose *cdr* is a list of assertion terms, and whose *car* is either a symbol or a `lambda` function. The free variables of the *assertion* are computed in the usual way, with the special note that the free-variable computation not descend inside any `lambda` function.

For example, assume a simple structure

```
(defstructure struct a b c (:options ...)
```

Then the *assert-option*

```
(:assert a)
(:assert (integerp a))
(:assert (and (integerp a) (>= a 0)))
```

assert respectively that slot *a* is non-`nil`, is an integer, or is a natural number.

Assertions must be written as `lambda` functions if the *assertion* uses macros that have special syntax. For example,

```
(:assert (cond ((eq a 'string) (stringp b)) (t (integerp b))))
```

is invalid because the `cond` macro has a special syntax (the `cond` clauses have a non-atomic *car* that is not a `lambda` function). This is easily remedied, however, by

```
(:assert ((lambda (a b)
            (cond ((eq a 'string) (stringp b)) (t (integerp b)))
           a b)))
```

which is a legal assertion for `defstructure`.

The predicate for the structure is a conjunction of the *weak-predicate*, all of the *assertion* specified for the slots and the structure as a whole, and the atom `t`, which is always added to guarantee that the *predicate* is a Boolean function. This may be sufficient if the intention of supplying slot assertions is simply to document the structure. For guarded specifications, however, it is usually necessary to capture the assertions in the form of lemmas about the *reader* and *writer* functions. No lemmas are generated for an *assertion* unless the *assertion* also includes one or more *assertion-rule-descriptor*. The *assertion-rule-descriptor* can be thought of as a shorthand description of a `defthm` rule-class, where the default `:corollary` of the rule-class is the *assertion*.

The simplest form of an *assertion-rule-descriptor* is a *rule-token*, i.e., one of the keywords recognized as a `defthm` rule-class. For example,

```
(:assert (integerp a) :type-prescription)
```

will add the assertion that the *a* slot of the structure is an integer, and also will record this fact as a `:type-prescription` lemma about the *reader* function for slot *a*. Assuming that the structure was defined as above, the complete rule-class for this assertion is:

```
(:type-prescription
 :corollary
 (implies (struct-p struct)
          (integerp (struct-a struct))))
```

An *assertion-rule-descriptor* may also include a *corollary*, which must adhere to the same syntactic rules as an *assertion*. In this case, the new *corollary* does not become a conjunct of the *predicate*, but instead is only used as a corollary formula of the generated rule. This form of the *assertion-rule-descriptor* is useful when a single assertion implies different kinds of information. For example:

```
(:assert (primep a)
         :rewrite (:type-prescription (and (integerp a) (> a 0))))
```

will generate the `:rule-classes`

```
(:rewrite
 :corollary
 (implies (struct-p struct)
          (primep (struct-a struct))))

(:type-prescription
 :corollary
 (implies (struct-p struct)
          (and (integerp (struct-a struct))
               (> (struct-a struct) 0))))
```

thus recording the fact that slot *a* is `primep`, as well as the fact that it is an integer greater than 0.

The *other-rule-forms* are any other forms that are legal in an ACL2 rule-class (see the ACL2 documentation for `RULE-CLASSES`). For example, hints can be passed to the proof of the corollary formula.

```
(:assert (primep a)
         :rewrite
         (:type-prescription
          (and (integerp a) (> a 0))
          :hints (("Goal" :in-theory (enable primep)))))
```

The above will generate the `:rule-classes`

```
(:rewrite
 :corollary
 (implies (struct-p struct)
          (primep (struct-a struct))))

(:type-prescription
 :corollary
 (implies (struct-p struct)
          (and (integerp (struct-a struct))
               (> (struct-a struct) 0)))
 :hints
 (("Goal" :in-theory (enable primep))))
```

An *assert-option* may be associated with a slot, or with the structure as a whole. The choice of where to place the *assert-option* is completely up to the user, and usually does not affect the semantics of the option. As a consistency check, however, `defstructure` does check that every *assertion* appearing with a slot at least mentions the associated slot. Thus the following are all equivalent.

```
(defstructure struct
  a b c
  (:options (:assert (integerp a))
            (:assert (integerp b))))

(defstructure struct
  a b c
  (:options (:assert (and (integerp a)
                          (integerp b)))))
```

```
(defstructure struct
  (a (:assert (integerp a)))
  (b (:assert (integerp b)))
  c)
```

The only complication occurs when the assertions include guarded functions, and the `:guards` option is selected. In this case the user must be careful that the predicate includes (or provably implies) all of the guards of the guarded assertions. It must also be the case that the body of the predicate asserts (or provably implies) the guards before the occurrence of the guarded function. For example, the following `defstructure` form

```
(defstructure struct
  (x (:assert (eqlablep x)))
  (l (:assert (eqlable-listp l)))
  (:options :guards (:assert (member x l))))
```

generates the predicate

```
(struct-p struct)
≡
(and (eqlablep (struct-x struct))
     (eqlable-list (struct-l struct))
     (member (struct-x struct) (struct-l struct))
     t)
```

whose guards can be verified. Note that the *assertion* are collected in the *predicate* in slot order, followed by any global *assertion* in their specification order. On the other hand,

```
(defstructure struct
  (x (:assert (and (eqlablep x)
                   (member x l))))
  (l (:assert (eqlable-listp l)))
  (:options :guards))
```

generates the predicate

```
(struct-p struct)
≡
(and (and (eqlablep (struct-x struct))
          (member (struct-x struct) (struct-l struct)))
     (eqlable-listp (struct-l struct))
     t)
```

whose guards can not be verified, because the guard for `member` has not been satisfied before the occurrence of `member`.

There may be cases where the guards of the *predicate* are difficult to verify. `defstructure` provides the `:predicate-guard-hints` option as a way to pass `:hints` to the `verify-guards` proof for the *predicate*. The `verify-guards` proof for the *predicate* is carried out in the theory currently in force when the `defstructure` form appears.

`defstructure` will create a lemma named (by default) `defs-`*name*`-assertions` whenever the *predicate* is defined and the user has specified at least one *assertion-rule-descriptor*. The lemma itself is a simple implication that the *predicate* implies its definition; the `:rule-classes` of the lemma are generated from the *assertion-rule-descriptor* as defined above. Most lemmas generated by `defstructure` have easy structural proofs, and are proved in a special 'minimal' theory set up by `defstructure`. Unlike the other lemmas generated by `defstructure`, there is no guarantee that the *corollary* forms of the assertions will prove. For example, a user can easily specify an unsatisfiable corollary. The proof attempt for this lemma will be made in the theory that prevails when the `defstructure` form is submitted. The `:assertion-lemma-hints` option to `defstructure` may be used to specify hints for this lemma.

A Examples

This appendix contains example instances of `defstructure` from ACL2 specifications produced by Computational Logic, Inc. The first three examples are from an ACL2 translation of Yuan Yu's Nqthm specification of the MC68020 microprocessor. The first example is the global 'machine state' of the processor model. Note that type information is often stored as multiple rules, and note that the `:conc-name` option is used to make the *reader* function names more compact.

```
(defstructure mc-state
  "The state of the MC68020."
  (status (:assert (symbolp status) :type-prescription))
  (rfile  (:assert (rfilep rfile)   :rewrite))
  (pc     (:assert (longword-p pc)
                   :rewrite
                   (:type-prescription (naturalp pc))))
  (ccr    (:assert (ccr-p ccr)
                   :rewrite
                   (:type-prescription (naturalp ccr))))
  (mem    (:assert (memoryp mem)     :rewrite))

  (:options (:conc-name mc-)))
```

This is an effective address abstraction. The `loc` field is a symbol whose value determines the type of the `addr` field. Note the use of the `lambda` function to express this complex relationship.

```
(defstructure s&addr
  "An MC68020 effective address abstraction."
  (s     (:assert (mc-state-p s) :rewrite))
  (loc   (:assert (symbolp loc)  :type-prescription))
  (addr  (:assert ((lambda (loc addr)
                     (case loc
                       ((d a) (rn-numberp addr))
                       ((m i) (longword-p addr))
                       (otherwise (null addr))))
                   loc addr)
                  (:rewrite
                   (implies
                    (or (equal loc 'd) (equal loc 'a))
                    (rn-numberp addr)))
                  (:rewrite
                   (implies
                    (or (equal loc 'm) (equal loc 'i))
                    (longword-p addr))))))
```

Finally, this is an abstraction of a value computed by the ALU and the condition-code flags. The `defstructure` options make this nothing more than a typed *cons* cell.

```
(defstructure v&cvznx
  "An MC68020 value abstraction."
  (v     (:assert (longword-p v)
                  :rewrite
                  (:type-prescription (naturalp v))))
  (cvznx (:assert (ccr-p cvznx)
                  :rewrite
                  (:type-prescription (naturalp cvznx))))

  (:options (:representation (v . cvznx)) (:do-not :tag)))
```

The following examples are from the ACL2 specification of the Motorola CAP DSP, which included 76 occurrences of the `defstructure` macro. The first example specifies a `:conc-name` of `||` (the atom whose print name is the empty string) to suppress the usual default. The names of the *reader* functions for this structure are simply the slot names of the structure.

```
(defstructure cap-outputs
  ":doc-section cap-structures
  Output bus and pin values for the CAP chip.
  ...  documentation deleted  ...
  "
  (ta_bar        (:assert (@bitp ta_bar) :rewrite))
  (low_data-out  (:assert (@bus-word-p low_data-out) :rewrite))
  (high_data-out (:assert (@bus-word-p high_data-out) :rewrite))
  (hr_bar        (:assert (@bitp hr_bar) :rewrite))

  (:options :guards (:conc-name ||)
            #+speed (:representation :dotted-list)
            #+speed (:do-not :tag)
            #+inline-structures :inline))
```

The CAP specification included `defstructure` options for increasing run-time performance. We used the Common Lisp 'features' mechanism to include the representation and performance options when we were compiling the specification for use as a fast architectural simulator. On one typical example we found that the execution time for the 'fast' specification (with the `#+speed` and `#+inline-structures` features enabled) was approximately 84% of the execution time of the 'slow' specification.[3]

The CAP specification included 32 multiple-value structure definitions. The way that multiple-value structures are used is discussed in more detail in Section 2.3. Below is a typical example, the specification of the XY address generation units. First, `defstructure` is used to define a multiple-value template.

```
(defstructure xy-mem$state-mv
  (xwen/ywen (:assert (complex-bit-p xwen/ywen) :rewrite))
  (xy-mem    (:assert (xy-mem-p xy-mem) :rewrite))
  (:options (:representation :mv)))
```

[3]These results were measured on a 150 MHz Cyrix 6x86 with 96Mb of RAM running Linux 2.0, using ACL2 2.0 on GCL Version 2.2.2.

Next appears the specification of the multiple-value function. In this case the function was defined using mv directly.

```
(defun xy-mem$state (agu-mode s/d command xy-mem zx/zy)
  ...  definition deleted  ...
        (mv (complex-bit xwen ywen)
            (update-xy-mem
             xy-mem
             :x new-x-side
             :y new-y-side)))
```

Then, a proof that `xy-mem$state` returns values satisfying `xy-mem$state-mv-p`.

```
(defthm type-of-xy-mem$state
  (implies
   (xy-mem-p xy-mem)
   (xy-mem$state-mv-p (xy-mem$state agu-mode s/d command xy-mem zx/zy))))
```

Finally, the *mv-intro-macro* is used to generate a lemma rewriting `mv-nth` applied to `xy-mem$state`.

```
(xy-mem$state-mv-intro (xy-mem$state agu-mode s/d command xy-mem zx/zy))
```

B Macro Expansion

```
(defstructure struct
  (n (:assert (primep n)
              :rewrite
              (:type-prescription (and (integerp n)
                                       (< 0 n)))))
  o
  (s (:assert (stringp s) :rewrite))
  (:options :guards))
```

Following is the `:trans1` macro expansion of the above `defstructure` form.

```
(PROGN
  (STRUCTURES::CAPSULE
   "
;  We define the structure and all of the events (except the assertion theory)
;  in the absolute minimum theory possible in order to expedite the proofs
;  and guarantee that they will always work.  If you ever find a case where
;  one of these proof fails (except due to user syntax errors) please
;  report it as a bug in DEFSTRUCTURE.
 "
   (LOCAL (IN-THEORY (THEORY 'STRUCTURES::MINIMAL-THEORY-FOR-DEFSTRUCTURE)))
   "
;  The constructor is defined as a function that accepts every slot.
 "
   (DEFUN STRUCT (N O S)
          (DECLARE (XARGS :GUARD T))
          (LET ((STRUCT 'STRUCT))
               (CONS STRUCT (CONS N (CONS O (CONS S NIL))))))
```

```
   "
;  This lemma justifies recursion on any slot of the structure.  It is
;  unlikely to be used unless the structure is itself recursive.
  "
   (DEFTHM DEFS-ACL2-COUNT-STRUCT
           (EQUAL (ACL2-COUNT (STRUCT N O S))
                  (+ 4 (ACL2-COUNT N)
                     (ACL2-COUNT O)
                     (ACL2-COUNT S))))
   "
;  This predicate defines the `structure' of the structure, and is used as a
;  weak guard on the readers and writers (if defined).
 "
   (DEFUN WEAK-STRUCT-P (STRUCT)
          (DECLARE (XARGS :GUARD T))
          (AND (CONSP STRUCT)
               (CONSP (CDR STRUCT))
               (CONSP (CDR (CDR STRUCT)))
               (CONSP (CDR (CDR (CDR STRUCT))))
               (NULL (CDR (CDR (CDR (CDR STRUCT)))))
               (EQ (CAR STRUCT) 'STRUCT)))
   "
;  The weak-predicate is satisfied by any explicit reference of the
;  constructor.  We also store this information as a :BUILT-IN-CLAUSE
 "
   (DEFTHM DEFS-WEAK-STRUCT-P-STRUCT
           (EQUAL (WEAK-STRUCT-P (STRUCT N O S)) T)
           :RULE-CLASSES
           ((:REWRITE)
            (:BUILT-IN-CLAUSE :COROLLARY
                              (WEAK-STRUCT-P (STRUCT N O S)))))
   "
;  These are the `readers' for the structure.
 "
   (DEFUN STRUCT-N (STRUCT)
          (DECLARE (XARGS :GUARD (WEAK-STRUCT-P STRUCT)))
          (CAR (CDR STRUCT)))
   (DEFUN STRUCT-O (STRUCT)
          (DECLARE (XARGS :GUARD (WEAK-STRUCT-P STRUCT)))
          (CAR (CDR (CDR STRUCT))))
   (DEFUN STRUCT-S (STRUCT)
          (DECLARE (XARGS :GUARD (WEAK-STRUCT-P STRUCT)))
          (CAR (CDR (CDR (CDR STRUCT)))))
   "
;  This is the predicate, which contains the weak predicate and every
;  assertion made about the slots of the structure.  The final T guarantees
;  that all DEFSTRUCTURE predicates are Boolean.
 "
   (DEFUN STRUCT-P (STRUCT)
          (DECLARE (XARGS :GUARD T :VERIFY-GUARDS NIL))
          (AND (WEAK-STRUCT-P STRUCT)
               (PRIMEP (STRUCT-N STRUCT))
               (STRINGP (STRUCT-S STRUCT))
               T))
```

```
  "
;  This lemma shows that the predicate includes the weak predicate, as
;  :REWRITE, :FORWARD-CHAINING, and :BUILT-IN-CLAUSE rules.  Note that the
;  :REWRITE rule is sometimes implicated in thrashing in conjunction with the
;  normalization lemmas.
 "
  (DEFTHM DEFS-STRUCT-P-INCLUDES-WEAK-STRUCT-P
          (IMPLIES (STRUCT-P STRUCT)
                   (WEAK-STRUCT-P STRUCT))
          :RULE-CLASSES
          (:FORWARD-CHAINING :REWRITE :BUILT-IN-CLAUSE))
  "
;  This lemma rewrites the predicate on an explicit reference of
;  the constructor.
 "
  (DEFTHM DEFS-STRUCT-P-STRUCT
          (EQUAL (STRUCT-P (STRUCT N O S))
                 (AND (PRIMEP N) (STRINGP S) T)))
  "
;  This is the keyword constructor macro.  It will expand into a call of the
;  constructor, with appropriate defaulting.
 "
  (DEFMACRO MAKE-STRUCT
            (&WHOLE STRUCTURES::FORM &REST STRUCTURES::ARGS)
            (STRUCTURES::KEYWORD-CONSTRUCTOR-FN STRUCTURES::FORM
                                                STRUCTURES::ARGS 'STRUCT
                                                'MAKE-STRUCT
                                                '((:N) (:O) (:S))
                                                '(:N :O :S)
                                                '(:N :O :S)))
  "
;  This is the macro that provides for updates of multiple slots of a
;  structure.
 "
  (DEFMACRO
       UPDATE-STRUCT
       (&WHOLE STRUCTURES::FORM STRUCTURES::STRUCT
               &REST STRUCTURES::ARGS)
       (STRUCTURES::KEYWORD-UPDATER-FN STRUCTURES::FORM STRUCTURES::STRUCT
                                       STRUCTURES::ARGS 'STRUCT
                                       'UPDATE-STRUCT
                                       '(:N :O :S)
                                       'NIL
                                       ':COPY
                                       '(STRUCT N O S)
                                       '((:N . STRUCT-N)
                                         (:O . STRUCT-O)
                                         (:S . STRUCT-S))
                                       '((:N) (:O) (:S))))
```

```
   "
;  This lemma simplifies reads of an explicit constructor.
 "
   (DEFTHM DEFS-READ-STRUCT
           (AND (EQUAL (STRUCT-N (STRUCT N O S)) N)
                (EQUAL (STRUCT-O (STRUCT N O S)) O)
                (EQUAL (STRUCT-S (STRUCT N O S)) S)))
   "
;  This lemma lifts IF through calls of the slot accessors.
 "
   (DEFTHM DEFS-STRUCT-LIFT-IF
           (AND (EQUAL (STRUCT-N (IF STRUCT-TEST STRUCT-LEFT STRUCT-RIGHT))
                       (IF STRUCT-TEST (STRUCT-N STRUCT-LEFT)
                           (STRUCT-N STRUCT-RIGHT)))
                (EQUAL (STRUCT-O (IF STRUCT-TEST STRUCT-LEFT STRUCT-RIGHT))
                       (IF STRUCT-TEST (STRUCT-O STRUCT-LEFT)
                           (STRUCT-O STRUCT-RIGHT)))
                (EQUAL (STRUCT-S (IF STRUCT-TEST STRUCT-LEFT STRUCT-RIGHT))
                       (IF STRUCT-TEST (STRUCT-S STRUCT-LEFT)
                           (STRUCT-S STRUCT-RIGHT)))))
   "
;  This is the :ELIM lemma for the constructor.
 "
   (DEFTHM DEFS-ELIMINATE-STRUCT
           (IMPLIES (WEAK-STRUCT-P STRUCT)
                    (EQUAL (STRUCT (STRUCT-N STRUCT)
                                   (STRUCT-O STRUCT)
                                   (STRUCT-S STRUCT))
                           STRUCT))
           :RULE-CLASSES (:REWRITE :ELIM))
   "
;  This theory consists of all :DEFINITION runes associated with the
;  constructor, predicates, and slot readers/writers.  Only the
;  :TYPE-PRESCRIPTIONS and :EXECUTABLE-COUNTERPARTS remain ENABLEd.
 "
   (DEFTHEORY DEFS-STRUCT-DEFINITION-THEORY
              '(STRUCT WEAK-STRUCT-P
                       STRUCT-P STRUCT-N STRUCT-O STRUCT-S))
   (IN-THEORY (DISABLE DEFS-STRUCT-DEFINITION-THEORY)))
  (STRUCTURES::CAPSULE
   (LOCAL (IN-THEORY (ENABLE STRUCT-P)))
   "
;  The guard verification for the predicate is performed here since it may
;  need the current environment.  If it does not prove then you may need some
;  hints.  Any :PREDICATE-GUARD-HINTS option to DEFSTRUCTURE will be attached
;  to this lemma.
 "
   (VERIFY-GUARDS STRUCT-P)
```

```
  "
; This lemma captures all assertions about the structure.  This lemma is not
; guaranteed to prove.  If it does not prove than you may have to provide
; some :HINTS.  Any :ASSERTION-LEMMA-HINTS option to DEFSTRUCTURE will be
; attached to this lemma.  Be sure that you have not specified
; unsatisfiable assertions.
 "
  (DEFTHM DEFS-STRUCT-ASSERTIONS
          (IMPLIES (STRUCT-P STRUCT)
                   (AND (WEAK-STRUCT-P STRUCT)
                        (PRIMEP (STRUCT-N STRUCT))
                        (STRINGP (STRUCT-S STRUCT))
                        T))
          :RULE-CLASSES
          ((:REWRITE :COROLLARY
                     (IMPLIES (STRUCT-P STRUCT)
                              (PRIMEP (STRUCT-N STRUCT))))
           (:TYPE-PRESCRIPTION :COROLLARY
                               (IMPLIES (STRUCT-P STRUCT)
                                        (AND (INTEGERP (STRUCT-N STRUCT))
                                             (< 0 (STRUCT-N STRUCT)))))
           (:REWRITE :COROLLARY
                     (IMPLIES (STRUCT-P STRUCT)
                              (STRINGP (STRUCT-S STRUCT))))))
  "
; This theory lists every lemma generated by this DEFSTRUCTURE.  These are
; normally to remain ENABLEd.
 "
  (DEFTHEORY
       DEFS-STRUCT-LEMMA-THEORY
       '(DEFS-ACL2-COUNT-STRUCT DEFS-STRUCT-ASSERTIONS
                                DEFS-ELIMINATE-STRUCT
                                DEFS-STRUCT-LIFT-IF DEFS-STRUCT-P-STRUCT
                                DEFS-STRUCT-P-INCLUDES-WEAK-STRUCT-P
                                DEFS-READ-STRUCT
                                DEFS-WEAK-STRUCT-P-STRUCT))))
```